Shared Memory Computing on Networks of SMPS

Zhenghua Li

Abstract

Parallel systems supporting a shared memory programming interface have been implemented both in software and hardware. Hardware shared memory systems are fast, but generally expensive. Software shared memory systems are cheaper but slower. A clustered software shared memory system on a network of symmetric multiprocessors (SMP) is a hybrid solution to implementing shared memory with good performance at a reasonable cost. Within one SMP node, the hardware supports fine grained cache-coherence without any software intervention. Across nodes, a shared memory abstraction is implemented in software. Such systems are cheaper than pure hardware shared memory machines and more efficient than pure software shared memory systems.

We built a software distributed shared memory system for clusters of SMP machines and designed a unified fork-join programming model for our system. Our model allows users to take advantage of the physically shared memory within each SMP machine while preserving a uniform interface for both inter-node and intra-node parallelization. We also developed a front-end compiler that provides a loop annotation interface to the user and automatically generates parallel programs in the unified fork-join model. Applications written in our model have performance comparable with the handwritten code. Using the local hardware shared memory in our system results in better performance when comparing with pure software distributed shared memory system because of reduced inter-node communication.

1 Introduction

Parallel computing has gained more and more attention recently. Computation intensive applications that used to run for a long time on uniprocessor machines can finish in a much shorter time with enough hardware and good parallel algorithms.

Shared memory is an attractive parallel programming model. This model provides programmers with a global shared memory abstraction such that the programmer does not need to worry about where data is actually located. This shared memory abstraction has been implemented in two ways: software and hardware. Hardware shared memory systems connect processors with special-purpose inter-connects that provide fast inter-process communication. In such systems, the main memory may be centralized or distributed among individual processors, and the memory coherence is supported in hardware. Large-scale hardware shared memory machines are normally very expensive because of the complexity of the hardware. An alternative is to implement the shared memory abstraction in software. In this approach, off-the-shelf machines are connected by a general-purpose network and the global shared memory is simulated in software. Such systems are cheaper than hardware shared memory machines but suffer from the slow network connection and the overhead of the software coherence mechanisms.

A hybrid approach to implementing shared memory is a clustered software distributed shared memory system built on a network of small to medium-scale shared memory multiprocessor machines [8, 10, 13]. Such systems give high performance at low costs. Within one machine, hardware supports cache coherence. Across machines, a software layer provides the global shared memory abstraction. Such systems are much cheaper to build than using a pure hardware system and more efficient than a pure software system. They can benefit from both hardware and software shared memory. Local physically shared memory allows intra-node data sharing without any software protocol intervention, reducing the software coherence messages to only the cross node level.

In this paper, we built a software distributed shared memory for clusters of SMP machines, and designed a unified fork-join programming model for our system. This model allows programmers to describe parallel computation within and across SMP nodes using a unified interface. In addition, we developed a source-to-source compiler that automatically generates parallel programs in our unified fork-join model given loop annotated sequential programs.

The inter-node shared memory is based on TreadMarks [16], a software shared memory system developed at Rice University. Four applications were used to demonstrate the performance of our hybrid shared memory system. We present performance results on a cluster of Alpha multiprocessors connected by a Memory Channel [11] network. We compare the performance of the compiler generated fork-join style programs with hand-coded TreadMarks programs. The performance results show that the compiler generated fork-join programs perform comparably to hand-written programs. We also analyze the clustering effect on each application: the results show that all applications benefit a great deal from clustering because there are fewer software messages and the hardware coherence protocols are faster.

The rest of this paper is organized as follows. In Section 2 we describe the background of shared memory systems and the motivation for the clustered shared memory system. Section 3 discusses our system design. We describe our implementation of the system in Section 4, and present performance results in Section 5. Section 6 discusses related research work and we conclude in Section 7.

2 Background

A shared memory system must guarantee memory coherence, that is, processors must read the most recent value written.[1] This guarantee can be implemented in either hardware or software.

The following sections discuss hardware and software shared memory implementations and motivate clustered software shared memory.

2.1 Hardware Shared Memory

In hardware shared memory systems, processors are connected by dedicated interconnection networks that support fast inter-processor communication. The main memory is physically shared by all processors, i.e., processors can access any part of the memory directly. The memory coherence is supported in hardware. With a small number of processors, a bus is normally used as the interconnection network. Cache coherence in such machines can be implemented using either invalidate or update protocols [2]. Whenever a processor writes to a particular location, an update or invalidate message is broadcast on the bus. Other processors will do the corresponding update or invalidate upon receiving the message. With more processors, a centralized bus becomes a bottleneck. To scale shared memory machines to a large number of processors, more complex mesh or crossbar networks are used to relieve contention. Such systems require cache controllers to implement complicated cache coherence protocols like directory-based protocols. Moreover, large-scale shared memory machines are normally very expensive.

[1]This is sequential consistency, for more information, refer to [1]

2.2 Software Shared Memory

Software implemented shared memory systems [17] use off-the-shelf machines connected by general purpose networks and implement a global shared memory abstraction all in software. The software makes the collection of machines appear as if they were one shared memory parallel machine although nothing is shared among all the machines. The system structure is shown in Figure 1. Such systems are much cheaper than their hardware counterparts. They scale conceptually to a large number of processors without any modification to the system. Since no physical memory is actually shared among processors in such systems, all communication between processes is performed by sending and receiving messages over the network. The general purpose network connecting all the machines in a software shared memory system is generally slower than the special purpose interconnection network in a hardware shared memory machine.

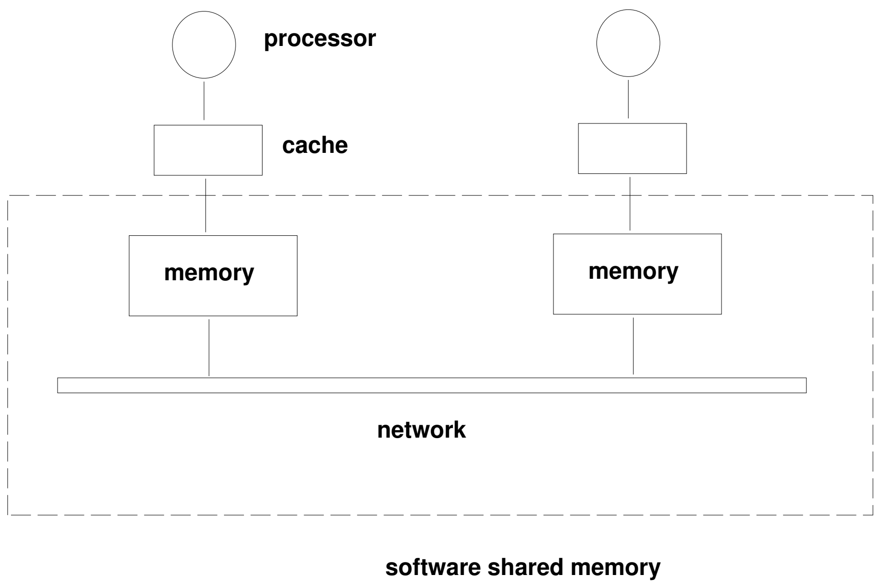

Figure 1. Software Shared Memory

One implementation of the software shared memory system is the shared virtual memory (SVM) system [17, 14], which uses the virtual memory hardware to detect accesses to non-local shared data. These systems communicate data and maintain coherence at the granularity of virtual pages. When two processes are modifying different parts of a single page, the page is communicated between these two processes back and forth for coherence reasons. This scenario is called false sharing. Because of the large data sharing units, SVM systems suffer from false sharing when applications have fine-grained sharing pattern.

