Improving the Performance DSM Systems via Compiler Involvement

Ravi Mirchandaney
Seema Hiranandani
Ajay Sethi

CRPC-TR94369
January, 1994

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

Updated June, 1994.

Improving the Performance of DSM Systems via Compiler Involvement

Ravi Mirchandaney* Seema Hiranandani† Ajay Sethi†
mirchand@cs.rice.edu *seema@cs.rice.edu* *sethi@cs.rice.edu*

**Shell Oil Company, Bellaire Research Center, Houston, TX 77025*
†*Center for Research on Parallel Computation, Rice University, Houston, TX 77251-1892*

Abstract

Distributed shared memory (DSM) systems provide an illusion of shared memory on distributed memory systems such as workstation networks and some parallel computers such as the Cray T3D and Convex SPP-1. This illusion is provided either by enhancements to hardware, software, or a combination thereof. On these systems, users can write programs using a shared memory style of programming instead of message passing which is tedious and error prone. Our experience with one such system, TreadMarks, has shown that a large class of applications do not perform well on these systems. TreadMarks is a software distributed shared memory system designed by Rice University researchers to run on networks of workstations and massively parallel computers. Due to the distributed nature of the memory system, shared memory synchronization primitives such as locks and barriers often cause significant amounts of communication.

We have provided a set of powerful primitives that will alleviate the problems with locks and barriers on such systems. We have designed two sets of primitives, the first set maintains coherence and is easy to use by a programmer, the second set does not maintain coherence and is best used by a compiler. These primitives require that the underlying DSM be enhanced. We have implemented some of our primitives on the TreadMarks DSM system and obtained reasonable performance improvements on application kernels from molecular dynamics and numerical analysis. Furthermore, we have identified key compiler optimizations that use the non-coherent primitives to reduce synchronization overhead and improve the performance of DSM systems.

1 Introduction

It is generally accepted that programming distributed systems is complicated and time consuming for programmers. Most of the complexities are associated with problem partitioning, global to local address resolutions, and the incorporation of explicit message passing. For our discussions, distributed systems encompass workstation networks as well as massively parallel processors (MPPs) that do not provide a shared address space across the nodes of the machine (e.g., Intel Paragon, TMC CM-5, nCUBE/2).

Most researchers agree that a programming model that hides the low level details of programming distributed systems is desirable. Consequently, we have seen an increased interest in the idea of using DSMs to program distributed memory systems. DSM systems allow users to write programs using a shared-memory style of programming. A DSM may be implemented in software such Midway [8] [11], Mirage [18], Munin [14] and TreadMarks [24] or in hardware such as the Cray T3D or a combination of hardware and software as in Stanford's Flash Multiprocessor.

The programming model provided by DSM systems is desirable. However, due to the distributed nature of the memory system, shared memory synchronization primitives are expensive. Furthermore, optimizations performed by existing shared memory compilers do not necessarily work well for DSM systems. We have provided primitives that will reduce synchronization overhead by eagerly prefetching and updating data. We have designed two classes of primitives, the first we denote as *coherent* since they maintain the coherency associated with the synchronization operation, and the second we denote as *non-coherent*, prefetch data without maintaining coherence and hence are best used by a compiler. We have identified key compiler optimizations that use the primitives to improve the performance of programs designed to run on DSM systems. Furthermore, these primitives require that the underlying DSM system be modified. We applied our extensions to TreadMarks and obtained reasonable reduction in communication overhead.

TreadMarks is a software DSM system designed by Rice University researchers to run on networks of workstations and MPPs. The evaluation of TreadMarks on workstations connected by ethernet and ATM networks is provided Dwarkadas et.al [17] and Keleher et.al [24]. The results of various applications and kernels running on TreadMarks can be summarized by saying that moderate to coarse grained computations such as Jacobi iteration and Traveling Salesman Problem (TSP), show fairly good speedups. Programs such as Water and Cholesky from the Splash benchmark suite typically do not perform well. These programs involve significant synchronization overhead that can be reduced by compiler optimizations. These characteristics are not unique in any way to the TreadMarks design and implementation.

1.1 The Programming Model

In this paper, we address issues related to compiler optimizations for DSM programs. From the results of application kernels from molecular dynamics (MD) and numerical analysis (NA), we have determined that a variety of compiler optimizations can be performed on programs in order to improve their performance when running on DSMs. A pictorial representation of the programming model is depicted in Figure 1.

Our compilation system takes as input a sequential program and produces an output program that runs on an enhanced DSM system. The input program may be a shared memory program, however in that case, the user may choose to manually perform the relevant optimizations. It should be noted that we do not propose to reinvent all the shared memory compiler optimizations, some of which are described in [27] [2] [3] [4] [5] [30] and possibly performed by a compiler such as KAP [26]. Rather, a shared memory compiler or an interactive tool will form the basis on which our enhancements are built.

The output shared memory program produced by the compiler has the following characteristics:

- The iteration space is tiled across the processors and appropriate synchronizations are inserted.
- The data layout of the shared arrays are modified, based upon the perceived access patterns, in order to increase locality of accesses. The layout algorithm explicitly takes into account pages for allocating data.
- Optimizations are performed to reduce the number of global synchronizations and/or the overhead associated with the synchronizations.

The outline of the rest of the paper is as follows: Section 2 briefly reviews the TreadMarks DSM system and the basics of the consistency model that TreadMarks in based upon. Section 3 describes compiler optimizations and extensions to DSMs in order to support the optimizations. Section 4 illustrates the use of compiler optimizations on kernels from NA and MD. Section 5 provides performance results for DSM programs that incorporate these optimizations. We briefly survey related work in Section 6 and conclude in Section 7.

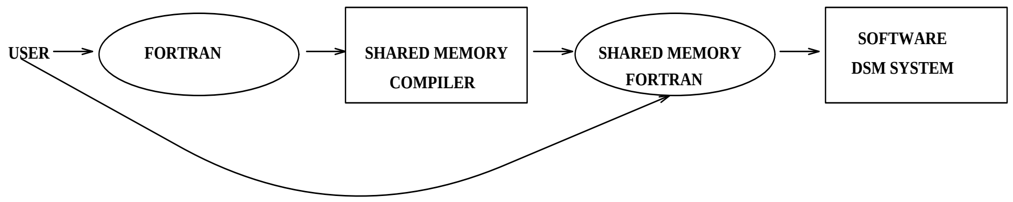

Figure 1 Programming Model

2 Description of TreadMarks

Release consistency (RC) [29] is a relaxed model of memory consistency that permits changes to shared memory to be delayed until certain synchronization actions are performed. Memory accesses can be described as being *ordinary* or *synchronization*, with the latter category further divided into *acquire* and *release* accesses. RC requires that ordinary accesses to shared memory be performed only when a subsequent release by the same processor is performed [24]. Acquires and releases may be thought of as being similar to synchronization operations performed on locks.

TreadMarks implements RC as well as mechanisms to minimize the number of messages that are sent. In that sense, it implements a *lazy* RC because changes to shared data are not propagated after a release. Modifications are only made visible to a processor after it performs an acquire. By comparison, Munin [7] implements an *eager* RC algorithm because modifications are made visible after a release is done by a processor. Both Munin and TreadMarks are able to delay modifications to pages by allowing multiple writers to modify disjoint portions of a shared page.

In TreadMarks, when a releaser grants a lock to a new acquirer, the releaser piggybacks on the lock grant message, a set of *write notices* that enumerate changed pages in the interval when the releaser owned the lock. Note that the actual pages are not sent at this time, only notices that specific pages have been modified. The *happened-before-1* partial order [1] is used to compute which changes need to be propagated to the acquirer. The *happened-before-1* partial order can be represented by means of vector timestamps on write notices. When an acquire is performed, the acquiring processor checks to see which pages in the set of write notices have vector timestamps greater than their corresponding vector timestamps on the local processor. Those pages are invalidated on the acquiring processor.

