Contention Elimination by Replication of Sequential Sections in Distributed Shared Memory Programs

Honghui Lu, Alan L. Cox, and Willy Zwaenepoel
Department of Computer Science
Rice University
Houston, TX 77005
hhl,alc,willy@cs.rice.edu

ABSTRACT

In shared memory programs contention often occurs at the transition between a sequential and a parallel section of the code. As all threads start executing the parallel section, they often access data just modified by the thread that executed the sequential section, causing a flurry of data requests to converge on that processor.

We address this problem in a software distributed shared memory system by *replicating* the execution of the sequential sections on all processors. Communication during this replicated sequential execution is reduced by using multicast.

We have implemented replicated sequential execution with multicast support in OpenMP/NOW, a version of of OpenMP that runs on networks of workstations. We do not rely on compile-time data analysis, and therefore we can handle irregular and pointer-based applications. We show significant improvement for two pointer-based applications that suffer from severe contention without replicated sequential execution.

1. INTRODUCTION

Contention is a well-known problem for parallel programs. Contention occurs when many processors send requests to the same processor at approximately the same time. The latency for the responses to these requests increases, and as a result the progress of the requesting processors gets delayed. This problem can occur both in message passing and in shared memory systems, where the messages are implicitly generated by the consistency mechanism.

Sequential sections in shared memory parallel programs are a major source of contention [11]. Parallel programs usually have sequential sections for initializing data and for parts of the code that are either too complicated or too expensive to be run in parallel. In the shared memory paradigm, a sequential section is executed by a single thread, the *master* thread. At the beginning of the parallel section following a sequential section, many or all threads often access shared data modified by the master thread during the sequential section. Each thread may access a different part of the data, but they all direct their data requests to the master thread, which has the single up-to-date copy of the data.

We propose a method that eliminates contention caused by the sequential sections of (software) distributed shared memory parallel programs. Our solution replicates the sequential execution on all threads, allowing each thread to carry out the writes on its local copy of the shared data. This completely eliminates the communication to propagate changes made during the sequential execution. During the replicated sequential execution, we take advantage of the fact that every thread executes the same code: When several threads need the same data, only one request is sent, and the result is multicast to all threads participating in the parallel computation. Because a multicast message has in principle the same cost as a point-to-point message, the replicated sequential execution would ideally take the same amount of time as the non-replicated version. In practice, multicast requires more complicated flow control, and the attendant cost substantially increases the execution time of the replicated sequential execution. In the applications that we have looked at, however, the gain achieved by reducing the contention in the subsequent parallel section is larger than the slowdown as a result of using flow-controlled multicast in the sequential section. This is the case even with the highly conservative flow control measures currently in use in our prototype. We are exploring better flow control algorithms that allow more concurrency in message delivery and should produce better execution times as a result.

Our work is done in the context of the OpenMP/NOW system [19]. The OpenMP directives allows easy identification of sequential sections. Our work does not rely on compile-time data analysis. As a result, even irregular pointer-based applications can benefit from our optimizations. We do assume, however, that execution of the sequential sections is deterministic.

OpenMP/NOW is an extension of the TreadMarks software distributed shared memory (SDSM) system [6], and uses the SUIF compiler toolkit [5] to generate TreadMarks code from OpenMP programs [19]. We have measured the performance of the techniques discussed in this paper on a 32-node Athlon-based cluster connected by 100Mbps switched Ethernet. Barnes-Hut [25] and Ilink [9] are used as applications.

PPOPP'01, June 18-20, 2001, Snowbird, Utah, USA.

The rest of this paper is organized as follows. Section 2 provides some general background on OpenMP/NOW. Section 3 explains the cause of contention and its effect on SDSM systems. Section 4 discusses our design choices. Section 5 details the implementation of our system. Section 6 presents the resulting improvements. Section 7 discusses related work. We conclude in Section 8.

2. BACKGROUND

The OpenMP Application Program Interface (API) [20, 21] supports shared memory parallel programming in C/C++ and Fortran programs. Current OpenMP implementations are limited to shared memory architectures. We have implemented the first prototype of OpenMP on distributed memory machines, and in particular on a network of workstations (NOW) [19]. We use the TreadMarks SDSM system to implement a shared memory abstraction on a NOW, and we use the SUIF compiler tool kit [5] to translate OpenMP source code into TreadMarks programs. We only discuss those aspects of OpenMP and TreadMarks necessary for an understanding of the rest of the paper. For a complete discussion, we refer the reader to the standard references [6, 20, 21]

2.1 The OpenMP API

The OpenMP Application Program Interface (API) [20, 21] specifies a collection of compiler directives, library functions, and environment variables that can be used to specify shared memory parallelism in C/C++ and Fortran programs. OpenMP is based on a fork-join model of parallel execution. The sequential code sections are executed by a single thread, called the *master thread*. The parallel code sections are executed by all threads, including the master thread.