Other software shared memory systems [3, 8] rely on the compiler and the runtime system to detect writes entirely in software at user-level. No virtual memory support is required. Memory store operations are instrumented to check shared data accesses. The shared data is made consistent at synchronization points. This approach has low overhead on individual shared data misses and supports fine-grained sharing since the unit of coherence is not tied to the virtual memory page size. The potential disadvantage of such systems is that the miss detection overhead occurs on every write. With SVM systems, a page fault occurs only at the first access to an invalid page. Further accesses to the same page can proceed without faulting anymore, thus amortizing the initial overhead.

2.3 Clustered Software Shared Memory

Recently, researchers have proposed using a cluster of small-scale symmetric multiprocessor (SMP) machines as a hybrid approach to implementing shared memory [8, 10, 13]. In a symmetric multiprocessor machine, the physically shared memory is accessible to all processors at an equal distance. A clustered software shared memory system uses the hardware shared memory within one shared memory machine, and implements a shared memory abstraction across nodes in software. The system structure is shown in Figure 2. Such systems have the advantages of both software and hardware shared memory systems, and potentially balance the shortcomings of both systems. Next we discuss the potential advantages of this hybrid system.

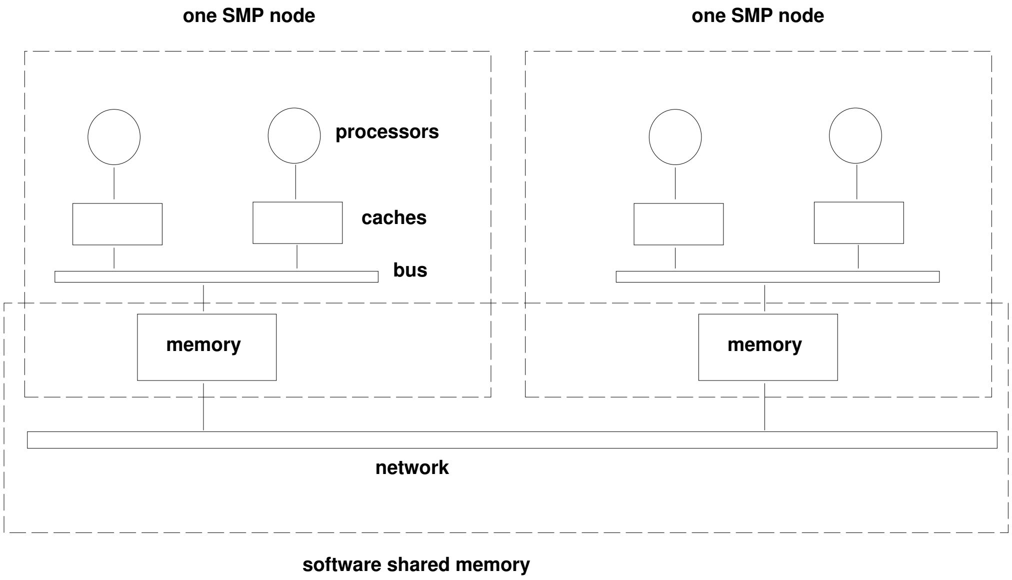

Figure 2. Clustered Software Shared Memory

The first and the most important advantage is that it is much cheaper than a pure hardware or software shared memory system. A network of small-scale bus-based SMP machines is much cheaper than a single shared memory machine with the same overall number of processors, which most likely would use more complicated interconnection networks and more complex cache-coherence protocols. It is also cheaper than using a network of single processor machines because less hardware is required.

Second, the physical shared memory within one SMP node allows intra-node application data sharing without software protocol intervention. Software intervention for misses along with explicit software messages can be limited to the cross-node level. Remote data fetched by one processor may be readily used by other processors on the same node without use of explicit messages. Applications that exhibit frequent nearest neighbor sharing are expected to run well on such systems since the common case of communication between neighbors is handled by the hardware shared memory.

Third, the system can benefit more if the application data as well as the software coherence data are all shared locally. Sharing the software coherence data makes it possible to filter requests for the same data from processors on the same node. If the home processor and the owner processor of some shared data are on the same node, the home processor can satisfy incoming remote requests without sending messages to the owner processor. If the requester and the home processor are within the same node, the requester can directly access the software coherence data without sending messages to the home processor.

Fifth, good load-balance is possible in such systems. If the software protocol data is shared locally, any processor within the same node can handle incoming cross machine messages. This is better than letting one processor handle all cross machine messages, which might result in load imbalance. The system also does not require extra dedicated processors just for the cross node communication purposes.

Finally, more choices of data sharing granularity are available on such systems. The system supports fine-grained sharing in hardware on local nodes without write detection overheads in software fine-grain shared memory systems. Across nodes, the flexibility of the software environment permits the implementation of different protocols and strategies.

3 System Design

In this section, we present the system architecture of our clustered shared memory and the programming model we support on top of it. We explain our design choices, and compare our system to other existing systems.

3.1 Exploiting Parallelism in Clustered Software Shared Memory

When designing a clustered software shared memory system, one needs to answer the following questions:

1. What is the unit of data sharing?
2. What should be shared through the physically shared memory on a local node?
3. Which processor will handle cross node software protocol requests?
4. Where should the software system be implemented?

Within one SMP node, the hardware provides fine-grained data coherence. The unit of sharing within one node is a cache line. Across nodes, a software shared memory system that supports either fine-grain or coarse-grain data sharing can be used. Fine-grained sharing eliminates the false sharing problem that occurs in page-based software shared memory systems whereas page-based systems amortize the cost of communication over the network by sending data in units of a page. The Shasta [8] system supports fine-grain software distributed shared memory on SMP clusters. Shasta implements a shared address space by inserting checks in an application at loads and stores. This inline code checks if the data is available locally and sends out the necessary messages to other processors if the operation cannot be serviced locally. Other studies [10] use page-based software shared memory for cross-node coherence.

The next two questions are related. We need to first decide what data we allow local processes to share through the physically shared memory. In software shared memory systems, two categories of data exist. One is the application data, the other is the software protocol data that is used to implement the software shared memory. To use the local physically shared memory efficiently, it is always advantageous to let local processes to share the application data. Then the question is narrowed down to whether the software protocol data should be shared or kept private.

Sharing both the application and the software protocol data not only limits the software protocol communication to the cross-node level, but also provides the possibility of letting any process on the same node handle protocol related requests. In addition, software messages can be eliminated when the owner and the requester of some data is located on the same node. Finally, the system can filter remote requests from multiple local processes for the same data and send only one request.

Sharing only the application data results in intra-node software messages for the software coherence protocol. When one process changes the sharing status of some shared data, software messages have to be sent not only to remote processes, but also to other local processes. Moreover, the system has to statically designate one or more processes to handle cross node requests. The Shasta system only allows local processes to share the application data. In their system, the directory that keeps track of the shared memory status is not shared among local processes. The system maintains a private state table for each process to keep track of which part of the shared memory each process has accessed. Based on these tables, the process that handles incoming requests will send 'downgrade' messages only to processes that have actually accessed the shared data.

