Compiler and Software Distributed Shared Memory Support for Irregular Applications

Honghui Lu†, Alan L. Cox‡, Sandhya Dwarkadas◇,
Ramakrishnan Rajamony†, and Willy Zwaenepoel‡†

† Department of Electrical and Computer Engg
‡ Department of Computer Science
Rice University
{hhl, alc, rrk, willy}@cs.rice.edu

◇ Department of Computer Science
University of Rochester
sandhya@cs.rochester.edu

Abstract

We investigate the use of a software distributed shared memory (DSM) layer to support irregular computations on distributed memory machines. Software DSM supports irregular computation through demand fetching of data in response to memory access faults. With the addition of a very limited form of compiler support, namely the identification of the section of the indirection array accessed by each processor, many of these on-demand page fetches can be aggregated into a single message, and prefetched prior to the access fault.

We have measured the performance of this approach for two irregular applications, moldyn and nbf, using the TreadMarks DSM system on an 8-processor IBM SP2. We find that it has similar performance to the inspector-executor method supported by the CHAOS run-time library, while requiring much simpler compile-time support. For moldyn, it is up to 23% faster than CHAOS, depending on the input problem's characteristics; and for nbf, it is no worse than 14% slower. If we include the execution time of the inspector, the software DSM-based approach is always faster than CHAOS. The advantage of this approach increases as the frequency of changes to the indirection array increases. The disadvantage of this approach is the potential for false sharing overhead when the data set is small or has poor spatial locality.

1 Introduction

Inspector-executor methods have been proposed as a way to efficiently execute irregular computations on distributed memory machines [18]. A separate loop, the *inspector*, precedes the actual computational loop (called the *executor*). The inspector loop determines the data read and written by the individual processors executing the computational loop. This information is then used to compute a *communication schedule*, moving the data from the producers to the consumers at the beginning and/or end of each loop. Communication aggregation is used to reduce the number of messages exchanged. In order to further reduce overhead, an attempt is made to execute the inspector loop only once for a large number of iterations of the executor loop. It has been argued that part or all of the above procedure can be automated by a compiler [21]. The compiler analysis involved can, however, be quite complicated [1, 6, 20].

In this paper we propose an alternative approach. We use a software *distributed shared memory* (DSM) layer that provides a shared memory interface on top of the message layer [13]. In its simplest form, the DSM layer supports irregular computations by demand-driven fetching of data. Our approach involves, in addition, a simple compiler front-end that generates data access information, enabling the run-time system to efficiently precompute the set of pages that will be accessed by each processor during the next iteration. These pages are then requested, prior to that iteration, in a single message exchange with each processor from which data is needed. In other words, our approach extends the base software DSM layer by enabling it to aggregate the communication of data for irregular programs.

The compiler support required for our approach is very simple: it suffices to determine the indirection array, and the part of the indirection array being accessed by each processor. This is usually a regular section [4]. In contrast, the inspector-executor approach requires complex analysis to determine whether the inspector loop can be hoisted out of the main loop [1, 20].

This paper presents our approach in detail. In order to gather experimental results, we use a modified version of TreadMarks [2] that supports prefetching and aggregation in the manner described above. Furthermore, we have augmented the Parascope parallel programming environment [12] to carry out the required compiler analysis. We present performance results for two irregular applications, moldyn and nbf. The results were obtained on an 8-processor IBM SP2 using base TreadMarks and TreadMarks with aggregation support. We compare these results to measurements of hand-coded inspector-executor versions of the same applications that use the CHAOS run-time library [7].

We find that TreadMarks augmented with compiler support for communication aggregation in irregular programs has similar performance to the inspector-executor method supported by the CHAOS run-time library. For moldyn, it is up to 23% faster than CHAOS, depending on the input problem's characteristics; and for nbf, it is no worse than 14% slower. In addition, it is up to 38% faster than the base TreadMarks system. If we include the execution time of the inspector, our approach is always faster than CHAOS. The advantage of our approach increases as the frequency of changes to the indirection array increases. Its disadvantage

To appear in the 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), June 1997, Las Vegas, NV

is the potential for false sharing overhead when the data set is small or has poor spatial locality.