The fundamental construct for expressing parallelism is the `parallel` directive. It defines a *parallel region* of the program that is executed by multiple threads. *Work sharing* constructs divide the computation among the threads within a parallel region. For example, the `for` directive divides the iterations of the associated loop among the threads so that each iteration is performed by a single thread. In addition, *data environment* directives allow the programmer to specify whether variables are shared or private.

Our prototype supports a subset of the OpenMP API. The subset includes the parallel construct, the work sharing constructs, the combined parallel and work sharing constructs, various data environment directives, synchronization methods (locks and barriers) and the library functions. We support static block or cyclic partition of loops. We do not support nested parallel regions.

2.2 TreadMarks

TreadMarks [6] is a user-level SDSM system. It provides a global shared address space on a distributed memory machine.

2.2.1 Thread and Synchronization Model

In addition to conventional locks and barriers, TreadMarks includes the `Tmk_fork` and `Tmk_join` synchronization primitives, specifically tailored to the fork-join style of parallelism expected by OpenMP and most other shared memory compilers [5].

For performance reasons, all threads are created at the start of a program's execution. Initially, the master thread executes the program while the slave threads are blocked inside the runtime system waiting for the master to issue a `Tmk_fork`. When the master arrives at a `Tmk_fork`, it sends a fork message to all the waiting slave threads to wake them up. The fork message also contains information for the slaves to direct their execution during the parallel section, essentially a subroutine to be executed, its arguments, and some additional information. After the parallel execution, all threads call `Tmk_join`. A `Tmk_join` on a slave sends a join message to the master thread indicating completion of the parallel section. On the master, `Tmk_join` causes the master thread to wait for receipt of a join message from each of the slaves. Afterwards, the master continues with the program and the slave threads are blocked waiting for the next `Tmk_fork`.

2.2.2 Memory Model

TreadMarks relies on the operating system's virtual memory page protection mechanism to detect accesses to the shared pages, and the threads communicate via software messaging such as UDP on a local area network. Because sending messages over a network is very expensive, TreadMarks takes great effort to minimize messages.

TreadMarks features a *release consistent* (RC) [14] shared memory model. In the RC model, shared memory accesses are categorized either as *ordinary* or as *synchronization* accesses, with the latter category further 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 mutual synchronizations. In practice, this model allows a thread to buffer multiple writes to shared data in its private memory.

TreadMarks implements a *lazy invalidate* version of release consistency [15]. The propagation of modifications is postponed until the time of the *acquire*. Furthermore, an invalidate protocol is used. Instead of sending new data to the acquirer, the releaser notifies the acquirer of which pages have been modified, causing the acquirer to *invalidate* its local copies of these pages. A thread incurs a page fault on the first access to an invalidated page.

False sharing also causes frequent communication. False sharing occurs when two or more threads access different variables within the same page, with at least one of the accesses being a write. TreadMarks uses a *multiple-writer* protocol to address this problem. With the multiple-writer protocol, two or more threads can simultaneously modify their own copy of the 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. In order to distinguish changes made by different threads, at the time the thread sends out updates to the shared page, instead of sending the whole page, only the modified values are sent. Those modified values are called *diffs*.

2.3 The OpenMP to TreadMarks Translator

The OpenMP to TreadMarks translation is relatively simple, because TreadMarks already provides a shared memory API. A large part of the translator deals with transforming

the parallel regions to the fork-join format and implementing the data environment directives.

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the translator encapsulates each parallel region into a separate subroutine. This subroutine also includes code, generated by the compiler, that allows each thread to determine, based on its thread identifier, which portions of a parallel region it needs to execute. At the time of a fork, the master passes a pointer to this subroutine to the slave threads. To make the address of shared variables available to the slaves, the master passes their addresses to the slaves at the fork.

Variables defined outside the parallel region are redefined in the new subroutine. The private variables keep their original types. The types of shared variables are changed to the reference types derived from their original types.

In TreadMarks, shared variables must reside on the shared heap. The translator gathers all shared global variables in a structure, and allocates that structure on the shared heap. Private variables are implemented by redefining them in the parallel subroutine generated by the compiler, so that each thread accesses the private copy on its own stack.

3. CONTENTION

This section first illustrates the effect of contention on SDSM systems, and then explains why sequential sections are major sources of contention.

We define *contention* as the arrival of one or more diff requests on a node before the diff in response to a previous request has left the node. Contention can result from limitations in the node or in the network. In the former case, the time that the node requires to process a request is longer than it takes for the next request to arrive. In the latter case, the node fails to push out responses fast enough due to bandwidth limitations in the network link. Most systems, under this condition, wait for the interface to free an entry in its output queue.

Although contention happens at the node holding the updates, its effects are felt by the requester. As most SDSM systems handle requests in the same order in which they arrive, the service time for a request that arrives at a node with pending requests is increased by the time required to process all pending requests.