Statically designating one or more processors for inter-node communication may result in load imbalance. To solve this problem, people have tried using dedicated processor(s) for software protocol handling. Karlsson et al. [13] found that it does not pay off to assign software protocol execution to a dedicated coherence protocol processor, because in a clustered environment the likelihood is high that spare cycles on a compute processor can be used. In their experiment, one additional processor is used for protocol handling. Erlichson et al. found that in SoftFlash, using dedicated software protocol handling processors improves the performance of the applications, but not proportionally with the number of extra processors used. They used five dedicated processors for protocol handling for every four processors involved in computation, resulting in performance improvement of up to 20%. The Shasta system relieves the load imbalance problem by letting the process that handles the last downgrade message execute the protocol action associated with the request, including updating the shared state and sending the reply.

Implementing the system in user space requires no changes to the operating system. But kernel space implementations may provide better performance. Karlsson et al. implemented their system on top of TreadMarks, which is implemented entirely in the user space. Shasta also does not require any operating system support. SoftFlash implemented the whole system in the kernel space. Coherence faults are taken by the standard virtual fault handlers inside the kernel and passed directly to the SoftFlash handlers, permitting rapid dispatch of the coherence violation without the long delays normally associated with running fault handlers in user space via the Unix signal mechanism.

3.2 Our Design Choices

We used a page-based SVM system to implement the cross-node shared memory abstraction. Since fine grained sharing is already available within local nodes, coarse grained cross-node sharing will have an aggregation effect on the data communication. Instead of sending many small data messages across nodes like in the Shasta system, a page-based system will communicate a smaller number of larger messages, thus amortizing the high per-message overhead of the general purpose network. Moreover, the false sharing problem in our page-based system is only limited to true remote accesses.

Our system allows local processes to share the whole address space to take full advantage of the local physically shared memory. There are several ways to achieve this. Some systems provide mechanisms for users to create a group of processes in which all processes in the same group share the whole virtual address space. Messages can either be sent to a particular process in a group, or can be sent to all processes in a group. But such mechanisms would require us to statically decide which process will handle the inter-node communication. Another way is to use light-weight processes or threads for multiple contexts to share the address space. Threads share signals, which automatically makes the cross-node message handling randomly distributed. We choose to use threads as our local contexts because of these advantages. In our system, TLB flushes are necessary when one thread changes the access privilege of the shared data. But this overhead is smaller than sending software broadcasting messages.

Our system is implemented entirely at the user level. Although more optimizations may be possible if the system is implemented at kernel level, having to change the operating system is impractical.

Table 1 compares our system with the SoftFlash and Shasta system.

	Our System	*SoftFlash*	*Shasta*
cross node data sharing	page based	page based	fine-grained
share software protocol data	yes	yes	no
load balancing	yes	no	somewhat
dedicated protocol handling processors	no	yes	no
kernel or user level	user	kernel	user
need compiler support	no	no	yes

Table 1. Comparison of different systems

3.3 Programming Model

There are two commonly used programming models in shared memory systems, fork-join and Single Program Multiple Data (SPMD). In the fork-join model, a master process spawns subprocesses with a `fork`, and waits for them to finish with a `join`. In the SPMD model, each process runs the same program but executes different code depending on its process id or data in shared memory.

In the SPMD style of programming, multiple processes are usually created once at the beginning and processes exist until the end of the program. Sequential parts are executed by one process based on process id. All-to-all barrier synchronization is always inserted both at the start and the end of each parallel region.

In the pure fork-join style programming, sequential regions are always executed by one master process. The parallel computation starts when the master process does a `fork` and ends at a `join`. A `Fork` serves as a one-to-all synchronization and a `join` serves as a all-to-one synchronization.

The fork-join model is more modular than the SPMD model and a fork-join pair incurs less communication than two all-to-all barrier synchronizations. But depending on the system implementation, multiple calls to fork/join may be too expensive. Another disadvantage of the pure fork-join model is that the sequential work is always done by the master process. In the SPMD model, the sequential region can be allocated to any local process based on process id. This flexibility can benefit certain applications due to better data locality [7].

Our system supports the fork-join programming model since it is more modular than the SPMD model and the potential disadvantages of the fork-join model can be tackled by proper implementation.

In our hybrid software and hardware shared memory system, there are two levels of parallelism to express: inter and intra node. Supporting one model for one level of parallelization and a different model for another level of parallelization is going to be very messy. For instance, using the SPMD model for the shared memory abstraction across nodes and the fork-join model for the physically shared memory within one node is neither modular nor easy for the programmer to use. We developed a unified model for the two-level parallelism we support. Since we want to use threads on the local node and the thread library supports the fork-join model, it is easier to use a unified fork-join model for our system. In our system, it is hard to provide the SPMD model for both intra and inter node parallelization without having to require the user to do explicit thread manipulation.

We implemented the fork-join model in the SVM system we used, and we integrated it with the thread fork-join. Our unified fork-join model provides a clean and modular parallel interface for programmers to express the two level parallelism, making the shared memory programming on SMP clusters easy and clean. To write a shared memory program, a user just needs to write in the normal fork-join model, without having to explicitly handle the different levels of parallelism. Applications written in our model can either run on networks of single processor machines or on networks of SMPs.

To reduce the overhead of multiple calls to `fork` and `join` in our system, we implemented these two calls in a way such that child processes and threads are always created once at the beginning of the program. Later calls to `fork` will only wake up child processes and/or threads. Child processes and threads will synchronize with their parent on `join` calls and then wait until the parent signals them with another fork.

4 Implementation

This section discusses the implementation of our clustered shared memory system and the unified fork-join programming model. We also describe the front-end compiler we developed that gives programmers a simple loop annotation interface, and automatically generates parallel code in the unified fork-join model.

4.1 TreadMarks

TreadMarks is a page-based software shared memory system that is implemented entirely as a user-level library on top of Unix. For inter-machine communication, TreadMarks uses the Berkeley sockets interface. It uses the operating system's page protection mechanism to detect accesses to the virtual shared memory. To keep the shared virtual memory consistent, TreadMarks uses the lazy release consistency model [15] and a multiple writer protocol [16].

A memory consistency model determines when a process's update to the shared memory should be visible to the other processes. Various memory consistency models have been proposed. Release consistency model distinguishes all shared memory accesses into two categories: normal shared data accesses and synchronization operations. Synchronization operations are further divided into acquires and releases. An acquire precedes a series of shared memory accesses and a release signals the end of a series of shared memory accesses. Furthermore, release consistency only requires that modifications to the shared memory of one process be visible at the release. Lazy release consistency model goes one step further. The propagation of modifications is postponed until the time of the acquire.

A multiple writer protocol allows multiple processes to modify the same page concurrently, as long as they are not modifying the same data. This protocol gets rid of the ping-pong problem incurred by false sharing.

The implementation of the multiple writer protocol in TreadMarks uses twining and diffing. First, the execution of each process is divided into intervals which are periods between synchronization points. To record which process modifies which page, when a process writes to a shared page for the first time, the software system makes a copy of the page (twin). The twin and the current copy can be later on compared to create a diff, which is an encoding of the changes to the shared page. Whenever a process modifies a shared page, a write-notice is created to indicate that a particular page has been modified by some process at a certain interval. When a process p_1 synchronizes with another process p_2, they will exchange the list of write-notices and invalidate corresponding pages. A lazy diffing mechanism is used such that the diff creation is deferred until a subsequent request for the diff or until a write notice for the same page is received from another process. When a process wants to get an up-to-date view of a particular page, it will send requests to the previous modifiers and get all the relevant diffs and apply them in the right order.

