Combining Compile-Time and Run-Time Support for Efficient Software Distributed Shared Memory

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

[†] Department of Computer Science, University of Rochester
[‡] Department of Electrical and Computer Engineering, Rice University
[¶] Department of Computer Science, Rice University

Abstract

We describe an integrated compile-time and run-time system for efficient shared memory parallel computing on distributed memory machines. The combined system presents the user with a shared memory programming model, with its well-known benefits in terms of ease of use. The run-time system implements a consistent shared memory abstraction using memory access detection and automatic data caching. The compiler improves the efficiency of the shared memory implementation by directing the run-time system to exploit the message passing capabilities of the underlying hardware. To do so, the compiler analyzes shared memory accesses, and transforms the code to insert calls to the run-time system that provide it with the access information computed by the compiler. The run-time system is augmented with the appropriate entry points to use this information to implement bulk data transfer and to reduce the overhead of run-time consistency maintenance.

In those cases where the compiler analysis succeeds for the entire program, we demonstrate that the combined system achieves performance comparable to that produced by compilers that directly target message passing. If the compiler analysis is successful only for parts of the program, for instance, because of irregular accesses to some of the arrays, the resulting optimizations can be applied to those parts for which the analysis succeeds. If the compiler analysis fails entirely, we rely on the run-time's maintenance of shared memory, and thereby avoid the complexity and the limitations of compilers that directly target message passing. The result is a *single* system that combines efficient support for both regular and irregular memory access patterns.

I. Introduction

Parallel programming using a shared memory platform has the advantage of ease-of-use. In contrast to message passing, the user does not have to worry about data location or have to explicitly manage communication. Unfortunately, as parallel computers move away from the uniform memory access model in order to improve scalability, this transparency of shared memory comes into question. Message passing programs, tuned to non-uniform memory access latencies, often produce better performance. Our goal is to develop a system that continues to provide the user with a transparent shared memory programming model, but underneath is capable of exploiting the hardware's message passing capabilities. We focus our work on distributed memory machines, in which the shared memory abstraction is provided entirely in software.

A software distributed shared memory (SDSM) system (e.g., [22]) provides a shared memory abstraction on a distributed memory machine using purely run-time mechanisms. During execution, a SDSM system detects shared memory accesses, handles faults by fetching the missing data, and caches data for future reference. Such a system can handle any kind of data access pattern. However, when the access patterns are predictable, the on-demand data fetching causes extra messages and consistency actions, increasing overheads and resulting in reduced performance compared to message passing.

Research and commercial compilers for parallel computing on distributed memory machines have to date targeted the underlying message passing layer directly (e.g., [3], [14]). The compiler analyzes memory access patterns to generate message passing code, which is then optimized to aggregate communication and minimize data movement. For programs with regular access patterns that can be precisely analyzed, these compile-time systems provide superior performance since they avoid the run-time overhead present with SDSM systems. However, when the access patterns cannot be analyzed precisely, the message passing code generated by the compiler becomes inefficient. In the case of irregular accesses, for example, a simplistic compiler approach would result in a broadcast of all data produced by a processor, causing large amounts of communication.

Inspector-executor methods have been proposed to deal with this problem of irregular computations on distributed memory machines [29]. A separate loop, the *inspector*, precedes the actual computational loop, called the *executor*. The inspector precomputes the data that will be accessed by the individual processors when executing the computational loop. This information is used to create a *communication schedule*, which is then used to aggregate the movement of data from the producers to the consumers at the beginning and/or end of each loop. The high cost of the

This work was supported in part by NSF grants CCR-9702466, CCR-9705594, CCR-9410457, 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.

inspector is amortized, when possible, by executing it only once for a set of executor iterations. A compiler algorithm to automate this procedure is described in von Hanxleden et al. [32]. However, the required compiler analysis can be quite complex ([1], [8], [31]).

Our goal is to combine the benefits of SDSM systems with those of compiler-based approaches for generating code for distributed memory systems. In the combined system, the run-time library remains the basic vehicle for implementing shared memory, while the compiler performs optimization rather than implementation. Instead of generating a message passing program directly, the compiler generates a shared memory program augmented with run-time calls that describe the data access patterns. By informing the run-time system of future shared access patterns, these calls allow the run-time system to avoid memory access detection and on-demand fetching of missing data. Furthermore, they permit the aggregation of several data fetches into a single message.