When a processor tries to access an invalid page, an access miss occurs. The processor contacts those processors that had sent write notices for this page. Updates to shared pages are sent by those processors in the form of *diffs*. Diffs are runlength encodings of changes to a page. In general, diffs are likely to be smaller in size than a full page, resulting in reduced network traffic. It is relatively straightforward to receive diffs of a given page from several processors and merge them into one coherent version of the page.

Several lazy RC algorithms can be implemented, depending upon whether the modified pages are invalidated or updated at the time of an acquire. In lazy invalidate (LI), the acquirer invalidates all pages for which it sees write notices in the synchronization message. Actual changes are transferred only when the processor tries to use a particular page that has been invalidated. In the case of lazy update (LU), the last releaser is notified by the acquirer. This processor (the last releaser) sends all changes to pages from its previous interval to the acquirer. No pages are invalidated at the acquirer. A variation which combines the advantages of LI and LU is called lazy hybrid (LH). In LH, a releaser sends all changed pages it believes the acquirer has cached at the time of an acquire. This may or may not include all pages that the acquirer actually needs. Those pages for which the processor has seen write notices but no diffs, are invalidated. The diffs are requested at the time the pages are accessed. The current version of TreadMarks supports LI. Hence, all our discussions in subsequent sections of this paper will assume an LI algorithm, unless we specifically say otherwise.

3 Compiler and DSM Optimizations

It is often possible for a user or a compiler to transform a program in such a way that its performance is improved when running on DSM systems. In this section we describe compiler optimizations, and the extensions required to DSM systems to take advantage of these optimizations. Our optimizations are targeted towards scientific array based applications.

In many problems, it is easy for a compiler or a user to associate a particular data item (e.g., an array section) with a lock or a barrier. Using this information, the latency associated with transferring the data between the last releaser and the subsequent acquirer is minimized. The idea behind this is to inform the DSM system that this particular memory section will be referenced and thus to update the section as soon as possible. In the case of a page based DSM system, the pages containing the memory section must be updated whereas in the case of cache based DSM systems, the cache line should be brought into memory using prefetch instructions if they exist. Those memory sections that have been modified but have not been associated with a lock or barrier operation, may simply be invalidated by the DSM. The following sections describe the primitives that enable these optimizations and provide algorithms for implementing these primitives in TreadMarks

3.1 Optimized Section Transfers

This set of primitives is designed to transfer data in an efficient manner between the releaser of a lock and the subsequent acquirer.

3.1.1 Section Locks

In an eager system such as Munin, when a lock is released by a processor p, all modifications to shared pages made by p are made visible to processors in the system. This is expensive, since every processor may not need to see all the changes. In a lazy system such as TreadMarks, modifications are made visible at other processors only when they access modified pages. This results in increased latencies for updating pages. We propose the notion of *section locks* (SL) which will alleviate these problems. Section locks provide information to the underlying DSM about the memory sections that should be sent across as early as possible. Appropriate usage of section locks can significantly reduce communication latency by sending the section specified in the lock operation early so that the updated section is in the processor's cache (as in hardware based DSMs) or local memory (as in software based DSMs) by the time it accesses the section. SL also provides a strategy for incorporating an update/invalidate protocol. An update protocol has the disadvantage of potentially over communicating whereas an invalidate protocol has the disadvantage of higher communication latencies. SL provides an efficient mix of both protocols by performing an update on the memory section specified in SL and performing an invalidate on other memory sections that have been modified. If correctly inserted by either a programmer or a compiler, SL provides the best alternative between an update and an invalidate protocol.

DSM Support

Figure 2 provides an algorithm to implement SL in TreadMarks. In order to maintain coherence the releaser sends diffs for pages encompassing the section and generates write notices for pages not included in the section. This information is sent in a single message. The releaser also increments its interval to reflect the release operation (roughly speaking, an interval is the period since the last synchronization point).

3.1.2 DSM_Send and DSM_Recv

Scientific computations that exhibit regular data access patterns have been successfully compiled for distributed memory machines. Such compilers are often able to analyze the exact array sections referenced in a loop nest. DSM_send and DSM_recv (DSR) may be inserted by a compiler in such situations. The idea here is similar to SL; however, this primitive allows updated data to be prefetched selectively without incurring the overhead of a lock operation. The cost associated with this primitive is lower than SL because the compiler can dispense with unnecessary consistency related overhead normally incurred by the DSM system.

Compiler Support

The Fortran D compiler uses Regular Section Descriptors (RSDs) [12] [6] to summarize the data region accessed in a loop nest. The RSDs are used to generate appropriate *send* and *receive* communication calls. For more details refer to Hiranandani et.al [22] [21]. We propose to use similar information to determine the data that must be communicated during a DSR operation. By communicating only the data actually used by the program, the DSR operation reduces the amount of communication incurred as well as hides

```
Given:  lock_acquire(id, mem(x1:x2)) operation by $receiver_p$
        lock_release(id, mem(x1:x2), $receiver_p$) operation by $sender_p$
Perform:  Section Lock operation
Algorithm:
  Construct set S from mem(x1:x2), s.t.  each $e \in S$
  is a page number and memory_sect(page(e)) $\subseteq$ mem(x1:x2)

  • lock_acquire(id, mem(x1:x2))
      receive diffs for pages $\in$ S and write notices
      for remaining modified pages from $sender_p$
      validate and update pages $\in$ S
  • lock_release(id, mem(x1:x2), $receiver_p$)
      for each $e \in S$ do
        if ( page(e) == invalid)
          receive diffs from processors who sent write notices
      endfor
      create new interval
      for each $e \in S$ construct diffs endfor
        send diffs to $receiver_p$
        piggyback write notices for pages modified in last interval
      endfor
```

Figure 2 Supporting Section Locks

message latency by performing the DSM_send as soon as the data is computed by the sender and performing the DSM_recv as late as possible. DSR takes as arguments, a memory section and a processor id p, and guarantees that the processor performing the DSM_recv will see the values of the memory section on processor p. DSR may be used to limit the amount of communication. A compiler can insert a DSR instead of either a barrier or a lock whenever it is able to analyze, at compile time, the sections referenced by the program. The potential improvements of DSR are described in the context of ADI (alternate direction integration) in Section 4.1. Below we show that for software DSM systems, the DSR primitive is more efficient than using the generic send/recv operations provided by the underlying hardware since DSR optimizes the operations performed by the DSM.

DSM Support

Figure 3 depicts the algorithm that we will use to implement DSR in TreadMarks. In a lazy DSM system such as TreadMarks, all diffs associated with the invalid page(s) that span the required data have to be obtained before the processor is allowed to continue. If the page(s) encompassing the memory section specified in DSM_recv are invalid, the receiver requests diffs from the sender instead of obtaining them from the processors that sent it write notices. This minimizes the number of interrupts because the receiver needs to interrupt only the sender. Furthermore, by combining the diff request for all invalid pages that span the memory section contained in DSR, fewer interrupts are incurred. If the generic send/recv provided by the hardware is used, then the optimizations described above can not be achieved. By incorporating a send/receive operation inside the DSM system, we are able to minimize the number of interrupts. After updating the pages, $receiver_p$ waits on a receive and proceeds when $sender_p$ sends the required data. It should be noted that a compiler may invoke SL on occasions where it is able to distinguish certain memory sections but not others.