Sequential sections are a major cause of contention. While clever programming can reduce the amount of code in sequential sections, they are often hard to avoid completely. For instance, initialization and irregular load balancing codes often run sequentially. These sections usually take a very small portion of the execution time, and parallelizing them can be both complicated and unprofitable. A parallel region whose amount of work varies from iteration to iteration may also be executed sequentially when the amount of work associated with it is below a threshold.

In the shared memory paradigm, the sequential sections are executed by a single thread (the master thread in OpenMP). The result of the sequential computation automatically becomes visible to the other threads as a result of the fork synchronization between the sequential and the following parallel section. Data written during the sequential section is often immediately accessed by many threads in the parallel section that follows. Each thread may access a different part of the data, but they all send their requests to the last writer of the data, which is the master thread.

Consider for example, Barnes-Hut, a N-body simulation program from the SPLASH-2 benchmark suite [25]. A shared oct-tree is used in Barnes-Hut for load balancing purposes. The tree is rebuilt at the beginning of each iteration by reading all the particles updated in the previous iteration. Because the tree construction takes 0.3% of the complete program's sequential execution time and because it is rather complicated to parallelize, it is executed sequentially. At the beginning of the parallel force evaluation phase that follows tree building, each thread performs a top-down traversal of the tree to find its own particles. Contention occurs here because every thread requests parts of the newly computed tree from the master thread.

Contention may also occur between two consecutive parallel sections, but it is not inherent in this situation. Because parallel regions distribute the computation and data accesses among different threads, there need not be a single thread to which all requests for data are directed simultaneously. Careful programming practice (by staggering communications) can often avoid contention in this case.

4. DESIGN

4.1 Principle

Our solution *replicates* the execution of sequential sections on all threads, allowing each thread to carry out any writes during the sequential section on its local copy of the shared data. Quite obviously, writes performed during replicated sequential execution need not be propagated to other nodes, thus eliminating the contention after the sequential section.

Replicated sequential execution, by itself, would, however, introduce severe contention at a different place, namely during the execution of the sequential section itself. Indeed, while in normal sequential execution only the master thread accesses remote data, during replicated sequential execution *all* threads access remote data. Since they all execute the same code, remote accesses for the same data tend to occur at approximately the same time on all threads, thereby causing severe contention.

The fact that during replicated sequential execution every thread executes the same code and accesses the same data allows, however, the use of multicast to eliminate this newly introduced contention. When several threads need the same data, only one request for the data is sent, and the reply is multicast to all threads. Contention is reduced because there is only a single request for each data item, as during normal sequential execution.

Returning to the example of Barnes-Hut, each thread executes the sequential tree building. Because in doing so a thread reads all the particles, the particles are multicast during the replicated execution. As a result, by the end of the tree building phase, the tree is available locally on each node, and no contention occurs at the beginning of the parallel section.

4.2 Comparison to Alternatives

An obvious alternative solution is to try to predict which threads are going to access what data at the beginning of the parallel section, and unicast or multicast the pages to those threads accordingly. This prediction can be done either by the compiler or at runtime. Compile-time analysis can predict data access patterns [2, 4, 7, 18], but it is limited to regular access patterns, which do not include pointer-based

applications such as Barnes-Hut. In contrast, our method does not rely on compile-time data access analysis. Runtime speculation has also been proposed to predict accesses based on the history of previous accesses [23], but such an approach works only for applications that have repetitive access patterns from one iteration to another.

Another possible solution is to multicast all data modified during the sequential execution to all threads before parallel execution starts. This method is expensive if threads access only a small part of the modified data. For instance, in Barnes-Hut, except for the nodes near the root of the tree that are accessed by all threads, most of the tree is accessed by only a subset of the threads. With a larger problem size and more processing nodes, we expect most data to be accessed by an ever smaller number of threads.

5. IMPLEMENTATION

The TreadMarks runtime library is modified to support the replicated sequential execution and multicast as described above. We start with some necessary background on the implementation of the multiple-writer, lazy invalidate, release consistency protocols in TreadMarks.

5.1 TreadMarks Implementation Background

At a synchronization, the acquiring thread needs to be informed of the modifications to shared memory it needs to see according to the definition of release consistency (see Section 2.2). To do so, the execution of each thread is divided into *intervals*. A new interval begins every time a thread executes a release or an acquire. Each thread has an *interval index*, which is incremented every time a new interval starts on this thread. Intervals on different threads are partially ordered [1]: (i) intervals on a single thread are totally ordered by program order, (ii) an interval on thread p *precedes* an interval on thread q if the interval of q begins with the acquire corresponding to the release that concluded the interval of p, and (iii) an interval precedes another interval by transitive closure. With locks, the interval corresponding to the release of a lock directly precedes the interval beginning with a subsequent acquire to the same lock. With barriers, any interval corresponding to the barrier arrival precedes all intervals corresponding to the subsequent barrier departures. However, no ordering exist among the barrier arrivals, or among the barrier departures.