An interesting aspect of this combined system is that it efficiently supports programs with regular accesses, programs with both regular and irregular accesses, and programs with completely irregular accesses. If the accesses are completely regular, then the compiler can analyze all of them, and the resulting code is as efficient as that of hand-coded or compiler-generated message passing. If the program contains code in which an array is accessed indirectly through an indirection array, we can still analyze the (usually regular) accesses to the indirection array, and derive considerable performance improvement from that analysis. If the compiler analysis fails, the program is unmodified, and handled solely by the run-time system. The combination of a shared memory compiler and an SDSM system thus avoids the complexity of the inspector-executor approach for irregular access patterns, without compromising efficiency for regular access patterns.

We extended the Parascope parallel programming environment [19] to analyze and transform explicitly parallel programs. We use *regular section analysis* [13] to determine the shared data access patterns. The resulting regular section descriptors (RSDs) describe the accesses to the data array (in the case of regular accesses) or to the indirection array (in the case of irregular accesses). We also extended the interface ([10], [23]) to the TreadMarks [2] run-time SDSM system to take advantage of the compiler analysis.

We have measured the performance of these techniques on an 8-node IBM SP/2 for applications with both regular and irregular access patterns. Compiler optimization in conjunction with the augmented run-time system achieves substantial execution time improvements in comparison to the base run-time system, ranging from 0% to 59% on 8 processors. Performance is also comparable to that using compile-time alternatives such as Applied Parallel Research's XHPF compiler (for regular access patterns) and the CHAOS [29] inspector-executor based system (for irregular access patterns).

The outline of the rest of this paper is as follows. Section II describes the combined compile-time run-time shared memory system. Section III presents the performance results. In Section IV, we outline the applicability of our techniques to other platforms and architectures. Finally, we survey related work in Section V and conclude in Section VI.

II. The Combined Compile-Time Run-Time Shared Memory System

We first provide some background on TreadMarks [2], the run-time system we used in our implementation. We then discuss how the compiler analyzes the shared data accesses in TreadMarks programs. The run-time primitives by which the compiler informs TreadMarks of the results of its analysis are discussed next. We are then ready to describe the transformation from TreadMarks source code into code augmented by calls to these primitives. Finally, we illustrate the entire process with two sample programs.

A. The Base Run-Time Shared Memory System

TreadMarks [2] is an SDSM system built at Rice University. It is an efficient user-level SDSM 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 explicitly parallel programming primitives similar to those used in hardware shared memory machines, namely, process creation, shared memory allocation, and lock and barrier synchronization. The system supports a *release consistent* (RC) memory model [11], requiring the programmer to use explicit synchronization to ensure that changes to shared data become visible.

TreadMarks uses a *lazy invalidate* [17] version of RC and a multiple-writer protocol [6] to reduce the overhead involved in implementing the shared memory abstraction.

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*).

With the *lazy invalidate* protocol, a process invalidates, at the time of an *acquire* synchronization operation [11], those pages for which it has received notice of modifications by other processors. On a subsequent page fault, the process fetches the *diffs* necessary to update its copy.

B. Compiler Analysis

The purpose of our compiler analysis is to provide access pattern information to the run-time system. This involves not only analyzing the program statements to determine what data is accessed, but also determining at which statement in the program to supply this information to the run-time system.

To answer the latter question, we take advantage of the special role that synchronization points play in release-consistent parallel programs. First, they are the points in the execution of a program where shared data needs to be made consistent. Second, they are also the points at which it is determined what data modified on other processors needs to be reflected locally for memory to be consistent. We therefore analyze code segments between consecutive synchronization statements, and provide the run-time system with a description of the accesses in a segment at that segment's initial synchronization statement.

In practice, limitations of the analysis tool may restrict the extent to which we can implement this general principle. For instance, the presence of conditional statements or — in the absence of interprocedural analysis — procedure calls may limit the region of code for which we can summarize the shared memory access patterns. In those cases, we may need to limit analysis accordingly, and place the calls that provide the access information to the run-time system at procedure entry points or at control flow statements.

Our main tool for access analysis is regular section analysis [13]. Regular section descriptors (RSDs) concisely represent the array accesses in a loop nest. The RSDs represent the accessed data as linear expressions of the upper and lower loop bounds along each dimension, and include stride information. When indirection arrays are involved, the RSDs can be used recursively, with each indirection representing the regular section for the indirection array used. The access patterns that can be analyzed are, however, limited to linear expressions of the loop indices. In addition to the memory locations accessed, our RSDs also contain a tag indicating, among other things, whether the accesses are read or write or both. Figure 1 outlines the steps in our algorithm.

1. Create V, the set of shared variables in the program. Create S, the set of all synchronization operations in the program. Initialize F, the set of all transformation points, to S.

2. For each statement p in the program

(a) By traversing the abstract syntax tree (AST) in all possible control flow directions along which p can be reached, create the set $F_{prec}(p)$ of all the directly preceding synchronization points. If no synchronization statements are found along any one direction, include the control flow statement along that direction in F and $F_{prec}(p)$.