4.2 Building the Unified Fork-join Model in TreadMarks

The original TreadMarks system supports the SPMD programming model. We added in function calls to support the fork-join model. Processes are still created only once at the beginning of the program. A master process is assigned to execute all the sequential work and schedule all the parallel computation. Parallel regions are encapsulated into different functions and the master uses the `Tmk_sched_fork` call to tell the slaves which function to call, addresses of shared data and enough information to do the computation partition. Slave processes will just spin wait, periodically time out, and send messages to the master process asking for work to do.

Using this programming model, the structure of the new TreadMarks program will be as shown in Figure 3.

The TreadMarks system has been modified to support multi-threading within one machine. Mutual exclusion among threads is added to various places where the TreadMarks system protocol data is being updated. When multiple threads fault on the same page, we want to make sure only one thread will send out requests and then do updates to the shared page. For instance, if thread t_1 faults on some shared page, it will either get the whole page or just the diff for this page from some other process. In order to do this, it needs to set the write permission for this shared page. If after the write permission for the page is set, another thread t_2 tries to modify the same page, then t_2 will incorrectly get the write permission to do the write. To prevent this scenario from happening, another shadow mapping is created for the shared memory region. Threads synchronize on lock acquire before they send out page and/or diff requests. Thread t_1 that gets the lock will only change the protection of the shadow mapping page, enabling itself to make updates to it, while other threads which fault on the same page will block at the lock acquire. When thread t_1 is done with the updates, it will change the protection of the main mapping to unblock other threads. When other threads get the unblocking signal, they will see the updated version of the page.

The original multi-threaded TreadMarks system exposes all the thread details to the programmer. The programmer has to do all the thread creation and thread synchronization at the application level. The SPMD model is used for inter-node parallelization and the fork-join model is used for intra-node thread level parallelization. We used a unified fork-join model for both cross-node and thread synchronization in our system. It enables us to include multi-threading and make all the thread operations transparent to the user. The TreadMarks library will create and manage threads instead of requiring the user to explicitly do the thread manipulation. All the user needs to do is to specify the number of threads when executing the program. The unified fork-join model is shown in Figure 4.

To reduce the overhead of creating threads many times when there are multiple fork/join calls in the application, we implemented the system such that threads are only created once. After the program starts, each node will create the right number of threads that the user requests. Within each node, we assign one master thread, just as we assign one master process before. The master thread will be responsible for work scheduling within one SMP node. Slave threads will wait on certain synchronization points until the master thread signals them.

In addition to fork/join synchronization, our system also provides barrier and lock synchronization. For barriers, each processor updates a local counter until the last processor on the node has reached the barrier. The last processor sends the arrival message to the barrier manager. When the last arrival message arrives at the manager, the manager issues a departure message to each node. Locks are implemented such that the cross node locking is enclosed by the thread level locking.

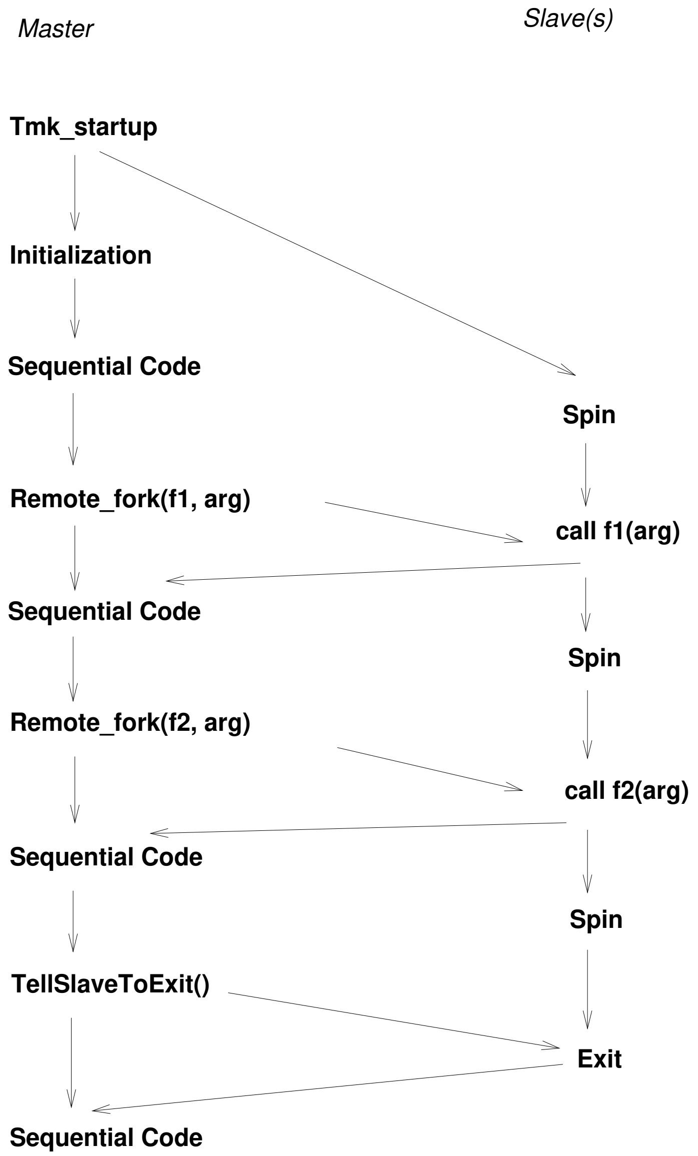

Figure 3. Fork-Join Model in TreadMarks

Figure 4. Unified Fork-join Model

4.3 Compiler Support for Shared Memory Programming

Compiling sequential programs with loop annotation directives to parallel code has been studied intensively. Loop annotation directives are used to specify which loop to parallelize and how to partition the loop iterations. Based on these directives, the compiler will generate parallel code in the form of message passing or shared memory. Studies have shown that software shared memory is a promising target for parallelizing applications [7]. With the compiler support, writing a shared memory parallel program is simplified to just inserting parallel directives to the sequential program.

In this paper, we developed a source-to-source compiler that generates fork-join style shared memory programs from sequential programs with loop annotations. We provide an interface similar to SGI Power-C [19] interface to the programmer, and our compiler generates TreadMarks shared memory program in the unified fork-join model.

4.3.1 SGI Power-C

SGI Power-C is targeted for parallel programming on shared memory multiprocessor machines. The strategy of the Power-C language is to introduce parallelism into the serial code without jeopardizing the serial execution. This is accomplished by introducing compiler directives masked as C language *pragma*s into the serial code.

The Power-C programming model is based on parallel loops and shared variables. Users can define a parallel region which can consist of different kinds of work-sharing constructs. A parallel region is started by the *parallel* pragma. This pragma has a number of modifiers for users to specify the data sharing pattern, the number of threads required, the scheduling method and also the parallelization condition.

Variables inside a parallel region can be one of the following three types:

- shared
  Every process can read and modify the variable.
- by value
  Every process can read the variable, but nobody can change it.
- private
  Every process has its own copy of the variable, modifications will be local.

The `if` modifier allows the user to set up a condition that is evaluated at run time to determine whether or not to run the statements inside the parallel region serially or in parallel.