Each interval has a vector *timestamp* to record its knowledge of intervals in other threads that precede it. A timestamp contains an entry for each thread. In the timestamp of the ith interval of thread p, the entry for thread p is equal to i. The entry for a thread q other than p denotes the most recent interval of thread q that precedes interval i of thread p according to the partial order.

RC requires that before a thread p may continue past an acquire, the updates of all intervals preceding the current interval must be visible at p. Therefore, at an acquire, p sends its current interval timestamp to the previous releaser q. The releaser then compares the corresponding entries of both timestamps, and sends a message to p including *write notices* for all intervals named in q's current interval timestamp but not in the timestamp it received from p. A *write notice* is an indication that a page has been modified in a particular interval. Process p invalidates all pages for which it receives a write notice, and computes a new vector timestamp according to the pair-wise maximum of its previous timestamp and the releaser's timestamp.

TreadMarks uses *lazy diff creation*. A diff is not created at a synchronization, but *only* when it is requested. A diff is also created when a write notice from another thread invalidates the page. In this case, it is essential to make a diff in order to distinguish the modifications made by the different threads. Lazy diff creation results in a decrease in the number of diff creations and an attendant improvement in performance [16].

5.2 Replicated Sequential Sections

A join before a replicated sequential section behaves like a barrier. The master thread waits until all threads have arrived at the join, then issues a departure message so that all threads continue to the replicated sequential section.

At the fork at the end of a sequential section, threads wait until all other threads have finished the sequential execution before proceeding to the next parallel section. No memory coherence information is exchanged at the fork, because every thread executes exactly the same code during the replicated sequential execution.

Shared memory allocation, input and output instructions are not duplicated. We use the runtime variables provided by OpenMP to guard each memory allocation, input or output instruction so that it is executed only by the master thread during the sequential execution.

5.3 Integration with the TreadMarks Consistency Protocol

The synchronization before the sequential execution causes memory consistency information to be distributed to all threads according to the definition of release consistency. As a result, pages modified by another thread in the prior execution are invalid, but each thread can locate the diffs that it needs to bring its pages up-to-date by virtue of the write notices it received (see Section 5.1).

During replicated sequential execution shared memory modifications must not be propagated, because their propagation would quite obviously be redundant and may in fact lead to incorrect results.

In general, the TreadMarks consistency protocol achieves this result automatically, with one exception as explained below. Since there is no synchronization inside the sequential section, and since the TreadMarks lazy release consistency protocol propagates consistency information only at synchronization, updates made during the replicated sequential section by one thread, in general, do not become visible at other threads. However, lazy diff creation (see Section 5.1) may still cause some updates made during the sequential section to be propagated.

Lazy diff creation can cause an update made during replicated sequential execution to be propagated in the following scenario. Suppose thread p faults on a page modified by q before and during the replicated sequential execution, and suppose furthermore that the corresponding diff has not yet been requested by any other thread. In this case, the diff is created only at this point (when it is requested), and includes the modifications made during the replicated sequential execution.

To avoid this problem, all dirty pages are write-protected before entering the sequential section. During the sequential execution, the first write to the page causes a page fault,

after which a diff is created for the page. A twin is allocated after the diff creation and the page is made writable. A diff request during the replicated sequential section obtains this diff, which contains only the modifications made before the sequential section. At the end of the replicated execution, the remaining write-protected dirty pages are unprotected and returned to their normal state. An alternative solution to this problem would be to disable lazy diff creation, and create diffs for all modifications before entering the sequential section, but we have found this solution to be more expensive.

5.4 Multicast Implementation

Our implementation is based on IP multicast. There is a single multicast group which every thread joins at the beginning of the program. The main task of the multicast implementation is to add reliability to IP multicast. How to provide reliable and efficient multicast at either the transport layer or the application layer is an ongoing research topic. This paper does not focus on multicast protocols; we build a simple multicast mechanism tailored to our needs, that provides reliability and flow control. In the future, this multicast protocol could be improved, or suitable protocols developed elsewhere could be used here.

In the simplest possible form of multicast implementation, a node that faults could simply send out its requests for missing diffs, according to the write notices it has, and the appropriate threads would respond by multicasting their diffs. Upon receipt, each thread, including the faulting thread, would incorporate the appropriate diffs into its copy of the page. When all necessary diffs are received, the faulting thread would continue. On other threads, if all the diffs are received to make the page valid, the page protection would be changed to valid so that no further page faults are taken. If diffs are still missing, the page would remain invalid, causing a fault when accessed and causing the diffs that are missing to be fetched.