The outline of the rest of the paper is as follows. Section 2 presents some background on the basic run-time protocol used to implement shared memory, specifically, the TreadMarks implementation. Section 3 presents the run-time and compiler support for irregular applications. Section 4 provides a summary of CHAOS, the leading run-time system for support of irregular applications on message passing platforms. Section 5 presents the results of our evaluation of the shared memory run-time support, and compares the results to those from CHAOS. Section 6 provides a summary of related work. Finally, we conclude in Section 7.

2 Background - TreadMarks

TreadMarks [2] is a software DSM system built at Rice University. It is an efficient user-level DSM system that runs on commonly available Unix systems. We use TreadMarks version 1.0.1 as the base shared memory run-time system in our experiments.

TreadMarks provides programming primitives similar to those used in hardware shared memory machines, namely, process creation, shared memory allocation, and lock and barrier synchronization. Shared memory must be allocated dynamically using TreadMarks primitives that have the same syntax as conventional memory allocation calls. A barrier stalls the calling processor until all processors in the system have arrived at the same barrier. Locks are used to control access to critical sections. No processor can acquire a lock if another processor is holding it.

TreadMarks uses a *lazy invalidate* [2] version of *release consistency* (RC) [9] and a multiple-writer protocol [5] to reduce the overhead involved in implementing the shared memory abstraction.

RC is a relaxed memory consistency model. In RC, *ordinary* shared memory accesses are distinguished from *synchronization* accesses, with the latter category divided into *acquire* and *release* accesses. RC requires ordinary shared memory updates by a processor p to become visible to another processor q only when a subsequent release by p becomes visible to q via some chain of synchronization events. In other words, to ensure that changes to shared data are visible, a program must use explicit synchronization. In practice, this model allows a processor to buffer multiple writes to shared data in its local memory until a synchronization point is reached.

The virtual memory hardware is used to detect accesses to shared memory. Consequently, the consistency unit is a virtual memory page. The *multiple-writer protocol* reduces the effects of false sharing with such a large consistency unit. With this protocol, two or more processors can simultaneously modify their own copy of a shared page. Their modifications are merged at the next synchronization operation in accordance with the definition of RC, thereby reducing the effects of false sharing. The merge is accomplished through the use of *diffs*. A diff is a run-length encoding of the modifications made to a page, generated by comparing the page to a copy saved prior to the modifications (called a *twin*).

TreadMarks implements a *lazy invalidate* version of RC [2]. A *lazy* implementation delays the propagation of consistency information until the time of an acquire. Furthermore, the releaser notifies the acquirer of which pages have been modified, causing the acquirer to *invalidate* its local copies of these pages. A processor incurs a protection violation on the first access to an invalidated page, and gets diffs for that page from the most recent modifiers of the page.

3 Optimizations for Irregular Applications

The run-time system described in Section 2 performs communication purely on demand. This can result in extra messages due to the fact that data is brought in at a page granularity, and additional run-time overheads in terms of page faults and interrupts in order to trigger the communication [8, 14]. This section describes the enhancements to the TreadMarks run-time system as well as the compiler analysis necessary in order to optimize performance for programs with irregular access patterns.

Our approach involves a simple compiler front-end that identifies the indirection array(s) to the run-time system. Specifically, the compiler performs a source-to-source transformation of the program, inserting calls to the (augmented) run-time DSM library before the indirect accesses. These calls identify the base addresses of the data arrays, and the sections of the indirection arrays accessed by a particular processor. The run-time system then uses this information to determine the set of shared pages that this processor accesses. The pages are requested in a single message exchange with each of the processors from which data is required.

In order to avoid having to recompute the set of pages accessed on every iteration, the run-time system subsequently write-protects the shared pages containing the indirection array. If no memory protection violation occurs for these pages, then the same set of pages are requested in the next iteration. Otherwise, the indirection array has been changed, and the set needs to be recomputed.

3.1 Example

We first illustrate our approach with an example using the moldyn program [3] (see also Section 5). Figure 1 illustrates the program structure of moldyn, and the force computation subroutine in which the indirect accesses occur. Each molecule `i` has two attributes: its position, `x(i)`, and the force, `forces(i)`, acting on it.

Figure 2 shows the program transformations applied to the force computation subroutine. In the transformed version, each processor first accumulates its contributions to `forces` in the `local_forces` array that is stored in private memory. After this computation, the processors update the shared forces in a pipelined fashion. This reduces communication by eliminating the need to synchronize on every access to `forces` and by aggregating the updates to `forces`.