3.2 Broadcasts

It is often the case that all processors reference the same memory location(s) that has been written to by a single processor. Examples of this type of memory reference pattern are seen in parallelized versions of

```
Given:  DSM_recv(mem(x1:x2), $sender_p$) operation by $receiver_p$
         DSM_send(mem(x1:x2), $receiver_p$) operation by $sender_p$
Perform:  DSM_Send_Recv operation
Algorithm:
  Construct set S from mem(x1:x2), s.t.  each $e \in S$
  is a page number and memory_sect(page(e)) $\subseteq$ mem(x1:x2)

   • DSM_recv(mem(x1:x2), $sender_p$)
       Construct I $\subset$ S s.t.  I is the set of invalid pages
       request diffs for pages $\in$ I from $sender_p$ and
         receive diffs, apply diffs and validate pages
       receive section from $sender_p$
       endfor
   • DSM_send(mem(x1:x2), $receiver_p$)
       send section to $receiver_p$
```

Figure 3 Supporting a DSM_Send_Recv

Gaussian Elimination and Cholesky Factorization where every processor reads the pivot element that has been written by one of the processors. In order to prevent a race condition from occurring, shared memory compilers are forced to insert a barrier after the write to the pivot element. A barrier ensures that all processors will see the changes that occurred prior to it, even though there may be only a single variable that is of interest to all processors.

We present two efficient broadcast primitives, one that can be inserted by a user and is coherent, and another which is non-coherent. Similar to DSR primitives, the non-coherent version can be directly invoked by the user, however the underlying DSM system provides no guarantees regarding the coherence of shared data and hence, we feel that such a primitive should be used only by a compiler.

3.2.1 Broadcast Barriers

Instead of inserting a normal barrier operation in parallelized versions of programs such as Gaussian Elimination and Cholesky Factorization, the user can insert a broadcast barrier (BB). The broadcast barrier provides information to the underlying DSM system that certain sections of memory should be updated as soon as possible. In a lazy system such as TreadMarks, the pages that contain the memory section are updated immediately and the rest of the pages that have been modified are set to invalid. BB, if used appropriately, has the advantage of combining the update/invalid protocol. Communication latencies are minimized by updating pages encompassing the section eagerly. Furthermore, unnecessary data is not sent; instead, modified pages that do not contain the memory section specified in the BB are simply invalidated. BB provides an excellent balance between an eager and lazy DSM system. With help from a user or a compiler, the DSM system is informed about the memory sections that will be referenced by processors soon after a barrier has occurred. Clearly, this information is very valuable to a DSM system which should use it to optimize data transfers.

Figure 4 describes the algorithm used to implement a BB in TreadMarks. For a BB, all pages involved in the memory section specified in the call will be updated at all processors. In general, a BB guarantees that the last write to the memory location(s) specified in the broadcast will be made visible to all the processors *immediately* after the BB has been executed. For modified pages not contained in the memory section specified in BB, write notices are sent and the pages are invalidated. For pages containing the specified memory section, the broadcasting processor receives diffs for the relevant pages from all processors. It then applies these diffs to its copy of the page and then broadcasts the updated page. Upon receipt, each processor sets the page to valid and copies the page over its older version.

By inserting a BB we expect to achieve significant improvements over a barrier in both an eager and lazy system. Since the underlying DSM system will communicate diffs for only those memory locations specified in the broadcast rather than the diffs for all pages that have been modified, we expect to do better than an eager system. For those pages updated in a BB, we will not incur the overhead of interrupts in a lazy system. More importantly, the latency of data transfer will be reduced as a consequence of incorporating the data transfer inside the barrier arrival routine.

3.2.2 Non-coherent Broadcasts

Compiler Support

A barrier on a DSM system is an expensive operation, particularly so on a software based DSM system. However, compilers for distributed-memory machines, such as the Fortran D compiler, are able to recognize

```
Given:  $barrier(id, mem(x1{:}x2))$, memory section to be broadcast
$t_p$ = my processor number
$bcast_p$ = broadcaster based on id
P = number of processors
Compute:  Broadcast barrier operation
Algorithm:
Construct set S from mem(x1:x2), s.t.  each $e \in S$
is a page number and memory_sect(page(e)) $\subseteq$ mem(x1:x2)
  if ($t_p \neq bcast_p$)
    $\forall e \in S$ compute diffs
    piggyback diffs with write notices on barrier arrival message
    receive updated pages from $bcast_p$
  endif
  if ($t_p == bcast_p$)
    receive diffs & write notices for new interval on barrier
    arrival message
  for each $e \in S$ do
    if ( $page(e)$ == invalid)
      acquire diffs for pending write notices
    endif
    apply all diffs to pages $\in S$
    send updated pages to all processors
  endfor
  endif
```

Figure 4 Supporting a Broadcast Barrier

that only a broadcast of the pivot element is required. We propose to use the same analysis performed by such compilers to insert a broadcast operation rather than a barrier. Below we describe the extensions required to a software DSM system to efficiently implement a broadcast operation. Section 4.2 indicates the possibilities of the improvements of a broadcast over a barrier on a representative linear algebra kernel.

DSM Support

We propose to implement a broadcast in TreadMarks using the non-coherent broadcast (NB) algorithm depicted in Figure 5. In a broadcast, one processor sends a section of its memory to all other processors. The implementation of broadcast will be different depending upon the underlying DSM system. The algorithm depicted in Figure 5 describes a broadcast operation on a lazy DSM system such as TreadMarks. If a page corresponding to the broadcast section is invalid at the receiver, it requests diffs for that page from the broadcaster. Else, it applies the section of data transferred by the broadcaster. The NB algorithm is more efficient than the BB since only a section of the memory is made consistent. However, we do not recommend that NB be used directly by a user. On the other hand, a compiler may invoke the BB primitive on certain occasions when it is unable to analyze all data accesses within a loop which involve a broadcast.

```
Given:  DSM_bcast(mem(x1:x2), $sender_p$), memory section to be broadcast
$t_p$ = my processor number
P = number of processors
Compute:  Non-coherent broadcast operation
Algorithm:
Construct set S from mem(x1:x2), s.t.  each $e \in S$
is a page number and memory_sect(page(e)) $\subseteq$ mem(x1:x2)
 if ($t_p$  ==  $sender_p$)
   send broadcast section to all processors
 else
   for each $e$
     if ( page(e) == invalid)
       request diffs from $sender_p$
       validate(page(e))
       apply diffs to page(e)
     endif
   receive broadcast section from $sender_p$
 endfor
```

Figure 5 Supporting a Non-Coherent Broadcast

3.3 Reductions

On DSM systems it is very difficult to extract any parallelism from reductions. Typically, the reduction variable is defined as a global, shared variable and, therefore, locks needs to be inserted to provide mutually exclusive access to the reduction variable. Inserting locks often sequentialize the whole computation. We propose compiler support and DSM enhancements to efficiently implement reductions consisting of global accumulation operations that are associative and commutative such as sum, product, min and max.

Compiler Support
Compilation techniques for distributed memory machines have been successful in recognizing and parallelizing reductions by allowing processors to compute partial values locally. At the end of the loop, all processors perform a global accumulation operation that updates their copy of the variable [22]. We will use the same compilation technology for parallelizing reductions. The compiler will insert a global accumulation call at the end of the reduction in order to collect the partial values computed on each processor. The underlying DSM must be extended to handle such global accumulations. Below, we describe the extensions to the underlying DSM.

DSM Support
We propose to implement accumulation operations in TreadMarks using the algorithm depicted in Figure 6. Since TreadMarks allows multiple writers, we allow each processor to write to its local copy of the page. At the end of the reduction computation, each processor will have written to the same location, but into a local copy of the page. In order to guarantee that all processors have a consistent version of the page after the reduction has occurred, we perform a global accumulation to collect the partial results. Notice that we "assign" pages to processors in a cyclic manner. Hence, a particular reduction may be computed concurrently by several processors. In the case of reductions, we expect the implementation to be identical

for lazy and eager DSM systems.