The constructs inside a parallel region can be

- parallel loop
  (`#pragma pfor iterate...`)
- independent block
  (`#pragma independent`)
- local code executed by all processes
- code executed by only one process
  (`#pragma one processor`)

Users can also specify the schedule type of a particular parallel loop to be one of the following:

- simple
  Each process is given an equal number of iterations of the loop.
- interleave
  Each process will work on interleaved iterations of the loop.
- dynamic
  Each process grabs a fixed number of iterations dynamically from the statically partitioned work pool.
- runtime
  The schedule type is selected by the MP_SCHEDTYPE environment variable.
- gss (guided self-scheduling)
  The run time scheduler give each process a chunck of iterations of the loop. The chunksize iterations begin with big pieces and end with small pieces.

Suppose we want to compute the sum of two arrays and the maximum value of the resulting array. We can write this in Power-C as shown below.

```
int A[100], B[100], C[100];
int max = 0;
int n = 100;
int want_parallel;

#pragma parallel shared(A, B, C, max)
#pragma byvalue(n)
#pragma if(want_parallel == 1)
 {
    #pragma pfor iterate(i=0; n; 1)
    #pragma schedtype(simple)
     {
        for(i = 0; i < n; i++)
         {
            a[i] = b[i] + c[i];
            #pragma critical
            if(a[i] > max){
               max = a[i];
            }
         }
     }
 }
```

The code computes the sum of two arrays B and C and the maximum element of the resulting array A. The user defines arrays A, B, and C to be shared by all processes; the maximum value is also defined as shared. The main computation is the `pfor` loop. The compiler will give each process an equal block of of arrays to work on since the schedule type is specified as `simple`. To make sure that at one time, only one process can modify the max value, a critical section is necessary. The parallel region will only be executed in parallel if the `want_parallel` flag is set at run time, otherwise the sequential code without the pragmas runs on one processor.

4.3.2 Compiling from SGI Power-C to TreadMarks Program

In order to generate correct and efficient TreadMarks code from Power-C programs, we introduced new directives for users to specify the start of the major parallel working region and shared data allocation directives.

- #pragma rfork_start
  This directive specifies that the main parallel working region starts afterwards. Since the TreadMarks system requires the `getopt` operation to interprete command line arguments, our compiler uses this directive to decide where to insert the `getopt` operation.

- #pragma shared_malloc
  This directive will enable the compiler to insert the shared data allocation statements into the generated code. Since it is very complicated to detect shared data access and generate the shared data allocation automatically, our compiler requires user to explicitly tell what data will be shared by this directive.

We modified the front end of GCC compiler, made it accept Power-C programs, and generate TreadMarks programs after some necessary analysis. During the parsing of the Power-C program, parallel regions are encapsulated into different functions, and the necessary TreadMarks library calls are generated.

When the compiler sees a `#pragma parallel` directive, it will put the following block that contains the whole parallel region into a new function and replace the block with a `Tmk_sched_fork` call with the new function pointer as its argument. Also, a `Tmk_sched_join` call is inserted after the `Tmk_sched_fork` call for synchronization purposes. From the parallel modifiers, the compiler would know which variables in the parallel region are shared, byvalue or local. Shared variables and byvalue variables will be passed to the new function while local variables will be declared in the new function. Other synchronization directives inside the parallel region will be replaced by corresponding TreadMarks library calls: `#pragma synchronization` is replaced by `Tmk_barrier`, `#pragma critical` is replaced by a `Tmk_lock_acquire` and `Tmk_lock_release` pair.

For the Power-C sample program shown before, the TreadMarks code that will be generated by the compiler will be:

```
int *A, *B, *C, *max;
int n = 100;
int begin, end;

Tmk_startup(argc, argv);

/* master process allocates the shared arrays */

if(Tmk_sched_start(0) == 0){
    A = (int *) Tmk_malloc(n*sizeof(int));
    B = (int *) Tmk_malloc(n*sizeof(int));
    C = (int *) Tmk_malloc(n*sizeof(int));
    max = (int *) Tmk_malloc(sizeof(int));

    {
```

```
            struct Tmk_sched_arg tmk_arg;
            tmk_arg.func = tmk_func_0;
            tmk_arg.shared[0] = A;
            tmk_arg.shared[1] = B;
            tmk_arg.shared[2] = C;
            tmk_arg.shared[3] = max;

            Tmk_sched_fork( &tmk_arg);
            Tmk_sched_join();
        }

        later_computation_starts();
    }

    Tmk_exit(0);
```

The main computation for each process is carried out in the generated function `tmk_func_0`.

```
    void tmk_func_0( thread_arg)
         void  *thread_arg;
    {
       struct Tmk_sched_arg *tmk_arg = (Tmk_thread_arg_t)thread_arg->tmkArg;
       int *A = tmk_arg->shared[0];
       int *B = tmk_arg->shared[1];
       int *C = tmk_arg->shared[2];
       int *max = tmk_arg->shared[3];
       int begin, end;
       int thr_id = NUM_THREADS*Tmk_proc_id
                    + ((Tmk_thread_arg_t)__threadArg)->id;
       int nthrs = NUM_THREADS*Tmk_nprocs;

       begin = thr_id * n/nthrs;
       end = (thr_id + 1) * n/nthrs;

       for ( i = begin; i < end; i++){
          A[i] = B[i] + C[i];

          if(A[i] > *max){
            Tmk_lock_acquire(0);
              *max = A[i];
            Tmk_lock_release(0);
          }
    }
```

5 Performance Results

We developed Power-C programs for 4 applications, namely MGS (Modified Gramm-Schmidt), Red-Black SOR (Successive Over-Relaxation), TSP (Traveling Salesman Problem) and ILINK. Then we used our front-end compiler to generate the TreadMarks fork-join style code. We compare the performance of the compiler generated code with the handwritten TreadMarks SPMD code on a SMP cluster of 3 Alpha machines connected by a Memory Channel network. Each Alpha machine has 4 processors. We also analyze the clustering effect on all applications.

Program	Problem Size	Time(sec)
GS	1024x1024	43.03
SOR	2000x1000, 100 iterations	27.50
TSP	19 city	26.50
ILINK	CLP 2x5x4	20.44

Table 2. Data Set Sizes and Sequential Execution Time of Applications

Table 2 presents the data set size as well as the sequential execution time for each application. Sequential execution times are obtained by removing all synchronization from the hand-coded TreadMarks programs and executing them on a single processor. These times are used as the basis for all the speedup figures reported in this paper.

5.1 Applications

5.1.1 MGS

This application computes an orthornormal basis for a set of N-dimensional vectors. At each iteration i, the algorithm first sequentially normalizes the ith vector, then makes all vectors $j > i$ orthogonal to vector i. The Power-C version of MGS puts the inner loop in one parallel region and a synchronization statement at the end of the inner loop. In the hand written code, two barriers are necessary at the start and the end of the inner loop for correct synchronization: the first barrier is to make sure that every process will have an up-to-date value of the normalized ith vector and the second barrier is to make sure that the new values of all modified vectors are propagated to each process for the next iteration.