The compiler-inserted code consists of a `Validate` at the start of the `Compute_Forces` subroutine. The `Validate` initializes the data structures for the fetch. Then, if necessary, it computes the pages accessed through the indirection array `interaction_list`. Finally, it requests the updates to each page of `x` that will be accessed by the executing processor. To improve performance, `Validate` aggregates requests for multiple pages from the same processor.

3.2 Augmented Run-Time System

The run-time system was augmented in order to take advantage of the access information provided by the compiler. We concentrate here on the support for communication aggregation for irregular accesses. Support for regular accesses and other optimizations was described in earlier work [8].

```
PROGRAM MOLDYN

DO step = 1, NSTEPS

   IF (mod(step,UPDATE_INTERVAL) .eq. 0) then
      call build_interaction_list()
   ENDIF
   ... ...
   call ComputeForces()
   ... ...
ENDDO
... ...
END
```

```
SUBROUTINE ComputeForces()

DO i = 1, num_interactions

    n1 = interaction_list(1, i)
    n2 = interaction_list(2, i)

    force = x(n1) - x(n2)
    forces(n1) = forces(n1) + force
    forces(n2) = forces(n2) - force

ENDDO
END
```

Figure 1: Moldyn - main program and the subroutine ComputeForces

```
SUBROUTINE ComputeForces()

    Validate(1, INDIRECT, x, interaction_list[1:2, 1:num_interactions], READ, 1)

    DO i = 1, num_interactions

        n1 = interaction_list(1, i)
        n2 = interaction_list(2, i)

        force = x(n1) - x(n2)
        local_forces(n1) = local_forces(n1) + force
        local_forces(n2) = local_forces(n2) - force

    ENDDO

END
```

Figure 2: Transformations for the Subroutine ComputeForces in Moldyn

```
/* fetch_pages is the list of pages to be fetched
   pages[sch]  is the list of pages associated with each schedule */

Validate( va_alist ) /* Handles variable number of arguments */
{
  va_list  *desc_ptr
  int      number             /* number of descriptors */
  int      descriptor

  va_start(desc_ptr)

  fetch_pages = NONE

  number = va_arg(desc_ptr, int)

  for (descriptor = 1; descriptor <= number; descriptor++)
  {
      int type        = va_arg(desc_ptr, int)     /* descriptor type - DIRECT or INDIRECT */
      char *base      = va_arg(desc_ptr, char *)  /* base address of shared data */
      RSD section     = va_arg(desc_ptr, RSD)     /* section of shared data or indirection array */
      int access_type = va_arg(desc_ptr, int)     /* READ, WRITE, READ&WRITE, WRITE_ALL, or READ&WRITE_ALL */
      int sch         = va_arg(desc_ptr, int)     /* schedule number */

      if (type == INDIRECT)
      {
          if (modified(section))
          {
              pages[sch] = Read_indices(base, section)
              Write_protect(section)
          }
      }
      else
          pages[sch] = pages in section

      fetch_pages += pages[sch] that are invalid
  }

  Fetch_diffs(fetch_pages)

  Apply_diffs(fetch_pages)

  for (descriptor = 1; descriptor <= number; descriptor++)
  {
      if (access_type == WRITE || access_type == READ&WRITE)
          Create_twins(pages[sch])
  }

  va_end(desc_ptr)
}
```

Figure 3: Augmented Run-Time Interface for Indirect Accesses

Figure 3 provides a summary of the Validate interface for both regular and irregular accesses.

To support aggregated communication, Validate can fetch multiple data objects at the same time. Thus, it takes a variable number of arguments. The first argument is the number of access descriptors that follow. There is an access descriptor for each data object. An access descriptor consists of type, base, section, access_type, and schedule_number. The type specifies the descriptor type. It is either DIRECT for regular accesses or INDIRECT for accesses through an indirection array. The base is the address of the shared data structure being accessed. The section is the section of the indirection array used to access the shared data, or the section of shared data itself in the case of regular accesses. The access_type is one of READ, WRITE, or READ&WRITE. Direct accesses have two additional access types, WRITE_ALL and READ&WRITE_ALL, which indicate when every element in the section is known to be written at compile-time. The run-time system can use this information to reduce consistency maintenance overheads by eliminating *twinning* on those pages that are completely written. The schedule_number is an identifier for the set of pages to be fetched.