If the DSM system does not allow multiple writers such as the Cray T3D, the compiler will create a corresponding local variable. Each processor writes to the local variable and at the end of reduction, a global accumulation is performed to collect the partial results.

3.4 Sparse Matrix Computations

Loop nests which possess dependencies that cannot be characterized during compilation can be handled using the idea of *inspector-executor* [32] [20] [33] developed in the PARTI system. At compile time, the loop nest is transformed into two loops: an inspector and an executor. At run time, the inspector code is executed once to set up the communications on the processors such that each processor receives a message from those processors that have valid data to send to it. Appropriate code is generated for the executor to manipulate the remote data received from other processors. We will incorporate a framework similar to the inspector-executor in our compiler.

In certain computations an incomplete reduction, where not all processors concurrently update a variable, is performed. Such reductions can be parallelized by using scatter_ϕ routine provided by the PARTI framework [15]. The scatter_ϕ routine implemented for the DSM systems allows the compiler to perform a restricted reduction on array elements as well as to eliminate certain synchronizations. We now describe the compiler and DSM enhancements needed to permit concurrent updates to shared variables using the scatter_ϕ optimization.

Compiler Support
On DSM systems, like TreadMarks, that allow multiple writers to exist for each page, the scatter_ϕ routine

```
Given:  Global_Accumulate_φ(mem(x1:x2))
$\phi$ = Plus | Times | Min | Max | MinLoc | MaxLoc
$t_p$ = my processor number
P = number of processors
Compute:  Reduction operation in TreadMarks
Algorithm:
Construct set S, s.t.  each $e \in S$ is a 2-tuple <l:u>.
<l:u> is a memory section that resides on a single page
 for each $e \in S$ do
   $receiver_p = page\_num(e)$ mod $P$
   if ($t_p == receiver_p$)
     receive diffs from all processors
     combine partial values for <l:u> ∈ e using operator $\phi$
     send the updated page to all processors
   else
     send diffs to $receiver_p$
     receive updated page from $receiver_p$
 endfor
```

Figure 6 Supporting a Reduction in TreadMarks

can be implemented simply by letting each processor write to its local copy of the array during the loop and merging the relevant page locations using the operator ϕ after the loop. Scatter_ϕ routines needs an inspector to determine the correspondence between the processor numbers and the array elements modified by it. Since we are targeting a DSM system, we need to map elements onto pages, whereas the PARTI routines map elements to a processor and offset pair. We will elaborate on the use of this technique in the context of the MD kernel and TreadMarks in Section 4.4.2.

On the other hand, for the DSM systems that do not allow multiple writers, the compiler needs to create a local copy for each variable which is the target of a reduction operation. The inspector is used to determine the non-local references and, using this information, the compiler dynamically allocates local buffers only for the non-local references. Since DSM systems provide a fast mechanism for detecting non-local accesses, the compiler may eliminate the overhead of executing a separate inspector loop by using the first execution of the loop as the "inspector". Using this approach, the "inspector" iterations of the loop will record the non-local accesses (and other relevant information like the owner of the data, etc.) while the other iterations use this information. The Tempest/Typhoon system being developed by Reinhardt et.al [31] will allow certain iterations to be "inspector" iterations. Using the information collected by the inspector, the routines like scatter_ϕ are implemented as before.

DSM Support
Figure 7 depicts the modifications required to implement a Scatter_ϕ in a DSM system such as TreadMarks. Each processor uses the lists of elements to be received from remote processors to post receives. The sending processors send the elements, which are used at the receiving processors to perform incremental accumulations of portions of the array. This approach is similar to the approach taken by PARTI. We have implemented scatter_ϕ using a different approach described below.

The scatter_ϕ routine can be implemented using another approach which relies on the DSM nature of the system. During the execution of the executor, processors compute the partial sums for the array elements which are "owned" by other processors. Instead of sending these partial sums to the "owner" processor, each processor exclusively updates the global variable by adding the corresponding partial sums computed on the processor. Locality can be exploited by updating all the global variables owned by a processor consecutively. In other words, in this approach, instead of sending the partial sums to the owners, each processor acquires the appropriate array elements and adds the corresponding partial sums to them. Note that the processor updating the array elements need not know which processor last updated the array elements because the DSM system keeps track of this information.

The implementation of this primitive in either lazy or eager system will be identical. Pages that contain elements "owned" by a processor have diffs applied to them before the inner loop is performed. We are consequently required to update only the values of the array elements. The algorithm ensures that pages for which we have seen write notices but no updates will be invalidated at the end of the entire computation, but not between iterations of the time step.

3.5 Data Layout

Lazy release consistency with multiple writers mitigates some of the problems that can be caused by false sharing. However, improving locality of data accesses remains important. This is because updating pages by means of diffs in the presence of multiple writers is an expensive operation on distributed memory machines.

Unlike a message passing programming model, explicit data distribution among the processors is not relevant

```
Given:   Scatter_ϕ(mem(x1:x2))
$\phi$ = Plus | Times | Min | Max | MinLoc | MaxLoc
$t_p$ = my processor number
Compute:  Scatter_ϕ operation on TreadMarks
Algorithm:
Each $e \in R$ is a list of indices s.t.  owner(e) != $t_p$
  for each $e \in R$ do
    send diffs to owner(e)
  endfor
Each $e \in S$ is a list of indices s.t.  sender(e) == sender_p
  for each $e \in S$ do
    receive diffs from sender_p
    combine partial values for memory locations of e
    using operator $\phi$
  endfor
```

Figure 7 Supporting a Scatter_ϕ on TreadMarks

in a DSM system. However, data can be laid out in shared memory to increase the locality of accesses. Granston and Wijshoff [19] discuss several methods to improve data layout in DSM systems so that false sharing is minimized. In the best case, each data item is assigned to its own page(s). Unfortunately, this is impractical because fragmentation can be very high. Another optimization that is suggested remaps arrays based upon changing access patterns in sections of the programs. This approach can be expensive in a software DSM. We propose to address the issue of changing access patterns using our DSM_send and DSM_recv primitives. An example of its use is provided in Section 4.1. Array padding [10] can be used to ensure that no two rows of an array are mapped to a single page. However, if the rows are much smaller than pages, this strategy can result in severe fragmentation problems.

We investigate the effects of data layout based upon the tiling method for Cholesky.

3.6 Iteration Space Tiling

There is on going research at several institutions, including Rice University [13], to build a tool that automatically inserts data mapping directives in user programs. Directives like *align* and *distribute* are used by the compiler to tile the iteration space as well as to improve data locality by appropriately distributing the data among the processors.

In DSM systems, physical distribution of data among the processors has no relevance. What is relevant however, is that the iteration space be tiled in such a way that a processor accesses elements on the same page(s). Granston and Wijshoff [19] discuss tiling the iteration space using a page sensitive algorithm. They assume that the underlying system does not permit multiple writers, or if so, it does in a very limited set of circumstances. The example problem discussed in the paper is shown below. Note that this loop is a doall loop.

```
    do i = 1,n
      A[$\alpha \times i$] = ...
```

```
        enddo
```

If the iteration space is tiled using `i` as the index, and a canonical data layout is used, false sharing among the processors can be a problem, particularly if α spans a page. The idea is to tile the iteration space taking α into account so that a processor accesses all its data on a subset of the pages and minimal false sharing occurs. The block size and index computations are generalized to the case where the array may not start at a page boundary.

We also use a page sensitive strategy to tile the iteration space. However, in many problems tiling along page boundaries may in itself be inadequate. For instance, in Cholesky factorization described in detail in Section 4.2, a blocked tiling minimizes diffs but produces poor load balance, a cyclic distribution may improve load balance at the expense of diffs, while a block-cyclic tiling may provide the best tradeoff. If the block-cyclic tiling can be performed in such a way that the blocks are on disjoint sets of pages, the overhead due to diffs may be further reduced. Thus, the manner in which the computation proceeds is an important parameter for tiling.