Unfortunately, this simple solution does not work well, because it requires a rather wholesale re-design of how TreadMarks works, and/or it may lead to overflow in the nodes' receive buffers. The TreadMarks code is written under the assumption that diffs arrive at a node only when a thread on that node has requested and is waiting for the diffs. Therefore, diff arrival does not cause an interrupt. During replicated sequential execution, the above simple strategy could quickly fill up the incoming network buffers if a number of diffs were received for which the thread was not waiting (because those pages were valid on that node or the execution had not yet proceeded to that point). The alternative of generating an interrupt on each diff arrival is also un-attractive, because it is inefficient and it requires a major re-design of the code. Even if we did generate an interrupt on each diff arrival, there may be so many incoming diffs that the buffers could still overflow. For all these reasons, we have chosen for an implementation where some amount of flow control is exerted on the multicast messages.

5.4.1 Determining the Requester

In the absence of lost messages, only one node sends out a diff request for a particular page. In order to choose the sender without additional communication, a thread must know whether a page is valid or not on other threads, as well as which diffs are missing on other threads. Since this information is not available in the original TreadMarks system, we augment each page with an array of *valid notices*, with one entry corresponding to each thread. Valid notices are exchanged only at the join before a sequential section. A valid notice records, for each page and for each thread, the timestamp of the latest interval during which the page is brought up-to-date by a thread. By comparing the write notices and the valid notices for a particular page, each thread can infer which threads fault on this page during replicated sequential execution, and which diffs are missing on all of those threads. The thread with the lowest thread identifier sends the diff request, and requests the diffs necessary to make the page consistent on all threads.

5.4.2 Reliability and Flow Control

Diff requests from different threads are serialized at the master thread. A request is first sent via a point-to-point message to the master, and the master thread multicasts the *request* to all threads. This request performs two tasks: (1) it request the diffs, and (2) it alerts the other threads that a set of multicast diff replies is about to arrive.

After receiving the multicast diff request, the threads that have the diffs multicast them in turn, one thread sending at a time, in order of increasing thread identifier. Flow control is achieved by requiring each thread to multicast an acknowledgment after receiving all the diffs sent by the preceding thread. When a thread has diffs to send, the diff reply message serves as an acknowledgment, otherwise a null acknowledgment message is sent.

With this flow control in place, lost messages are extremely rare. We therefore use a rather expensive mechanism for recovering from lost messages, since it is almost never invoked. When a thread times out on receive, it sends out a request asking for its missing diffs regardless of other threads that may also fault on the same page, and the replies are multicast to all threads.

5.4.3 Discussion

This multicast implementation has considerable overhead, including (1) the overhead for choosing the sender of the multicast request, (2) the overhead for exchanging the valid notices at the entry of sequential sections, (3) the forwarding of requests to the master thread, (4) the null acknowledgment messages, and (5) some loss of concurrency. With normal sequential execution, all missing diffs for a page are requested in parallel. A diff is sent immediately after computing it, and the replies are processed by the requester in first-come-first-served order. The slave threads thus compute diffs concurrently. Furthermore, the diff creation on the slave threads and the diff application on the master thread proceed at the same time. This concurrency is lost under our flow control scheme.

As will be seen in Section 6, on one hand, the multicast overhead is small enough that replicated sequential execution wins over normal sequential execution, but, on the other hand, it is large enough to noticeably affect our results. For this reason, we are actively working on a flow control mechanism with less overhead.

6. RESULTS

Our experiments are conducted on a network of 32 nodes running FreeBSD 4.1.1. Each node contains an 800MHz AMD Athlon processor and 256MB of memory. The nodes

are connected by a 100Mbps switched Ethernet and a 100Mbps hub. We recognize that faster networks are available that would alleviate congestion with the given processors. We foresee, however, that future faster processors and multiprocessor nodes will re-introduce congestion, even with faster networks. All unicast messages go through the switch, while all multicast messages go through the hub. The switch was found to be rather slow at delivering multicast messages, and therefore we chose a hub to carry the multicast messages. The addition of the hub does not appreciably affect the results for normal execution, since the additional bandwidth offered to unicast messages is limited (100Mbps on a total of 3.2Gbps).

Our system is based on TreadMarks 1.0.3. We compare the system with replicated sequential execution with the original OpenMP/TreadMarks system. We use two pointer-based applications, Barnes-Hut and Ilink, in the evaluation.

Tables 1 and 3 show the execution times and the speedups of each application. We measure the execution time of the sequential program on a single node without any parallel directives. In addition to the total execution time, we also measure the time each program spent in the sequential and the parallel sections of the code.

The amount of communication is shown in Tables 2 and 4. In addition to the total message and byte count, we measure the number of messages and the amount of data sent for diff requests, and distinguish between diff requests made during the sequential and the parallel sections. Each multicast message is counted as a single message.

We also measure the number of page faults and the average response time for serving a page fault. These numbers show the effect of contention reduction as well as the multicast overhead in the replicated sequential sections. For the sequential sections, we only count the number of page faults on one thread, the master thread in the original version, and the thread incurring the most page faults in the replicated version. For the parallel sections, we count the average number of page faults per thread.