If the descriptor type is INDIRECT and the section of the indirection array has been modified since the last call to Validate, the modified function returns true, and pages[sch] is recomputed. Both local and remote modifications cause the modified function to return true. The Read_indices procedure recomputes the list of pages, pages[sch], using base and section. After pages[sch] has been computed, the pages in section are write protected. A more sophisticated version of this approach could use *diffing* (comparing an old version of the pages containing the indirection array to the current one) to incrementally recompute the page sets, but our current implementation does not do so. Those pages in pages[sch] that are invalid are added to fetch_pages, the list of pages to be fetched.

Fetch_diffs requests the *diff*s required to update the pages in fetch_pages. All of the *diff* requests to the same processor are aggregated into a single message. Apply_diffs waits for the diffs to arrive, and applies them to the appropriate page.

After updating the pages of shared data, consistency actions are performed preemptively in order to avoid write detection overhead during execution. The TreadMarks multiple-writer protocol requires an unmodified copy (a *twin*) of the page to be maintained for every page that is modified, unless the page is guaranteed to be modified in its entirety. Validate performs Create_twins on pages[sch] if the corresponding descriptor has an access_type of WRITE or READ&WRITE. Create_twins makes a *twin* of each page in pages[sch], and enables write access to these pages. This avoids the memory protection violation to create the twin.

3.3 Compiler Analysis

The compiler support required for our approach involves determining the indirection array used to access shared data, and the part of the indirection array being accessed. This is usually a regular section [4], and hence can be handled by the existing compiler framework for regular accesses. Our approach also naturally extends to multiple levels of the indirection in the access pattern without additional mechanisms. In contrast, the inspector-executor approach requires several inspector loops to be generated for such access patterns [6]. Furthermore, the inspector-executor approach also requires sophisticated compiler analysis to pull the inspector as far forward as possible in the program.

We concentrate here on the additions to the analysis necessary for handling indirect accesses (see [8] for details on how regular accesses are handled). Let V be the set of shared variables, let S be the set of all synchronization operations in the program, and let F be the set of "possible fetch points", the locations in the program where a Validate may be inserted. If "perfect" analysis were possible, the set F would be equal to the set S. Indeed, under lazy release consistency, invalidations only occur at a synchronization point, and hence synchronization points are the only places in the program where it makes sense to insert a Validate. In practice, F includes the set S, but in addition includes conditional statements, loop boundaries, and, in the absence of interprocedural analysis, procedure calls.

Access analysis generates a summary of shared data accesses associated with each element of F, and the type of such accesses. Our main tool is regular section analysis [11]. Regular section descriptors (RSDs) are used to concisely characterize the array accesses in a loop nest. RSDs represent the accessed data as linear expressions of the upper and lower loop bounds along each dimension, and include stride information.

For each statement p in the program, for each definition or reference in p to an indirection array, a section is constructed. A {READ}, {WRITE}, or {READ&WRITE} tag is associated with the section depending on the access type. This section is associated with each element of F that directly precedes p.

During the program transformation phase, for each f in F, if there are access descriptors associated with f, a Validate is inserted at f. Each access descriptor is then supplied as a parameter to Validate with either a DIRECT or INDIRECT type. If the type is INDIRECT, the base address of the shared data is supplied to the Validate call, along with the RSD for the indirection array as the section parameter.

See Figure 2 for the results of the analysis and the transformations on the moldyn program. Since we do not have interprocedural analysis, the relevant fetch point is entry to the procedure ComputeForces. The sections of the indirection array interaction_list are used to fetch the corresponding page sets of the data array x. After the initial execution of Validate, interaction_list is write protected. When the interaction list is modified UPDATE_INTERVAL iterations later, a memory protection violation occurs. The handler for this memory protection violation sets a flag. During the next execution of Validate, if the flag is set, modified clears the flag and returns true, and Validate recomputes the set of pages that must be fetched.

4 CHAOS

CHAOS [19] is a run-time library designed to handle irregular applications on distributed memory machines. There are three steps in solving irregular problems in CHAOS, namely, data and iteration partitioning, the inspector, and the executor.