The compiler may select a particular tiling for a loop nest. This tiling may or may not be the one that is optimal for the problem. Several compilers permit the user to provide an optional directive that forces a particular tiling strategy [25]. Because our approach is to build on current compiler technology, we will permit the user to provide such a directive.

4 Representative Kernels

In this section, we present four important kernels from the fields of numerical analysis and molecular dynamics. These kernels are used to illustrate the shortcomings of software DSMs as they currently exist. Using these kernels, we demonstrate the potential benefits of the optimizations described in the previous section.

4.1 ADI

In the ADI example depicted in Figure 9, the first `j` loop is a parallel loop and `i` loops enclosed inside the `j` loops are sequential. In order to execute the first `j` loop nest in parallel, the iteration space is tiled block wise. Hence, we have each processor assigning to a block of columns. We assume data is laid out in column major order as is the case with Fortran. This tiling results not only in parallelism but also in high data locality, as shown in Figure 8.

In order to parallelize the second loop nest, shared memory compilers will try to obtain parallel outer loops and may achieve this via loop transformations which include distribution, interchange and fusion to produce the parallelized version shown in Figure 9. The problem with the parallelized code is that spatial locality is sacrificed at the cost of parallelism. Figure 8 shows the reference pattern for array `x`. Every processor writes to all pages on which array `x` resides.

With such a reference pattern, in a lazy software DSM system, every processor will be interrupted $(P-1)*w_{tp}$ times, where w_{tp} is the number of pages written by processor t_p for array `x`. In an eager software DSM system, every processor will send diffs for w pages to every other processor. In both cases communication overhead is very high. We intend to reduce the communication by sacrificing some amount of parallelism and using DSM_Send_Recv to minimize unnecessary communication traffic. The resulting loop nest exhibits pipeline parallelism and has high spatial locality [22]. The loop nest with section locks and pipeline parallelism is shown in Figure 10. Experiments conducted on the iPSC/860 at Rice University have shown that pipelining

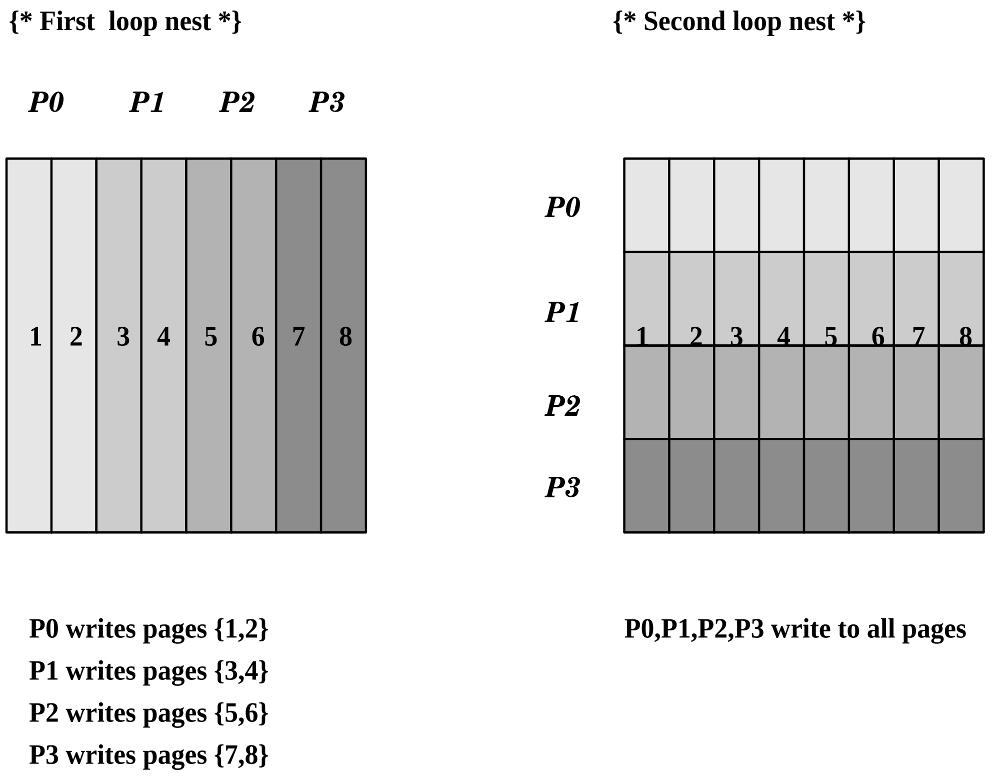

Figure 8 Page reference pattern for array `x` in ADI

the computation achieves good speedups [23].

4.2 Cholesky Factorization

In Figure 11, we depict cholesky factorization with a block-cyclic tiling. The b loop iterates over each block for every processor, and the i loop iterates over the blocks for each processor. The function *lower_block_cyclic* returns the lower bound for the i loop.

In order to obtain the the updated pivot element `b(j,j)`, a barrier must be inserted after the write to the pivot element. This ensures that all processors will read the pivot element that has been written by $p_{((j-1)/bsize \bmod P)}$. In an eager system, when a barrier occurs, diffs for *all* pages are sent to every processor. This may cause significant amount of communication to occur since diffs for pages that a processor may never read are also sent to that processor. On the other hand, in a lazy system, when a barrier occurs, all processors send *write notices*, a list of pages they have modified, to all other processors. Every processor then, invalidates the pages that have been modified by other processors. When a processor accesses an invalid page, it requests the diffs from all processors that sent write notices for that page. Clearly, the lazy system eliminates unnecessary communication by sending diffs only when needed.

However, in our example above, when a processor besides $p_{((j-1)/bsize \bmod P)}$ tries to read the pivot element, in TreadMarks, a data access fault occurs since the page has been invalidated at the barrier due to a write notice from at least $p_{((j-1)/bsize \bmod P)}$, This causes an interrupt to occur on $p_{((j-1)/bsize \bmod P)}$, who then sends the diffs for the entire page to the requesting processor. Once the diffs have been received, the processor

```
{* Original Program * }
do j = 1, N
 do i=2, N
  x(i,j) = x(i,j)-x(i-1,j)*a(i,j)/b(i-1,j)
 enddo
 do i=2, N
  b(i,j)=b(i,j)-a(i,j)*a(i,j)/b(i-1,j)
 enddo
enddo
..........
do j=2, N
 do i=1, N
  x(i,j)=x(i,j)-x(i,j-1)*a(i,j)/b(i,j-1)
 enddo
 do i=1, N
  b(i,j)=b(i,j)-a(i,j)*a(i,j)/b(i,j-1)
 enddo
enddo
```

```
{* Parallelized Version * }
    doall j = 1, N
     do i = 2, N
       x(i,j)=x(i,j)-x(i-1,j)*a(i,j)/b(i-1,j)
     enddo
     do i=2, N
      b(i,j)=b(i,j)-a(i,j)*a(i,j)/b(i-1,j)
     enddo
    end doall
     ..........
    doall i=1, N
     do j=2, N
      x(i,j)=x(i,j)-x(i,j-1)*a(i,j)/b(i,j-1)
     enddo
     do j=2, N
      b(i,j)=b(i,j)-a(i,j)*a(i,j)/b(i,j-1)
     enddo
    end doall
```