6.1 Barnes-Hut

6.1.1 Application

Barnes-Hut [25] is an N-body simulation program using the hierarchical Barnes-Hut method. A tree structure is used to represent the recursively decomposed sub-domains (cells) of the three-dimensional physical domain containing all of the particles. The leaves of the tree correspond to the particles, and are contained in a separate array.

Each iteration is divided into two steps. In the first step, a single thread reads the particles and rebuilds the tree. The second step, in which all threads participate, performs force evaluation. First, the threads divide the particles by traversing the tree in the Morton ordering (a linear ordering of the points in higher dimensions) of the cells. Specifically, the ith thread locates the ith segment. The size of a segment is weighted according to the workload recorded from the previous iteration. Then, each of the threads performs the force evaluation for its particles, which involves a partial traversal of the tree. During the force evaluation, a thread only modifies its own particles.

During the tree building, the master thread reads all the particles updated in the previous iteration and rewrites the tree structure. In the force evaluation phase that follows, every thread first traverses the tree to find its own particles. Contention occurs here because every thread requests updates to the tree simultaneously. Each thread then proceeds to update its own particles. During this period, a thread reads the tree, as well as a large part of the particles updated by the other threads during the last iteration.

In the replicated execution, the sequential tree building is replicated on all threads. Because a thread reads all the particles to build the tree, the pages containing the particles are broadcast during the replicated execution. As a result, by the end of the tree building phase, all threads have a local copy of the complete new tree and the up-to-date values of the particles.

6.1.2 Results

We run Barnes-Hut with 131072 particles for 2 timesteps. The sequential program runs for 359.4 seconds, of which 1.4 seconds are spent in the sequential tree building phase.

The speedup achieved by the base OpenMP system is 6.7 on 32 nodes. The sequential tree building takes 3.2 seconds, longer than the sequential program, because the master spends 2.1 seconds bringing in the particles from the slave threads. The parallel part takes 50.4 seconds, amounting to a speedup of 7.1 for that part alone.

The optimized OpenMP program achieves a speedup of 10.1 on 32 nodes, which is a 51% improvement over the original version. Time spent in the parallel sections is reduced to 21.1 seconds, compared to 50.4 seconds in the base version. The number of diff messages in the parallel sections is reduced from 5,006,252 to 3,045,226, and the amount of diff data is reduced from 739,139 kilobytes to 221,292 kilobytes. Because the parallel sections are free of contention in the optimized version, the average response time for a diff request is reduced from 3.34 milliseconds in the original version to 0.98 milliseconds. Since parallel sections are separated from the sequential sections by barriers, the execution time of the parallel section is determined by the slowest thread. In the original version, the slowest thread spends 34.6 seconds in diff requests during the parallel sections. With contention eliminated, this time is reduced to 5 seconds in the optimized version.

The improvement is both due to the contention elimination for the tree and the broadcasting of the particles during the replicated computation. To isolate the effect of contention elimination, we hand insert broadcasting of the tree between the non-replicated tree building and the parallel force computation, and measure the force computation part. Without contention, time for the parallel force computation is 36.9 seconds, the number of diff messages is 4,892,246, and the amount of diff data is 538,832 kilobytes. Thus, about half of the improvement stems from contention elimination and the other half from broadcasting the particles.

The replicated sequential sections take 14.4 seconds, 11.2 seconds longer than the base version. The time spent choosing a sender of a multicast request is very small, about 13 microseconds. 0.2 seconds of the overhead stem from exchanging valid notices at the beginning of the replicated execution. By far the largest part of the overhead comes from the increased communication and the multicast overhead during the sequential sections. In the original version, the master thread has 3072 page faults, and the average response time is 0.67 milliseconds. In the replicated execution, because other threads fault on pages valid on the master

	Sequential	Original	Optimized
Total time (sec.)	359.4	53.6	35.5
Total Speedup	N/A	6.7	10.1
Sequential time (sec.)	1.4	3.2	14.4
Parallel time (sec.)	358.0	50.4	21.1
Parallel speedup	N/A	7.1	17.0

Table 1: Barnes-Hut execution times on 32 nodes.

		Original	Optimized
Total	messages	5,106,237	3,254,275
	data (KB)	795,165	275,351
Seq	diff messages	96,848	205,892
	diff data (KB)	10,446	22,443
	diff requests	3,072	6,146
	avg response time (ms)	0.67	2.12
Par	diff messages	5,006,252	3,045,226
	diff data (KB)	739,139	221,292
	avg diff requests	8,479	3,116
	avg response time (ms)	3.34	0.98

Table 2: Barnes-Hut execution statistics on 32 nodes.

thread, the number of diff requests increases to 6146, and the average response time is increased to 2.12 milliseconds. The significant increase in average response time is largely due to the additional messages in the multicast implementation. The base system sends 96,848 messages during the sequential section, while the replicated version sends 205,892 messages, including 3,074 forwarded requests and 143,738 null acknowledgment messages.