CHAOS supports a number of parallel partitioners that partition data arrays using heuristics based on spatial position, computational load, etc. The partitioner returns a *translation table*, which contains an irregular assignment of array elements to processors. A translation table lists the home processor and offset address of each data array element. Depending on storage requirements, the translation table can be replicated, distributed regularly, or stored in a paged fashion. This table is used by the inspector to create the communication schedule. If the translation table is not replicated, communication may be necessary in the inspector. The loop iterations are partitioned by the *almost-owner-computes* rule, which assigns an iteration to the processor that owns a majority of data array elements accessed in that iteration. The data array can be remapped, so that data elements owned by a processor are adjacent in memory. Remapping has the potential advantage that the memory requirement on a processor is proportional to the size of the data partitions assigned to it.

The Recursive Coordinate Bisection (RCB) partitioner is one specialized partitioner supported by the CHAOS library that partitions nodes according to their physical positions. When simulating physical systems, particles close to each other in the physical space are more likely to interact with each other, or to be connected with each other. RCB results in less communication than a simple BLOCK or CYCLIC partition on these applications.

Each processor executes the inspector to construct its communication schedule. A communication schedule specifies which data is communicated and which processors are involved. The inspector constructs the communication schedule by first determining the data read and written on each processor and then consulting the translation table to determine the global placement of this data according to the partition. An important optimization in the inspector is to eliminate duplication. Duplication occurs when a data array element is pointed to by many elements in the indirection array. Removing duplication can dramatically reduce the amount of data communicated. A hash table whose size is proportional to the size of the data array is employed to eliminate duplicates. Because of the time to hash the indirection array, and the time to look up the translation table, the inspector can be expensive. However, this overhead can be amortized if the indirection array remains unchanged for a long period of time.

The executor uses the communication schedule generated by the inspector to `gather` and `scatter` data. `Gather` fetches off-processor data, and `scatter` propagates modifications to off-processor data back to their owners.

5 Experimental Evaluation

We use an 8-processor IBM SP2 running AIX version 3.2.5. Each processor is a thin node with 64Kbytes of data cache and 128Mbytes of main memory. Interprocessor communication is accomplished over the IBM SP2 high-performance switch. Unless indicated otherwise, all results are for 8-processor runs.

We compare the compiler-optimized TreadMarks programs with the hand coded CHAOS programs, as well as the base TreadMarks programs. The compiler-optimized TreadMarks programs include optimizations for both regular and irregular access patterns. Tables 1 and 2 present the execution times, speedups, number of messages and the amount of data communicated at 8 processors for the two applications discussed in this paper.

5.1 Moldyn

Moldyn is a molecular dynamics simulation. Its computational structure resembles the non-bonded force calculation in CHARMM [3], which is a well-known molecular dynamics code used at NIH to model macromolecular systems. Non-bonded forces are long-range interactions existing between each pair of molecules. CHARMM approximates the non-bonded calculation by ignoring all pairs which are beyond a certain cutoff radius. The cutoff approximation is achieved by maintaining an *interaction list* of all the pairs within the cutoff distance, and iterating over this list at each timestep. The interaction list is used as an indirection array to identify interacting partners. Since molecules change their spatial location every iteration, the interaction list must be periodically updated. Figure 1 illustrates the program structure of moldyn, and the force computation subroutine.

The CHAOS program uses the RCB partitioner to assign molecules to processors. This partition lasts through the execution. When the interaction list is updated, the program must again call the inspector to identify interacting partners. This call is inserted in the main program, right after the call to subroutine `build_interaction_list`. In ComputeForces, each processor uses the schedule created by the inspector to gather remote values of `x` and `forces` before the main loop. Both `x` and `forces` are modified elsewhere, necessitating the gather. After the main loop, the processors again use the schedule to scatter values of `forces` that will be read by other processors.

The TreadMarks program also uses the RCB partitioner. The coordinate array `x` and the `forces` array are allocated in shared memory. A `Validate` on `x` that was inserted by the compiler appears at the beginning of the subroutine ComputeForces. That is, the `Validate` is before the loop over the interaction list. Changes to the interaction list are detected by write protecting the pages it occupies (inside `Validate`). An explicit inspector call is hence not needed. In ComputeForces, each processor first accumulates its contributions to `forces` in the `local_forces` array (see Figure 2) that is stored in private memory. After `local_forces` is computed, the processors update the shared forces in a pipelined fashion in *nprocs* steps. In each step, a processor updates $1/nprocs$ of the total data.