5.1.2 Red Black SOR

This application solves a partial differential equation. The main computation iterates over a two dimensional array. Each iteration updates every array element to the average of its 4 neighbors, i.e., above, below, left and right. Every element is painted either black or red such that all the four neighbors of a red element are black and vice versa. Each iteration is divided into two phases. The first phase updates all red elements based on the values of their black neighbors. The second phase updates all black elements based on their red neighbors. The Power-C version of SOR puts the whole computation in one parallel region and inserts two synchronization directives at the end of each phase. The handwritten TreadMarks version divides all the rows of the array into equal number of continuous rows to each processor. Two barriers are used at the end of each phase for process synchronization.

5.1.3 TSP

The Traveling Salesman Problem solves the problem of finding the shortest path that starts at a designated node, passes through every other node exactly once and returns to the originated node. The branch-and-bound algorithm is applied to this problem. If the length of a partial tour plus a lower bound of the remaining portion of the path is longer than the current shortest path, the partial tour will be discarded since there is no point in exploring it further. A priority queue is used to keep all the partial tours with the shortest one at the head. The shortest one is fetched from the head of the queue and is either solved or partitioned into more partial tours and inserted to the queue. The Power-C program puts the whole computation into one parallel region. A global queue is used to keep all the partial tours and a global tour length records the best tour computed so far. Lock synchronization is used to protect accesses to the global queue and the best tour length. The handwritten TreadMarks code uses the same strategy.

5.1.4 ILINK

ILINK is a program from the LINKAGE [4, 5] package for genetic linkage analysis which tries to find the approximate location of disease genes. ILINK takes as input one or more family trees showing the phenotypes at some genes and the disease status for those individuals where it is known. It iteratively computes a vector which represents a maximum likelihood estimate of how close the disease gene lies to the other specified genes on the genome. Given a set of pedigrees and fixed values of the probability, the outer loop of the likelihood evaluation iterates over all the pedigrees calculating the likelihood of each one. Within a pedigree, the program visits each nuclear family and updates the probability of each genotype for each individual. The main structure that gets updated is a sparse array associated with each individual. In the handwritten TreadMarks version [9], the update to each individual's probability is parallelized. A master process stripes the sparse array for each individual process to do the update. The Power-C version uses the same strategy and puts the update function in a parallel region. Inside the parallel region, each thread will update different parts of the sparse array based on their thread ids.

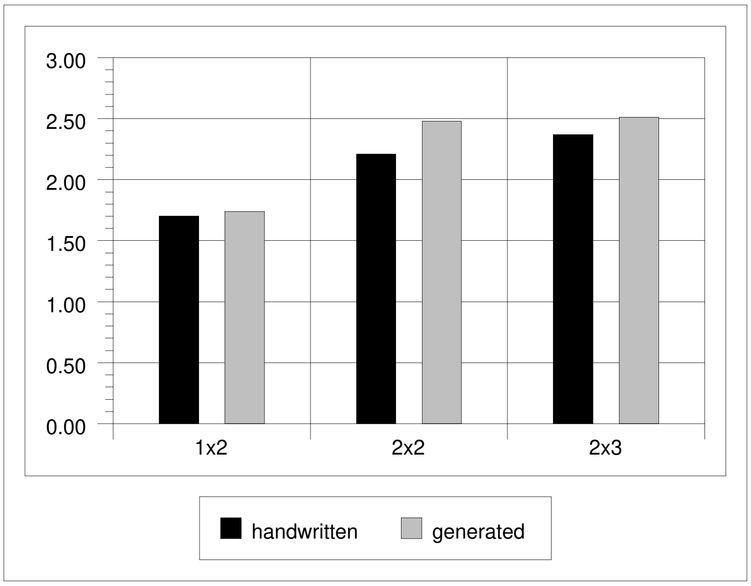

Figure 5. MGS – speedup

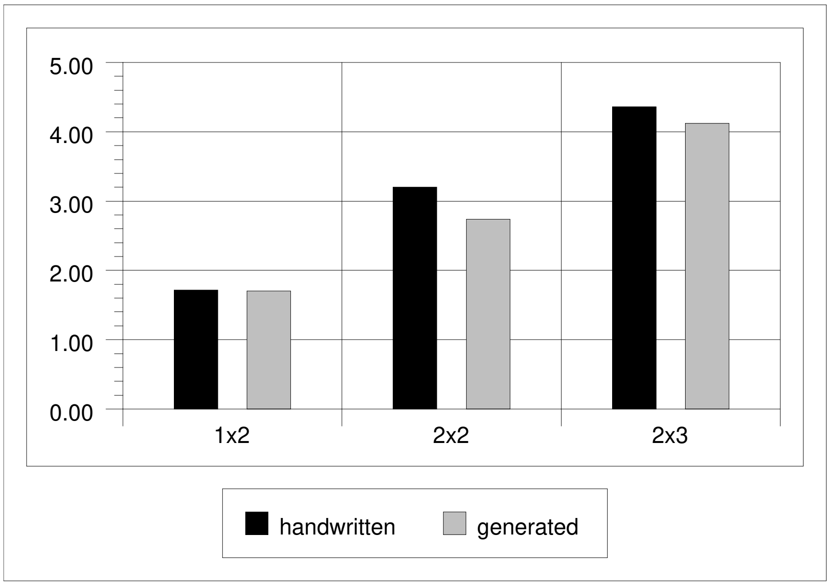

Figure 6. SOR – speedup

Applications	GENERATED	HANDWRITTEN
GS	69000	68989
SOR	11608	10220
TSP	14926	14871
ILINK	23924	28563

Table 3. Generated vs. Handwritten 4x2 - Data Communication(bytes)

5.2 Generated Code vs. Handwritten Code

Next we compare the performance of compiler generated fork-join code with the handwritten TreadMarks SPMD code for all applications.

5.2.1 MGS

The generated code performs a little bit slower than the handwritten TreadMarks program. This is because of the extra synchronization introduced by the compiler. In the handwritten code, two barriers are used at the start and the end of the inner loop for correct synchronization, the first barrier is to make sure that every process has an up-to-date value of the normalized ith vector and the second barrier is to make sure the new values of each vector are propagated to each process for the next iteration. The Power-C version of GS puts the inner loop in one parallel region. Inside the parallel region, at the end of the inner loop, a barrier synchronization is necessary. The transformed code just encapsulates the inner loop into a function and replaces the inner loop with a `Tmk_sched_fork` and a `Tmk_sched_join` call. The barrier synchronization at the end of inner loop in the Power-C code is not necessary any more, but it is still transformed to a barrier call in the generated code. This extra synchronization accounts for the slight performance degradation and extra messages communicated. The speedup is shown in Figure 5. In Lu's paper [7], her comparison was based on different programs in which a different cyclic algorithm

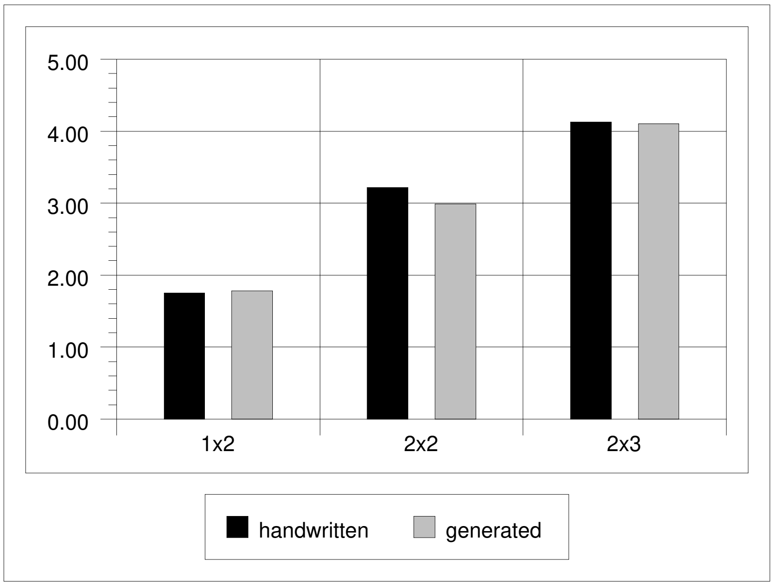

Figure 7. TSP – speedup

is used for scheduling loops. Since the Power-C language does not provide cyclic scheduling, we used simple block scheduling for our experiments.