6.2 Ilink

6.2.1 Application

Ilink [9, 17] is a widely used genetic linkage analysis program that locates specific disease genes on chromosomes. The input to Ilink consists of several family trees. The program traverses the family trees and visits each nuclear family. The main data structure in Ilink is a pool of `genarrays`. A genarray contains the probability of each genotype for an individual. Since the genarray is sparse, an index array of pointers to non-zero values in the genarray is associated with each one of them. A bank of genarrays large enough to accommodate the biggest nuclear family is allocated at the beginning of execution, and the same bank is reused for each nuclear family. When the computation moves to a new nuclear family, the pool of genarrays is reinitialized for each person in the current family. The computation either updates a parent's genarray conditioned on the spouse and all children, or updates one child conditioned on both parents and all the other siblings.

We use the parallel algorithm described by Dwarkadas et al. [12]. Updates to each individual's genarray are parallelized with the `parallel` directive. The bank of genarrays is shared among the threads. The master thread first examines the amount of work involved in the update, and decides to perform the update in parallel only if the amount of work exceeds a threshold. An `if` clause is used to express the conditional parallelization. The master thread then assigns the non-zero elements in the parent's genarray to all threads in a cyclic fashion. After each thread has worked on its share of non-zero values and updated the genarray accordingly, the master thread sums up the contributions of each of the threads.

In the base OpenMP program, contention occurs because all threads come to the master for the newly initialized or updated genarrays. Contention is extremely severe when the computation moves to a new nuclear family, because the whole pool of genarrays are overwritten by the master thread, and each thread has to read the genarrays of all family members in order to update one family member. In the optimized OpenMP program, the communication to fan-out the genarrays from the master thread, as well as the accompanying contention are eliminated. During the replicated execution, the contributions made by each thread during the previous iteration are broadcast to all threads.

6.2.2 Results

We run Ilink with the CLP input set, which requires 180 iterations. The sequential code runs for 99.0 seconds, with 2.2 seconds spent in the sequential sections. The base OpenMP system achieves a speedup of 1.9 on 32 nodes. The sequential part takes 5.5 seconds, of which 2.7 seconds are spent to bring in updates to the genarray from the slave threads. Because of the high degree of contention, the parallel sections take 48.1 seconds, amounting to a speedup of 2.0 for the parallel part.

The optimized OpenMP program achieves a speedup of 5.5 on 32 nodes, an 189% improvement over the original version. The improvement comes from the significant reduction in the parallel section execution time, from 48.1 seconds in the base version to 8.8 seconds in the optimized version. The number of diff messages in the parallel sections is reduced from 873,052 to 111,600, and the amount of diff data is reduced from 518,266 kilobytes to 13,895 kilobytes, which amounts to a 87% reduction of diff requests, and a 97% reduction of diff data from the original version. Because the parallel sections are free of contention, the average response time is reduced from 3.01 milliseconds in the original version to 0.64 milliseconds. In the original version, the slowest

	Sequential	Original	Optimized
Total time (sec.)	99.0	53.6	18.0
Total Speedup	N/A	1.9	5.5
Sequential time (sec.)	2.2	5.5	9.2
Parallel time (sec.)	96.8	48.1	8.8
Parallel speedup	N/A	2.0	11.0

Table 3: Ilink execution time on 32 nodes.

		Original	Optimized
Total	messages	1,002,787	230,392
	data (KB)	565,711	49,535
Seq	diff messages	104,530	94,589
	diff data (KB)	2,803	2,885
	diff requests	2,836	2,837
	avg response time (ms)	0.94	1.71
Par	diff messages	873,052	111,600
	diff data (KB)	518,266	13,895
	avg diff requests	12,318	540
	avg response time (ms)	3.01	0.64

Table 4: Ilink execution statistics on 32 nodes.

thread spends 39.8 seconds in diff requests during the parallel sections, while this time is reduced to 0.4 seconds in the optimized version. In contrast to Barnes-Hut, all the improvement in Ilink comes from eliminating contention for the genarrays. Because the genarrays are completely overwritten during the sequential sections, no benefit is gained from broadcasting each thread's contribution to the genarray in the previous iteration.

The sequential sections take 9.2 seconds, which is 3.7 seconds longer than the base version. The program spends 1.5 seconds in exchanging valid notices. This number is larger than the corresponding 0.2 seconds spent in exchanging valid notices in Barnes-Hut, because Ilink runs for 180 iterations instead of two, as in Barnes-Hut. The replication incurs about the same number of page faults as the original sequential sections. However, the average response time is increased from 0.94 milliseconds in the original version to 1.71 milliseconds. The replicated execution sends 94,589 messages versus the 104,530 messages sent in the original version. The slightly lower number of messages is the result of combining diff requests to several threads into one multicast message. Among the messages, 60,572 are diff requests and replies, and 33,016 are null acknowledgment messages. Because the number of messages does not increase, the multicast overhead in this case results mostly from the loss of concurrency in diff creation and application.