In TreadMarks, the `local_forces` array is indexed by the molecule number without any translation. Thus the `local_forces` array is proportional in size to the total number of molecules. In CHAOS, remapping creates an analog to the `local_forces` array that is proportional in size to the molecules assigned to that processor plus the molecules they interact with. For the default data set, which we used in our experiments, between 31% and 53% of the molecules interact. Consequently, remapping has little effect on the memory utilization of the CHAOS program.

5.1.1 Results

We simulated 16384 particles for 40 iterations, varying the number the times the interaction list is updated from 1 through 3. The results are presented in Table 1. The data initialization (and the data partitioning for the parallel programs) are not timed for either the sequential or parallel versions.

We first present results for the case where the interaction list is updated once, at the 20^{th} iteration. The sequential program without any calls to CHAOS or TreadMarks runs for 267 seconds. The TreadMarks execution time on a single processor is almost identical to that of the sequential program, spending only 0.4 seconds to check the indirection lists. On the other hand, the CHAOS program runs longer on a single processor than the sequential program, because it spends 6.2 seconds in the inspector.

At eight processors, the CHAOS program runs for 44.9 seconds. We were unable to use a replicated translation ta-

Update frequency		Time (sec.)	Speedup	Messages	Data (MB)
Every 20 iterations (seq = 267.2 sec)	CHAOS	44.9	6.0	15704	190
	Tmk base	42.3	6.3	62149	160
	Tmk optimized	37.7	7.1	14528	137
Every 15 iterations (seq = 365.8 sec)	CHAOS	61.7	5.9	16255	243
	Tmk base	56.4	6.5	70230	179
	Tmk optimized	48.9	7.5	14687	141
Every 11 iterations (seq = 467.3 sec)	CHAOS	78.2	6.0	16806	296
	Tmk base	68.1	6.9	71788	190
	Tmk optimized	60.4	7.7	14871	145

Table 1: Moldyn - 8 processor results. The interaction list is updated at varying intervals.

ble, owing to the amount of memory that it required. The translation table is hence distributed, necessitating communication in the calls to the inspector. In the case where the interaction list is updated at the 20^{th} iteration, the inspector is called twice, including once at the beginning of the program. Each processor spends 4.6 seconds in the inspector. Exchanging the translation tables causes the transfer of 85Mbytes of data in 878 messages.

The base TreadMarks program (without any compiler support) runs for 42.3 seconds on eight processors. TreadMarks is able to achieve a performance comparable to CHAOS because of the large problem size, and the good data locality provided by the RCB partitioner. However, the number of messages sent in TreadMarks is three times more than that in CHAOS. The reason is that TreadMarks obtains data one page at a time, while CHAOS sends all the data needed by a processor in a single message.

With the compiler optimizations, the TreadMarks running time comes down to 37.7 seconds, which is an 11% improvement over the base TreadMarks. Of this improvement, 7 percentage points come from the communication aggregation for regular accesses. The remaining 4 percentage points come from the compiler inserted call to `Validate` for the indirect accesses. The optimized TreadMarks program sends 23 Mbytes less data than the base TreadMarks program because the reductions in the base program cause multiple overlapping *diffs* to be sent for each diff request. In the optimized program, on encountering a reduction, the compiler recognizes read–write accesses to an entire regular section. It then flags via `Validate` that the entire page, and not the diff, must be sent on a diff request. This reduces the amount of data sent as compared to the base TreadMarks program. The optimized TreadMarks program spends 0.6 seconds in `Validate` to check the indirection array.

When the interaction list is updated more often, the running times increase because of the time taken to rebuild the interaction list. CHAOS suffers from having to rerun the inspector. When the interaction list is updated every 11 iterations, CHAOS spends 9.2 seconds per processor on average in the inspector, while TreadMarks spends only 0.8 seconds in scanning the indirection list. As a result, the optimized TreadMarks program is 23% faster than CHAOS.