Figure 9 ADI

```
{* 2nd loop nest in pipeline mode * }
myp = my processor number
begin = lower_block(2, N/P, P);
end = upper_block(N, N/P, P);
S = strip size
do i = 1, N, S
if (myp .gt.  0)
 DSM_recv(x(i:i+S, begin-1), myp - 1)
 DSM_recv(b(i:i+S, begin-1), myp - 1)
endif
do k = i, min(i+S,N)
 do j = begin, end
  x(i,j)=x(i,j)-x(i,j-1)*a(i,j)/b(i,j-1)
 enddo
 do j = begin, end
  b(i,j)=b(i,j)-a(i,j)*a(i,j)/b(i,j-1)
  enddo
if (myp .lt.  P-1)
 DSM_send(x(i:i+S, end), myp + 1)
 DSM_send(b(i:i+S, end), myp + 1)
endif
enddo
enddo
```

Figure 10 Pipelined loop for ADI

updates and validates the page and proceeds to read the pivot element. Note that since all processors read the pivot element, $p_{((j-1)/bsize \bmod P)}$ is interrupted $P-1$ times. We are also forced to incur the overhead for write notices when a barrier occurs even though in Cholesky, the second `i` loop nest does not reference any data computed in the first `i` loop nest. The analysis to determine that the intersection of the array sections referenced in the first and second `i` loops is an empty set is successfully performed by compilers for distributed memory machines such as the Fortran D compiler. A compiler can thus use similar analyses to eliminate a barrier and instead generate a broadcast.

```
do j = 1, N
  begin = lower_block_cyclic(j, bsize, P)
  do i = begin, N-1, numblocks*bsize
    end_block = end_b(i, bsize)
    do b = i, end_block
      do k = 0, j
        a(b,j) = a(b,j) - a(b,k) * a(j,k)
      enddo
    enddo
  enddo
  if (proc_id .EQ. mod((j/bsize),P) then
    a(j,j) = sqrt(a(j,j))
  endif
  call barrier()
  t = 1.0 / a(j,j)
  begin = lower_block_cyclic(j+1, bsize, P)
  do i = begin, N, numblocks*bsize
    end_block = end_b(i, bsize)
    do b = i, end_block
      a(b,j) = a(b,j) * t
    enddo
  enddo
  call barrier()
enddo
```

Figure 11 Cholesky Factorization on TreadMarks

4.3 Inner Product

Reduction is an operation that occurs frequently in scientific applications. Inner Product, livermore kernel 3, is a 1-d reduction. On a DSM system it is very difficult to extract any parallelism from the reduction loop shown below. Due to the loop-carried dependence on the k loop, the computation is sequentialized.

```
{* sequential *}
  q = 0.0
  do k = 1,n
    q = q + z(k)*x(k)
  enddo
```

```
{* parallel version *}
  q = 0.0
  do k = begin, end
   q = q + z(k)*x(k)
  enddo
  global_sum(q, sizeof(q))
```

The inner product exhibits a race condition in TreadMarks if the loop is tiled and executed concurrently by the processors. However, as mentioned earlier, compilers are able to recognize and parallelize reductions.

The compiler will transform the above loop nest by tiling the iteration space and allowing multiple processors to write the shared variable q. Since TreadMarks allows multiple writers, we allow each processor to write to its local copy of the page. At the end of the local computation, each processor will have written to the same location, but into a local copy of the page. In order to guarantee that all processors have a consistent version of the page after the reduction has occurred, we perform a global accumulate to collect the partial sums.

4.4 Molecular Dynamics Kernel

A simplified version of an MD kernel [20] is depicted in Figure 12. The **x** and **f** (representing the distance and force components respectively) arrays are laid out serially in shared memory, ordered by atom number. We refer to this type of data layout as the canonical layout. The important point to note is that in such a computation, the atom number is not related in any way to an atom's location in space. Our example problem assumes that the simulation in performed in a 1-D space whereas actual simulations are performed in 3-D space.

4.4.1 MD Kernel Using TreadMarks

One method to parallelize such a computation for a DSM system is to tile the iteration space such that blocks of atoms (index i) are assigned to processors. The problem with such an approach is that multiple processors may update the same array element in a particular time step. We, therefore, need locks to serialize these updates.

The MD kernel implemented using TreadMarks primitives is shown in Figure 13. Due to the locks, the computation will proceed in a sequential manner. Furthermore, at each acquire, the acquiring processor will receive write notices from the previous holder of the lock. The acquirer will invalidate the appropriate pages. On accessing an invalidated page, diff requests will be generated in order to get the most recent values of the `f` array. The overhead associated with this is significant.

4.4.2 The MD Kernel Using Inspector-Executor

The inspector for the MD kernel is summarized as follows: Based upon the loop partitioning, each processor knows the array elements it owns. Moreover, based upon the inb array, the inspector can determine the array elements each processor will update that are owned by other processors. By "owner" we mean the processor that is responsible for performing the global accumulation on the array element. The sends are set up on each processor by the inspector code, followed by the receives.

The executor loop is depicted in Figure 13. The lock in the inner loop is no longer needed because of the scatter_ϕ transformation generated by the compiler. In TreadMarks, multiple writes are allowed to occur for each variable. These writes occur to local copies of the pages on each processor.

After the loop, an explicit incomplete reduction operation is performed using the **scatter_ϕ** function on all variables that have been written by more than one processor. The reduction is referred to as incomplete because it involves only a subset of the processors (i.e., only those processors that actually write a particular

```
do t = 1, TimeSteps
 do i = 1, Natoms
  do p = 1, inb(i)
   j = partners(i,p)
   force = nbforce(x(i),x(j))
   f(i) = f(i) + force
   f(j) = f(j) - force
  enddo
 enddo
enddo
```

Figure 12 MD Kernel

```
{* Shared Memory Version *}

do t = 1, TimeSteps
  begin = lower_block(1, bsize, P)
  end = upper_block(N, bsize, P)
  do i = begin, end
    do p = 1, inb(i)
      j = partners(i,p)
      force = nbforce(x(i),x(j))
      call lock_acquire()
      f(i) = f(i) + force
      f(j) = f(j) - force
      call lock_release()
    enddo
  enddo
enddo
```

```
{* Scatter_φ Primitive *}
   ----Perform Inspector----
 do t = 1, TimeSteps
   begin = lower_block(1, bsize, P)
   end = upper_block(N, bsize, P)
   do i = begin, end
     do p = 1, inb(i)
       j = partners(i,p)
       force = nbforce(x(i),x(j))
       f(i) = f(i) + force
       f(j) = f(j) - force
     enddo
   enddo
   call scatter_φ(f, sizeof(f))
 enddo
```

Figure 13 Shared memory version versus Scatter_ϕ version

variable).

5 Performance Results

In this section, we describe the performance impact of the different optimizations we have described in Section 3 on the representative kernels. All our tests were conducted using TreadMarks on an ATM network of DECstations and the Intel Paragon at Rice University.

5.1 Experimental Environment

The ATM Network

Our ATM network consists of 8 DECstation-5000/240s running Ultrix V4.3. These processors have a 100×100 Linpack performance of 5.3 Mflops [16]. Each of these machines has a Fore Systems ATM board, connected to an ATM switch. The interface boards have a capacity of 100 Mb/sec and the switch has an aggregate capacity of 1.2 Gb/sec. The Fore Systems ATM boards perform programmed I/O into transmit and receive FIFOs. For performance reasons, TreadMarks employs a low-level adaptation layer protocol AAL4 for transmitting ATM packets rather than IP.

The Intel Paragon

The Paragon at Rice University, consists of 56 compute nodes and 5 control nodes running OSF/1.1 MK-AD. The compute nodes are built out of Intel's i860XP microprocessors, running at 50 Mhz, and rated at 50 Mflops. 100×100 Linpack performance is roughly 15 Mflops. The Paragon architecture is a 2-D mesh with channels capable of up to 200MB/sec connecting the processors. However, the current machine has channels running at 30 MB/sec, with 75 MB/sec channels due in early 1994. Wormhole routing is used to transfer messages between processors. In the design of the machine, each node has a Message Co-processor (MCP) for high speed message passing. However, the MCP option has not yet been enabled. The TreadMarks implementation on the Paragon uses the NX message passing library, which has been incorporated in the OSF/1 kernel. Each TreadMarks process (which runs on one node of the Paragon) consists of two POSIX threads, the first to execute user applications, and the second to handle TreadMarks requests from other processes.