5.2.2 Red Black SOR

For SOR, the generated code performs a little bit slower than the handwritten TreadMarks program and sends more messages because of extra synchronization incurred by the transformation from Power-C code. Inside the parallel region of the Power-C code, there is one barrier at the end of each phase to keep the shared data synchronized. The TreadMarks code generated from this Power-C code will contain not only these two barriers, but also the `Tmk_sched_join` call at the end of each iteration. The second barrier call is unnecessary because of the `Tmk_sched_join` call. The extra barrier synchronization accounts for an increase of messages in the generated code for up to 13%.The speedup is shown in Figure 6.

5.2.3 TSP

For TSP, the generated code performs almost the same as the the handwritten TreadMarks code. The amount of communication is also about the same. The main computation is carried out in routine Worker. At the start and the end of the routine, there are two barrier to synchronize every process. The generated code uses fork-join model, making the two barrier synchronizations unnecessary. These extra barriers account for the extra number of messages sent by the generated version.The speedup is shown in Figure 7.

5.2.4 ILINK

For ILINK, the generated code gets rid of several synchronization points by using the fork-join model. In ILINK, the main work distribution is done by the master process. All child processes run in a loop and wait for certain flags to decide which routine to call. Two main shared flags are used to synchronize the master with the slave processes. A done flag is used for the master to tell the slaves to exit. A travel

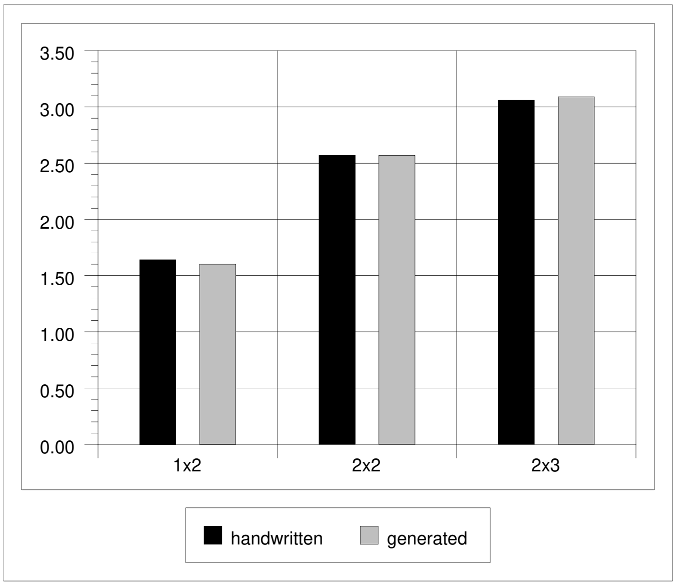

Figure 8. ILINK – speedup

flag is used to tell slave processes which routine to call. Every child process calls a barrier at the start of the loop to decide if the done flag is set. If the done flag is not set, child processes will compute some local information and then call barrier to check the travel flag to decide which routine to call. Child processes exit the loop when the done flag is set. In the generated code, the two barriers are replaced by a `Tmk_sched_fork` and `Tmk_sched_join` call. The master process will send the two flags with the `Tmk_sched_fork` call and upon receiving the message, the child processes would get the flags and decide which routine to call. In this way, the handwritten version sends up to 19% more messages.The speedup is shown in Figure 8.

5.3 Analysis of Clustering Effects

Next we analyze the benefits of using the local physically shared memory, i.e., the clustering effect of our system. We compare the results of using clustering of 4 processors vs. no clustering on 2 nodes. For all the applications we run, clustering reduces the running time and the amount of communication significantly. The data is shown in table 4. With clustering, performance increases by 26% and 45% for TSP and SOR respectively. As for communication, using clustering reduced the communication down to only 10% of the communication when no clustering is used.

To support clustering, thread mutual exclusion has been added to various places in TreadMarks to make the library thread safe. When no clustering of processors is used, the extra thread synchronization accounts up to 10% performance degradation when comparing with the single threaded version. We compared the multi-threaded with the single threaded version of all the applications for both the handwritten code and the generated code on one processor.

First we talk about how multi-threading affects the handwritten code. For GS and SOR, multi-threading added a significant overhead. By using 3 nodes, the multi-threaded GS runs 22% slower than the single threaded version, while for SOR it shows a 13% performance decrease. For TSP, multi-threading does not seem to add very much overhead. For ILINK, on 3 nodes, multi-threaded version is 9% slower than the single threaded version. We attribute these different performance results to

Applications	TIME (SEC)		BYTES (K)		MESSAGES	
	w/ cluster	w/o cluster	w/ cluster	w/o cluster	w/ cluster	w/o cluster
GS	18.05	26.20	16	161	7487	54732
SOR	4.96	8.99	4	9	2409	11608
TSP	4.34	5.89	0.6	5	3897	14926
ILINK	5.04	7.36	3	25	2605	23924

Table 4. Clustering vs. No Clustering 4x2

different synchronization patterns of the applications. For GS, processes/threads call barrier 1024 times over the iterations of the main loop, while SOR makes 200 and ILINK makes 423 calls. These barrier synchronizations turn out to be very costly with multi-threading. This is because thread locks and unlocks are called upon entering and exiting a barrier. During a barrier synchronization, messages are sent back and forth between the barrier manager and slaves. Upon receiving messages, the signal handler will also call thread locks and unlocks to process the messages. This overhead turned out to be significant for applications that call barriers frequently. As for TSP, only 3 barrier calls are made during the whole execution time, thus the performance of the multi-threaded version is about the same as the single threaded version.

Table 5 compares the execution times of the handwritten code with and without multi-threading by running one process/thread on each node.

Applications	1 NODE		2 NODES		3 NODES	
	w/ thr	w/o thr	w/ thr	w/o thr	w/ thr	w/o thr
GS	43.50	43.08	29.69	25.31	23.63	19.43
SOR	27.22	27.55	16.15	14.53	11.10	9.82
TSP	25.83	26.50	14.51	14.50	10.52	10.03
ILINK	22.26	20.54	13.70	12.50	10.02	9.22

Table 5. Handwritten code w/ vs. w/o Multi-threading

As for the generated code, the same trend shows. From Table 6, the multi-threaded GS and SOR on 3 nodes performs 27% and 15% slower respectively. For ILINK, we get rid of a lot of barrier synchronizations and the result show that the multi-threaded version performs comparable to the single threaded version.

Figure 9 shows the overhead differences between handwritten code and generated code on 3 nodes.