(b) By traversing the AST in all possible control flow directions starting from p, create the set $S_{succ}(p)$ of all possible synchronization points that directly succeed the statement.

(c) For each statement f in the set $F_{prec}(p)$,
i. Determine the location of the outermost loop that encloses p but not f or any member of the set $S_{succ}(p)$. Intuitively, this corresponds to determining the code segment between consecutive synchronization statements for which accesses must be summarized.
ii. Construct a regular section for each definition or reference, both regular and irregular, in p to a variable in V. Add a {`read`} or {`write`} tag to the section. Determine the reaching definitions for each reference to a variable in V (this can be done during the AST traversal to create $F_{prec}(p)$). If these definitions occur after f, add the `write-first` attribute to the tag.
iii. Perform a union of the resulting section, with the other sections that have already been generated for f. A union of the tags {`read`} and {`write`} is {`read, write`}. A union of the tags {`read, write-first`} and {`write`}, is {`read, write-first`}.

Fig. 1. Access Pattern Determination

C. The Augmented Run-Time System

In addition to the original TreadMarks primitives, the augmented run-time system provides two primary interfaces for use by the compiler: **Validate** and **Push**.

Validate and its variant, **Validate_w_sync**, support aggregated communication. They can fetch diffs for multiple pages with a single message exchange. `Validate_w_sync`, in addition, piggy-backs the request for diffs on the next synchronization operation. The calls provide a set of access descriptors corresponding to the RSDs obtained in the analysis (see Section II-B). The run-time system uses these descriptors to determine the set of invalid pages that will be accessed. The data for the invalid pages can then be requested in a single message exchange per processor. An additional access type parameter in the `Validate` interface allows further optimizations to avoid communication and to reduce the overhead of consistency maintenance.

Details of the interface are provided in Figure 2. An access descriptor consists of the `section`, `access_type`, and `schedule_number`. The `section`, or `RSD`, contains the following information about the accesses - the `base` address, the dimension or number of indices, followed by the `type` of access (`DIRECT` or `INDIRECT`), and either the `DIRECT` information (lower bound, upper bound, and stride), or the `RSD` for the indirection array, along each dimension. This basic structure allows us to handle any recursive indirections that might be used in a program. The `access_type` is one of `READ`, `WRITE`, or `READ&WRITE`. Shared arrays accessed directly along every dimension have two additional access types, `WRITE_ALL` and `READ&WRITE_ALL`, which are used when the compiler analysis can determine that every element in the section will be written. `WRITE_ALL` indicates that *all* data in the section will be written but not read. `READ&WRITE_ALL` indicates that *all* data will be both read and written. The run-time system uses this information to reduce consistency maintenance overheads by eliminating the creation of *twins* for such pages. In addition, since accesses marked `WRITE_ALL` are not read, the run-time system can also avoid the communication that would make such data consistent before the write.

The `schedule_number` is an identifier for the `schedule`, or the set of shared pages accessed in the section. For `INDIRECT` accesses, this set is recomputed by re-traversing the indirection array *only* if it has changed since the last time it was examined. The run-time system uses the virtual memory protection mechanism to detect any modifications to the indirection array. This eliminates the need for compile-time knowledge of when the indirection array will be modified.

Push is used to replace a barrier synchronization and to send data to a processor in advance of when it is needed. The arguments to Push are the sections of data that are written by individual processors before the barrier and read after the barrier. Details of the Push interface are also provided in Figure 2. A Push on processor P computes the intersection of the sections written by P with those that will be read by another processor, and sends the data in the intersection to the corresponding processor. P then computes the intersection of the sections written by other processors with the sections that will be read by P, and posts a receive for that data.

Unlike Validate, which does not change the underlying consistency guarantees (unless a WRITE_ALL or READ&WRITE_ALL access is specified), Push guarantees consistency only for the sections of data received through the Push. The rest of the shared address space may be inconsistent until the next barrier. Hence, Push can be used only if the compiler has determined with certainty that the processors do not read the regions of shared data left inconsistent. Given the large consistency unit, the Push directive can be useful in eliminating data communication due to false sharing. Push provides the capabilities of a message passing interface within a shared memory environment. However, unlike pure compile-time approaches, Push can be used selectively by restricting its use to a program phase where complete analysis is possible. The run-time system ensures that the entire address space is made consistent at the barrier that must terminate such a phase.

```
Validate(int num_descs,    /* number of descriptors  */
         RSD section,      /* section of shared data
                              (through indirection array
                               if necessary) */
         int access_type, /* READ, WRITE, READ&WRITE,
                              WRITE_ALL, or READ&WRITE_ALL */
         int sched_num,    /* schedule number */
         ... )

/* Similar to Validate except that the request for data is
   piggybacked on a synchronization */
Validate_w_sync( ... )

/* does not preserve consistency
   - N is the number of processors */
Push(r_section[0..N-1],    /* Sections of data read */
     w_section[0..N-1])    /* Sections of data written */
```