5.2 Execution Times and Speedups

In this section, we present performance results from running ADI, Cholesky, and the MD kernel on TreadMarks as it is currently implemented. We are currently enhancing TreadMarks to provide broadcasts, reductions, section locks and scatter_ϕ type operations and will be able to provide concrete performance results before the final version of this paper is due for the conference.

5.2.1 ADI

We ran ADI using TreadMarks on our ATM network of DECstations. Results of these experiments are presented in Table 1. We present actual execution times for 10 iterations of the ADI loop. Rows were assigned to processors in a blocked manner. We were unable to run larger problems because of certain restrictions in TreadMarks related to the amount of space allocated for storing diffs. These restrictions are being addressed. The main point to note is that no speedups are achieved in the problem sizes we tested. We expect pipelining the computation and using DSM_Send_Recv to improve the performance.

Problem Size	Number of processors	Execution Time (secs)
64×64	1	0.57
	2	1.55
	4	2.28
	8	4.23
128×128	1	3.98
	2	6.74
	4	18.66
	8	24.25
256×256	1	18.46
	2	28.49
	4	37.38
	8	-

Table 1 ADI Execution Times on DECstations

5.2.2 Cholesky

We constructed three versions of cholesky as follows.

- the iteration space is tiled so that each processor executes a block of rows. We use the functions below to compute the tile on each processor

```
$N$ = Array Size, $P$ = number of processors
$lb$ = lower loop bound, $l$ = lower array bound
$ub$ = upper loop bound, $u$ = upper array bound
$t_p$ = this processor (processors are numbered 0...P-1)
$bsize = N/P$,
$begin = max((t_p * bsize) + l, lb)$
$end = min((t_p + 1) * bsize + l - 1, ub)$
```

- the iteration space is tiled so that each processor executes rows in a cyclic fashion. We use the functions given below to compute the tiles :

```
if ($t_p < (lb - l) \bmod$ P) then
```

```
        $begin$ = $\lceil \frac{lb - l}{P} \rceil * P + t_p + l$
    else
        $begin$ = $\lfloor \frac{lb - l}{P} \rfloor * P + t_p + l$
    endif

    if ($t_p < (ub - l) \bmod$ P) then
        $end$ = $\lfloor \frac{ub - l}{P} \rfloor * P + t_p + l$
    else
        $end$ = $\lfloor \frac{ub - l}{P} \rfloor * P - P + t_p + l$
    endif
```

- the iteration space is tiled so that each processor executes a *block* of rows in a cyclic fashion. We use the functions given below to compute the tiles :

```
if ($t_p == \lfloor \frac{lb-l}{bsize} \rfloor \bmod P$) then
    $begin = lb$
else if ($t_p < \lfloor \frac{lb-l}{bsize} \rfloor \bmod P$) then
    $begin = \lceil \frac{lb-l}{bsize*P} \rceil * (bsize * P) + t_p * bsize + l$
else
    $begin = \lfloor \frac{lb-l}{bsize*P} \rfloor * (bsize * P) + t_p * bsize + l$
endif

if ($t_p == \lfloor \frac{ub-l}{bsize} \rfloor \bmod P$) then
    $end = ub$
else if ($t_p < \lfloor \frac{ub-l}{bsize} \rfloor \bmod P$) then
    $end = \lfloor \frac{ub-l}{bsize*P} \rfloor * (bsize * P) + (t_p + 1) * bsize + l - 1$
else
    $end = (\lfloor \frac{ub-l}{bsize*P} \rfloor - 1) * (bsize * P) + (t_p + 1) * bsize + l - 1$
endif
```

Table 2 depicts the execution times for Cholesky on 8 DECstations using block, cyclic, and block-cyclic tiling of the iteration space. Problem sizes are depicted in the first column of the table. For the block-cyclic tiling, the numbers accompanying the execution times are the tiling sizes which yielded the best performance for that problem size. From this table, we can see that for small problem size (64×64), none of the tiling schemes provide any execution time improvement. Block tiling performs well for the small problem sizes but for large problems, due to load imbalance problems, it fares badly compared to both cyclic and block-cyclic tiling. Cyclic tiling yields poor results for small problem sizes because it results in multiple writers for each page and hence low locality and concomitant exchange of *diffs* between the processors. For large problem sizes, when a column (or row) of a data array occupies a whole page, the differences between the cyclic and block-cyclic tilings vanish. To summarize, block-cyclic tiling provides the best performance because it provides high locality and avoids load imbalance; though picking the correct tiling size is a non-trivial problem.

We believe that implementing broadcasts in TreadMarks will have a significant impact on the performance of Cholesky. Tests similar to those depicted in Table 2 conducted on the Intel Paragon showed much worse performance characteristics. This is due to several reasons, including the architecture of the i860XP processor which makes interrupts expensive to handle, problems with thread scheduling, and relatively slow communication channels, as described above.

5.2.3 MD Kernel

We ran the MD kernel on the network of DECstations and the Intel Paragon. The results of this kernel for various problem sizes showed that the sequential execution was faster than the parallel executions. This is because of the computation was serialized owing to the lock, and the overhead associated with transferring diffs between processors.

We have implemented the scatter_ϕ optimization using the second approach mentioned in Section 4.4.2. In order to exploit locality so that once a page is acquired by a processor, all the modified array elements belonging to the page are updated together, the inspector sorts the non-local accesses according to the "owner" processor number. The scatter_add routine then acquires a lock corresponding to a processor and updates all the data belonging to the processor before releasing the lock. Each processor, in turn, adds its partial sum to the appropriate globally shared array elements. To ensure that after the loop all the processors access the updated values, a barrier synchronization is needed. Note that, in some cases, it might be possible to replace this barrier with locks among the subset of processors which actually modify the particular array elements. Figure 14 shows the performance improvements we achieved using the scatter_add routine for the MD kernel.

6 Related Work

Methods for reducing the performance degradation due to false sharing of pages in shared memory programs have been studied by several researchers. Bolosky et. al. [10] modified a Mach kernel running on the IBM ACE multiprocessor (a NUMA machine) to allow the notion of replicated, private and global pages. Replicated pages were either read-only pages or writeable pages that were not written. Private pages migrated to the processor that was writing them, while global pages were those that were often written by more than one processor. The actual packing of structures to fit in a page (or a set of pages) so as to reduce false sharing was done manually through modifications to the program's data structures. No compiler support was available for iteration space tiling or array layouts.