5.2 NBF

NBF is the kernel of a molecular dynamics simulation. It is taken from the GROMOS benchmark [10]. It was previously used as an example to demonstrate compiler generated message passing programs [22]. Instead of keeping a list of pairs of interacting molecules like moldyn, nbf keeps a list of interacting partners for each molecule. The lists of partners are concatenated together, with a per molecule list pointing to the end of each molecule's partners in the partner list. For each molecule, the program goes through the list of partners, and updates the forces on both a molecule and its partner based on the distance between them. In our experiments, the partner list is static. Each molecule has approximately the same number of partners, and the partners of each molecule spread evenly in about 2/3 of the total space. Because each molecule has about the same number of neighbors, a simple BLOCK partition suffices to balance the load.

In the CHAOS program, the inspector is called at the beginning of the program, outside the loop simulating the time steps. At the start of each time step, a `gather` is called to collect the updated values of coordinates from remote processors. A `scatter` is invoked at the end of each time step to propagate the modifications to the force array.

The TreadMarks program allocates both the coordinate array and the force array in shared memory. A `Validate` is performed at the start of each time step to fetch the updated values of the coordinate array. Like moldyn, updates to the forces are accumulated in private memory. After this computation, the processors update the shared forces in a pipelined fashion. The update is performed in *nprocs* steps. In each step, a processor updates $1/nprocs$ of the total data.

For the data set which we used in our experiments, 84% of the molecules interact. Consequently, remapping yields little reduction in the memory utilization of the CHAOS program.

5.2.1 Results

We ran nbf with varying numbers of molecules for the input problem size (see Table 2). Each molecule is represented by a double precision floating point number. Each molecule has 100 partners. The distance between two adjacent partners of a molecule is about 470 molecules. The test runs for 11 iterations, of which the last 10 iterations are timed. Thus, the results include neither the time to perform the inspector in the CHAOS version nor the time for checking the partner array in the TreadMarks program.

The unmodified (original) sequential program runs for 78.3 seconds with a problem size of 64×1024. The single-processor TreadMarks execution time is almost identical to that of the sequential program, spending only 0.001 seconds in scanning the indirection array. On the other hand, the CHAOS program runs longer on a single processor than the

Problem Size		Time (sec.)	Speedup	Messages	Data (MB)
64 × 1024 (seq = 78.3 sec)	CHAOS	10.9	7.2	2014	60
	Tmk base	19.6	4.0	34421	212
	Tmk optimized	12.1	6.5	4817	68
64 × 1000 (seq = 76.5 sec)	CHAOS	10.6	7.2	2014	59
	Tmk base	19.4	3.9	36278	209
	Tmk optimized	12.3	6.2	4920	76
32 × 1024 (seq = 39.1 sec)	CHAOS	5.5	7.1	2014	30
	Tmk base	9.1	4.3	18095	106
	Tmk optimized	6.2	6.3	3851	34

Table 2: NBF Kernel - 8 processor results.

sequential program, because it spends 7.3 seconds in the inspector.

At eight processors, the CHAOS program and the optimized TreadMarks program run for 10.9 seconds and 12.1 seconds, respectively. The inspector is not included in the timing for CHAOS. The main reason for the 10% difference is that CHAOS pushes the data to the processors that will use it in one message, while TreadMarks uses request–response communication (necessitating two messages). The 13% extra data sent in TreadMarks is due to false sharing.

Although we excluded the time to run the inspector from the timing, it is important to note that at eight processors, the CHAOS program spends 5.2 seconds per processor to create the schedule. In contrast, the TreadMarks program only spends 0.3 seconds going through the indirection array.

The compiler optimizations reduce the execution time of the base TreadMarks version by 38%. Of this reduction, 34 percentage points come from optimizations in the regular part of the code, such as the pipelined reduction. These optimizations reduce both the number of messages and the amount of data sent in the program. The remaining 4 percentage points come from prefetching the data for the irregular accesses at the beginning of each time step.

Reducing the problem size to 32×1024 does not affect the relative performance of TreadMarks and CHAOS much. The difference in performance comes from TreadMarks having to request data, as in the case of the 64×1024 problem size. Changing the data set size to 64×1000, we introduce false sharing at the boundary between pairs of processors. In this case, the optimized TreadMarks program is 14% slower than the CHAOS program, because of the extra messages and data caused by false sharing. However, the cost of the inspector in CHAOS overshadows the performance loss from false sharing in TreadMarks.