7. RELATED WORK

We are not aware of any system that replicates computation in shared memory environments. Process-replication has been used in distributed systems for fault-tolerance purposes in systems such as CIRCUS [8] and Manetho [13]. Manetho [13] depends on the fact that replicated processes execute the same deterministic program to optimize the multicast protocol. Compared with process-replication, our method only replicates parts of the program, and the replication is done for performance gains instead of for fault-tolerance.

Previous parallel computing systems using group communication spend a lot of effort on making sure the data is sent only to the threads that access it. Brazos [23] is an SDSM system that exploits hardware multicast to improve performance. Although their results are satisfactory on a cluster of six dual-processor nodes, their method does not scale to a large number of nodes. To avoid interrupting threads with too many unneeded multicast messages, they create a different multicast group for each access pattern. Although this method works fine with the six nodes used in their experiments, it does not scale. With the exponential increase of the number of access patterns, this method quickly exceeds the number of multicast groups allowed by the OS (typically 20 in Unix). Finally, they predict access patterns according to history, and thus the benefit is limited to repetitive access patterns. We are not limited by the number of multicast groups or by the need for repetitive access patterns.

There have been several compiler techniques proposed to generate collective communications for distributed memory systems [2, 4, 7, 18]. They are, however, limited to regular array based programs. The inspector-executor method has been proposed as a way to efficiently execute irregular array based computations on distributed memory machines [22]. Group communication can be applied when exchanging the data at runtime. However, the compiler analysis involved can be quite complicated [3, 10, 24].

8. CONCLUSION

Our system improves the performance of OpenMP on network of workstations. We do so by eliminating the contention caused by sequential sections of the program. Our solution assumes that sequential computation is deterministic, and replicates the sequential sections on all nodes. During replicated sequential execution, each thread modifies its local copy of the shared data. Therefore, threads do not need to send requests to the master thread for updates after the sequential section. We take advantage of the fact that all threads execute the same code to make use of multicast.

Our methods distinguish themselves from other uses of multicast in shared memory systems in that they do not require that future accesses can be predicted, either by the compiler or by the runtime. As a result, we are not limited to programs with regular array accesses or strictly repeating access patterns. We chose two pointer-based applications without strictly repeating access patterns that suffer from severe contention after sequential sections in the code, Barnes-Hut and Ilink. We demonstrated significant benefits for these applications. The main limitation of our implementation is the restrictive multicast flow control. We are exploring alternative flow control strategies that allow more concurrency in message delivery. We believe that such strategies are feasible and will substantially improve our results.

9. REFERENCES

[1] S.V. Adve and M.D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[2] A. Agarwal, D. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared memory multiprocessors. In *IEEE Transactions on Parallel and Distributed Systems*, volume 6, pages 943–962, September 1995.

[3] G. Agrawal and J. Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Proceedings of Supercomputing '95*, December 1995.

[4] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, June 1993.

[5] 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.

[6] 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.

[7] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN 93 Conference on Programming Language Design and Implementation*, June 1993.

[8] E.C. Cooper. Replicated distributed programs. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pages 63–78, December 1985.

[9] 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.

[10] R. Das, P. Havlak, J. Saltz, and K. Kennedy. Index array flattening through program transformation. In *Proceedings of Supercomputing '95*, December 1995.

[11] E. de Lara, Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. The effect of contention on the scalability of page–based software shared memory systems. In *Languages, Compilers, and Run-Time Systems for Scalable Computers(Proc. 5th Intl. Workshop LCR2000)*, Rochester, NY, May 2000. Springer-Verlag.

[12] 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.

[13] E.N. Elnozahy and W. Zwaenepoel. Replicated distributed process in Manetho. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 18–27, July 1992.

[14] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

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

[18] J. Li and M. Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[19] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In *Proceedings of Supercomputing '98*, November 1998.

[20] OpenMP Architecture Review Board. OpenMP Fortran Application Program Interface, Version 1.0. http://www.openmp.org, October 1997.

[21] OpenMP Architecture Review Board. OpenMP C and C++ Application Program Interface, Version 1.0. http://www.openmp.org, October 1998.

[22] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and run-time compilation. *Concurrency:Practice and Experience*, 3(6):573–592, December 1991.

[23] W.E. Speight and J.K. Bennett. Using multicast and multithreading to reduce communication in software DSM systems. In *Proceedings of the Fourth International Symposium on High-Performance Computer Architecture*, pages 312–323, February 1998.

[24] R. von Hanxleden and K. Kennedy. Give-N-Take – a balanced code placement framework. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, June 1994.

[25] 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.