Projects such as KOAN [28] and others [9] perform optimizations to reduce the effect of false sharing on software DSM systems. One of the problems that some DSM systems face is the ping-pong problem, where a page that is concurrently written by multiple writers moves rapidly between the processors. KOAN tries to minimize this problem by introducing run-time system directives that permit certain regions of data to have weak coherency. Modifications are merged at the end of the period. TreadMarks allows multiple writers by

Problem Size	Sequential Execution	Block			Cyclic			Block-cyclic		
		Number of processors (time in secs)								
	(in secs)	2	4	7	2	4	7	2	4	7
64×64	0.92	1.19	1.24	1.38	1.92	3.35	5.75	1.14 (64)	1.22 (64)	1.34 (64)
128×128	7.51	7.49	6.01	5.17	15.31	26.28	-	6.95 (32)	5.52 (16)	5.08 (16)
256×256	61.08	55.72	-	-	114.18	71.74	52.23	43.70 (32)	-	-
512×512	488.79	437.21	293.12	203.72	252.56	131.59	79.38	251.76 (2)	131.28 (1)	80.61 (2)
$1K \times 1K$	3992.86	3488.54	2309.83	2098.06	2013.87	1031.30	600.31	2018.12 (16)	1027.15 (4)	599.18 (1)

Table 2 Cholesky Execution Times on DECstations

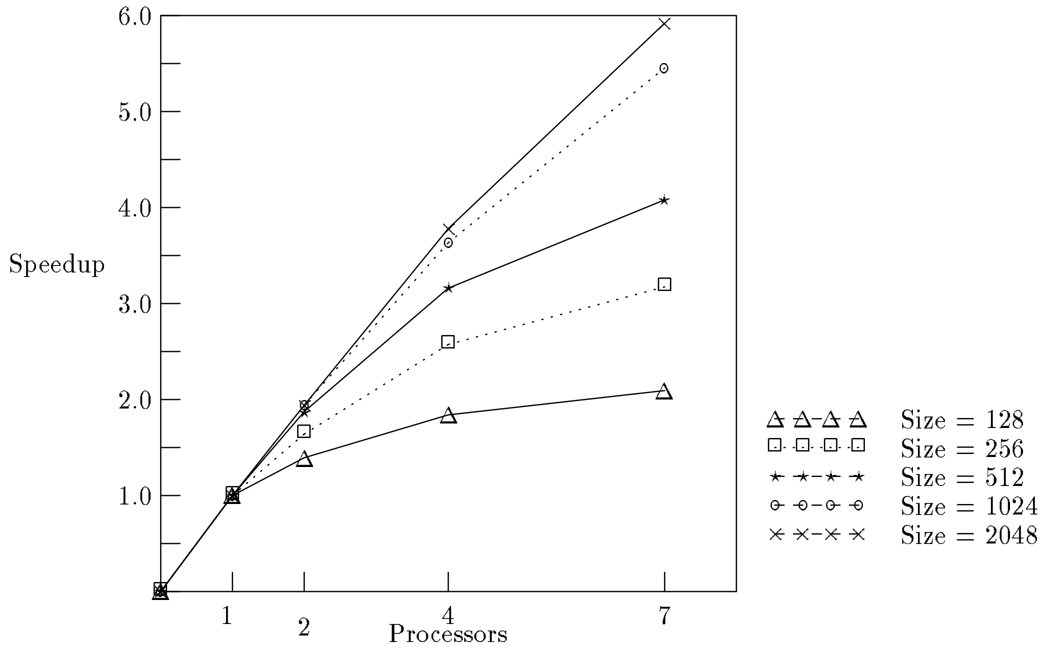

Figure 14 Speedup for MD kernel using Scatter_ϕ

default and hence the ping-pong problem does not occur. Furthermore, modifications to pages are not visible to another processor until a release is performed and the processor tries to access the page. This minimizes the amount of state information that is transferred.

In the area of compile time optimizations, Granston et. al. present loop transformations that encourage processors to perform multiple writes to a page before relinquishing the page. They also present transformations that tile the iteration space based upon page boundaries, and optionally perform data layout optimizations to reduce false sharing.

Our research is substantially different from those described above. Our compiler optimizations target a variety of scientific applications. We propose extensions such as broadcasts, barrier broadcasts, global reductions, DSM_send_recv, section locks and scatter_ϕ to DSM systems. Our approach incorporates mechanisms to handle iteration space tiling and data layout of arrays. Furthermore, we are able to use current compiler technology to perform program transformations to automatically insert DSM_send_recv, broadcasts, global reductions, and scatter_ϕ operations in user programs.

7 Summary

Advanced compilation techniques applied to programs for DSM systems have the potential of significantly improving performance. We have identified key compiler optimizations such as tiling, data layout, broadcasts, barrier broadcasts, reductions, DSM_send_recv and section locks that may be performed on shared memory programs. In addition, we have described the extensions to an underlying eager or lazy software DSM system. Our experience has shown that a compiler can effectively analyze scientific programs exhibiting regular access patterns.

We believe that compiler optimizations coupled with an advanced DSM system will make networks of high performance workstations and distributed memory machines such as the Paragon easier to program. Clearly, for such architectures to be successful, it is imperative that users be able to program these machines with ease and also obtain reasonable performance. Our optimizations will significantly contribute to improving the performance of DSM hardware and software systems. In addition, we have shown that compiler technology for performing our optimizations exist and hence the user need not perform these optimizations by hand.

8 Acknowledgements

The authors thank Ken Kennedy for his support, Alan Cox, Sandhya Dwarkadas, Pete Keleher, and Willy Zwaenepoel for allowing us access to TreadMarks and for helpful discussions, Elana Granston and John Mellor-Crummey for useful suggestions, and Reinhard von Hanxleden for helpful discussions about the molecular dynamics code fragment and for proofreading the paper. This research was supported by the Center for Research on Parallel Computation (CRPC), a Science and Technology Center funded by NSF through Cooperative Agreement Number CCR-9120008.

References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. Technical Report CS-1051, Univerity of Wisconsin, Madison, September 1991.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[4] J. Anderson and M. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN '93 Conference on Program Language Design and Implementation*, Albuquerque, NM, June 1993.

[5] B. Appelbe and B. Lakshmanan. Program transformations for locality using affinity regions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[6] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[7] J. Bennett, J. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, WA, March 1990.

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

[9] M. Beternitz, M. Lai, V. Sarkar, and B. Simon. Compiler solution for the stale data and false sharing problem. Technical Report TR03.466, IBM, Santa Teresa Laboratory, April 1993.

[10] W. Bolosky, R. Fitzgerald, and M. Scott. Simple but effective techniques for NUMA memory management. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, Litchfield Park, AZ, December 1989.

[11] R. Bryant, P. Carini, H-Y Chang, and B. Rosenburg. Supporting structured shared virtual memory under mach. In *Proceedings of the 2nd Mach Usenix Symposium*, November 1991.

[12] D. Callahan. *A Global Approach to Detection of Parallelism.* PhD thesis, Dept. of Computer Science, Rice University, March 1987.

[13] A. Carle, K. Kennedy, U. Kremer, and J. Mellor-Crummey. Automatic data layout for distributed-memory machines in the D programming environment. In *Proceedings of AP'93 International Workshop on Automatic Distributed Memory Parallelization, Automatic Data Distribution and Automatic Parallel Performance Prediction*, Saarbrücken, Germany, March 1993.

[14] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[15] R. Das and J. Saltz. Parallelizing molecular dynamics codes using parti software primitives. In *Parallel Processing for Scientific Computation, Proceedings of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, Norfolk VA, March 1993*, 1993.

[16] J. Dongarra. Performance of various computers using standard linear equations. Technical Report CS-89-93, University of Tennessee, February 1993.

[17] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[18] B. Fleisch and G. Popek. Mirage: A coherent distributed shared memory design. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, December 1989.

[19] E. Granston and H. Wishoff. Managing pages in shared virtual memory systems: Getting the compiler into the game. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993.

[20] R. v. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[21] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[22] 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.

[23] S. Hiranandani, K. Kennedy, and C. Tseng. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93*, Portland, OR, November 1993.

[24] P. Keleher, A. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[25] Kendall Square Research, Waltham, MA. *KSR1 Principles of Operation*, revision 6.0 edition, October 1992.

[26] Kuck & Associates, Inc. *KAP User's Guide*. Champaign, IL 61820, 1988.

[27] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[28] Z. Lajormi and E. Priol. Koan: Shared memory for the ipsc/2 hypercube. In *CONPAR-VAPP 92*. Springler-Verlig, September 1992.

[29] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[30] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[31] S. K. Reinhardt, J. L. Larus, and D. A. Wood. Tempest and typhoon : User-level shared memory. In *Proceedings of the 21th International Symposium on Computer Architecture*, 1994.

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

[33] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, April 1990.