6 Related Work

A large number of studies have been published on the performance of distributed shared memory and inspector-executor systems, but, to the best of our knowledge, only one paper has been published comparing the two approaches. Mukherjee et al. [16] compare the CHAOS inspector-executor system to the TSM (transparent shared memory) and the XSM (extendible shared memory) systems, both implemented on the Tempest interface [17]. Three applications are used: moldyn, unstructured, and DSMC, and the comparison is done on a 32-processor CM-5. They conclude that TSM is not competitive with CHAOS, while XSM achieves performance comparable to CHAOS after introducing several special-purpose protocols.

Our study differs from the cited paper in several aspects. First, our transparent shared memory system (TreadMarks) performs significantly better than TSM. We attribute this difference in performance to TreadMarks' use of lazy release consistency and multiple writer protocols, in contrast to the sequential consistency and single writer protocols used in TSM. Second, we use a compiler to optimize the shared memory programs, rather than relying on handcoded special-purpose protocols. As indicated in our study, the compiler analysis necessary is relatively straightforward.

Our study is also related to the many papers on prefetching and aggregation. In particular, Mowry et al. [15] use a somewhat similar strategy to prefetch and aggregate disk requests for sequential programs, and Dwarkadas et al. [8] study prefetching and aggregation for regular applications in software distributed shared memory systems.

7 Conclusions

We have described an integrated compile-time/run-time approach for executing irregular computations on distributed memory machines. This approach is based on a modified software distributed shared memory layer, and fairly simple compile-time support. The only required compile-time support is regular section analysis of the indirection arrays. Run-time support for dynamic detection of changes to the indirection array, as well as to the shared data, eliminates any unnecessary computation and communication. Furthermore, the communication by each processor is aggregated into fewer message exchanges.

We measured this approach for two irregular applications, moldyn and nbf, using the TreadMarks DSM system on an 8-processor IBM SP2. We find that it has similar performance to the inspector-executor method supported by the CHAOS run-time library, while requiring much simpler compile-time support. For moldyn, it is up to 23% faster than CHAOS, depending on the input problem's characteristics; and for nbf, it is no worse than 14% slower. The advantage of the software DSM-based approach increases as the frequency of changes to the indirection array increases. The disadvantage of this approach is the potential for false sharing overhead when the data set is small or has poor spatial locality. In addition, in both moldyn and nbf, the software DSM-based approach eliminated substantial inspector overheads. For both applications, the software DSM-based approach is always faster than CHAOS if we include the execution time of the inspector.

Acknowledgements

This work is supported in part by the National Science Foundation under Grants CCR-9410457, BIR-9408503, CCR-9457770, CCR-9502500, CCR-9521735, CDA-9502791, and MIP-9521386, by the Texas TATP program under Grant 003604-017, and by grants from IBM Corporation and from Tech-Sym, Inc. Ram Rajamony is also supported by an IBM Cooperative Fellowship.

References

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

[2] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] B.R. Brooks, R.E. Bruccoleri, B.D. Olafson, D.J. States, S. Swaminathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculations. *Journal of Computational Chemistry*, 4:187, 1983.

[4] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. *Journal of Parallel and Distributed Computing*, 5:517–550, 1988.

[5] J.B. Carter, J.K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related information in distributed shared memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

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

[7] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–479, September 1994.

[8] S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[9] 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.

[10] W.F. van Gunsteren and H.J.C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, 1988.

[11] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[12] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, 5(7), October 1993.

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

[14] H. Lu, S. Dwarkadas, A.L. Cox, and W. Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Proceedings SuperComputing '95*, December 1995.

[15] T.C. Mowry, A.K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 3–17, November 1996.

[16] S.S. Mukherjee, S.D. Sharma, M.D. Hill, J.R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed memory machines. In *Proceedings of the 5th Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[17] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21th Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

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

[19] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Interprocedural compilation of irregular applications for distributed memory machines. In *Proceedings SuperComputing '95*, dec 1995.

[20] R. von Hanxleden and K. Kennedy. Give-N-Take – a balanced code placement framework. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, June 1994.

[21] R. von Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. Compiler analysis for irregular problems in Fortran D. In *Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing*, August 1992.

[22] Reinhard von Hanxleden. Handling irregular problems with Fortran D – a preliminary report. In *Proceedings of the Fourth Workshop on Compilers for Parallel Computers*, December 1993.