5.4 Summary of Results

Our results show that the performance of the compiler generated code is comparable to that of the handwritten TreadMarks code. The fork-join synchronization model decreases the number of messages sent for ILINK. The source-to-source transformation introduces extra synchronization for GS, SOR and TSP, especially for SOR and GS, where an extra barrier call is made at every iteration. Clustering improves performance by reducing the running time and cross node communication significantly, although thread synchronization incurs overhead when no clustering is actually used. We expect to improve the

Applications	1 NODE		2 NODES		3 NODES	
	w/ thr	w/o thr	w/ thr	w/o thr	w/ thr	w/o thr
GS	43.43	43.11	30.24	25.60	23.99	19.32
SOR	27.31	28.00	15.10	13.54	10.53	9.18
TSP	26.01	26.82	14.93	14.49	10.66	10.42
ILINK	22.26	20.60	13.93	12.44	9.98	9.20

Table 6. Generated code w/ vs. w/o Multithreading

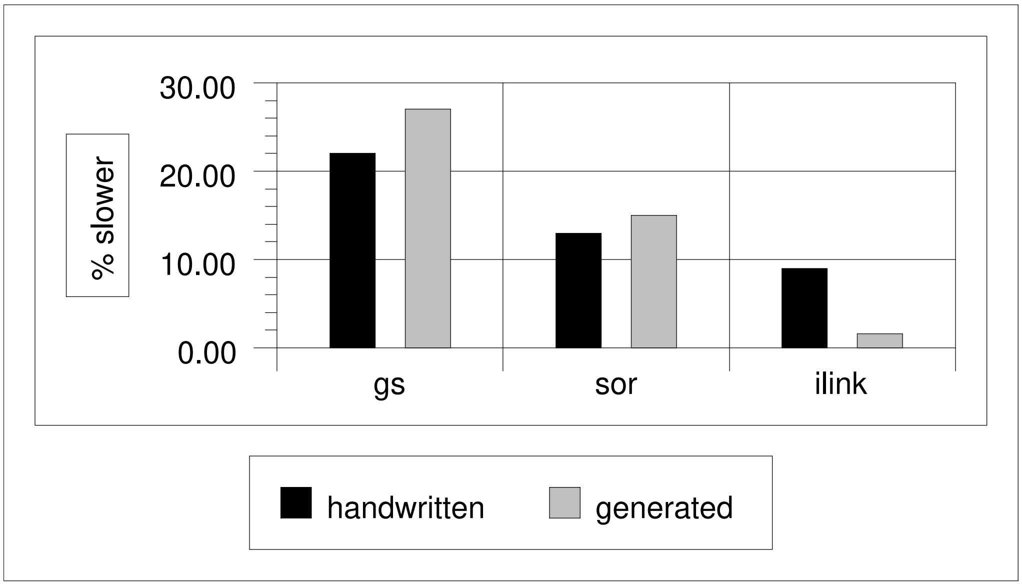

Figure 9. Thread Overhead 1x3

performance of the compiler generated code by using runtime debugging tools [18] to detect and get rid of extra synchronizations.

6 Related Work

Previous work has been done on using SMP nodes as building blocks for software shared virtual memory systems. Cox et al. [6] simulated a hybrid hardware and software shared memory system. They found that for applications with good locality and moderate synchronization rates, this hybrid approach results in performance comparable to that obtained using a pure hardware approach.

Karlsson et al. [13] provided a simulation study of the TreadMarks protocol running on an ATM cluster. They found that, given the parameters in their study, dedicating a processor in the SMP to the software protocol processing does not pay off since there is high likelihood of finding spare cycles on the compute processors on a node.

SoftFlash [10] uses a modified version of the Stanford FLASH protocol to support coherence in software at the granularity of 16KByte pages across a cluster of SGI Power Challenge machines connect by HIPPI links. The study shows that while clustering is effective in reducing inter-node communication, it is often accompanied by an increase in the latency of such communications due to TLB shoot-downs that are necessary for data coherence. They also use 5 dedicated processors for network communication and protocol handling for each node.

Daniel Scales et al. [8] studied the opportunities for improving performance of the Shasta system on a cluster of SMP nodes. The Shasta system supports fine-grain access to shared memory by inserting code before loads an stores in application code to check the state of the shared data being accessed. They found that clustering improves performance for Shasta system by exploiting the fast communication provided by SMP nodes. In their system, a private state table is maintained for each processor to keep track of whether a processor has accessed some shared data block. When sharing status of some data changes, downgrade messages are sent to processors that are accessing the same data based on the information stored in the private state tables.

Our system differs from above systems in several aspects. We used light-weight threads as local contexts to exploit the physical shared memory. We did not use dedicated processors for software protocol handling. Threads gave us good load balancing automatically. We also supported a unified programming model for the hierarchical parallel architecture.

SGI Power compilers [19] compile sequential Fortran or C programs with parallel directives to multi-threaded parallel programs targeted for SGI multiprocessor machines. The commercial APR Forge SPF compiler also compiles sequential programs into shared memory programs.

7 Conclusions

We built a clustered shared memory system for clusters of SMPs. We used TreadMarks to implement a cross node shared memory abstraction and used the physically shared memory available within each SMP node. Our system provides a unified fork-join programming model which allows users to express naturally the two level parallelism available in clustered shared memory systems. We also developed a front end compiler that generates TreadMarks parallel programs in our fork-join model from sequential programs with simple loop annotation directives. The performance of our compiler generated programs is comparable to that of handwritten TreadMarks programs. The analysis of the clustering effect on all applications show that clustering dramatically reduces cross node communications and thereby increases performance.

8 Acknowledgments

I would like to express my thanks to Dr. Zwaenepoel and Dr. Cox. Their knowledge and experience, together with their patience, has been a great help to me. My thanks to Cristiana Amza and Karthick Rajamani for developing the multithreaded TreadMarks library. Thanks to Johnny Chen, Honghui Lu, Ramakrishnan Rajamony, and Weimin Yu for the various contributions they made to this paper.

References

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

[2] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 7(4):273–298, November 1986.

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

[4] S.H. Blanton, J.R. Heckenlively, A.W. Cottingham, J. Friedman, L.A. Sadler, M. Wagner, L.H. Friedman, and S.P. Daiger. Linkage mapping of autosomal dominant retinities pigmentosa (RP1) to the pericentric region of human chromosome 8. *Genomics*, pages 857–869, 1991.

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

[6] A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, and W. Zwaenepoel. Software versus hardware shared-memory implementation: A case study. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[7] 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*, April 1997.

[8] K. Gharachorloo D. Scales and A. Aggarwal. Fine-grain software distributed shared memory on SMP clusters. Technical Report 97/3, Digital Western Research Laboratory, March 1997.

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

[10] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. Softflash: Anylyzing the performance of clustered distributed visrtual shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, October 1996.

[11] R. Gillett. Memory Channel: An optimized cluster interconnect. *IEEE Micro*, (16(2)), February 1996.

[12] S. Hiranandani, K. Kennedy, and C. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, August 1992.

[13] M. Karlsson and P. Stenstrom. Performance evaluation of a cluster-based multiprocessor built from atm switches and bus-based multiprocessor servers. In *Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pages 4–13, February 1996.

[14] P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency.* PhD thesis, Rice University, December 1994. Appeared as Rice Technical Report RICE COMP-TR-240 and available by anonymous ftp from cs.rice.edu under public/TreadMarks/papers.

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

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

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

[18] R. Rajamony and A. L. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 250–256, February 1996.

[19] Silicon Graphics, Inc. *SGI Power-C User's Guide.*