Fig. 2. Augmented Run-Time Interface

D. Compiler Transformations

Following the analysis described in Section II-B, the compiler transforms the program using the augmented run-time interface discussed in Section II-C. The compiler first attempts to find opportunities for using the Push interface, because this interface results in the largest performance gains. Subsequently, it tries to find opportunities to use Validate. Figure 3 describes the decision process used to determine whether Push or Validate can be applied.

For each statement f in F

1. If f is a barrier, create the set $F_{prec}(f)$ of elements of F that immediately precede f (by traversing the AST as before), and the set $F_{succ}(f)$ of elements of F that immediately succeed f.

2. If /* can a Push be applied? */
- $F_{prec}(f)$ contains one and only one barrier,
- $F_{succ}(f)$ is non-empty and contains only barriers,
- the sections associated with $F_{prec}(f)$ and f are all precise (the compiler is able to analyze all data accesses made between the two consecutive synchronization points), and
- the sections associated with $F_{prec}(f)$ contain write accesses,

then /* apply the Push transformation */
- replace f with a Push, passing as arguments, the read sections of f, and the write sections of $F_{prec}(f)$ in terms of processor identifiers (in practice, this transformation will involve the creation of functions that take the processor number as a parameter, and return the section of data accessed by that processor).

3. else if /* can a Validate be applied? */
- there are precise sections associated with f

then
- if
- f is a synchronization statement
- then
- insert a Validate_w_sync
- else
- insert a Validate
- for each precise section associated with f
- if
- the analysis for this variable is precise (no unanalyzable accesses),
- tagged as {read, write} but not {read, write, write-first},
- and refers to a contiguous range of addresses,
- then
- supply the section with access type READ_WRITE_ALL.
- else if
- the analysis for this variable is precise,
- the tag contains the attribute write-first, and
- the section refers to a contiguous range of addresses,
- then
- supply the section with access type WRITE_ALL.
- else
- supply the section with access type (READ, WRITE, or READ&WRITE) depending on the tag.

Fig. 3. Program Transformation

E. Examples

We illustrate our analysis and transformation with two examples: one with regular accesses, and one with irregular accesses through an indirection array.

E.1 Jacobi

Jacobi is an iterative method for solving partial differential equations, with nearest-neighbor averaging as the main computation (See Figure 4). The array b is shared, while a is a local scratch array. To simplify the discussion, we assume that there is no false sharing, i.e., boundary columns start on page boundaries and their length is a multiple of the page size (Our methods work in the presence of false sharing. This simplification is for explanatory purposes only). Processes arrive at Barrier(2) at the end of each iteration, resulting in $2(n-1)$ messages with n processors. At the departure from the barrier (an *acquire*), pages containing elements of the boundary columns are invalidated since they have been modified on the neighboring processors. When a processor accesses a page in one of its

```
do k = 1,100
  do j = begin,end
    do i= 2,M-1
      a(i,j) = 0.25 *
      (b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
    enddo
  enddo
  call Barrier(1)
  do j = begin,end
    do i= 1,M
      b(i,j) = a(i,j)
    enddo
  enddo
  call Barrier(2)
enddo
```

Fig. 4. Pseudo-code for the TreadMarks Jacobi program: The variables *begin* and *end* are used to partition the work among the processors, with each processor working on a different partition of the shared array *b*.

```
do k = 1,100
  do j = begin,end
    do i= 2,M-1
      a(i,j) = 0.25 *
      (b(i-1,j)+b(i+1,j)+b(i,j-1)+b(i,j+1))
    enddo
  enddo
  call Barrier(1)
  call Validate(1,{b,2,DIRECT,
            {b[1,M:begin,end]}},WRITE_ALL,1);
  do j = begin,end
    do i= 1,M
      b(i,j) = a(i,j)
    enddo
  enddo
  call Push(b[1,M:begin(p)-1,end(p)+1],
            b[1,M:begin(p),end(p)])
enddo
```

Fig. 5. Pseudo-code for the transformed Jacobi program: A `Validate` has been inserted, and `Barrier(2)` has been replaced by `Push`. In the arguments to `Push`, the dependence of `begin` and `end` on the processor number `p` has been made explicit.

neighbor's boundary columns in the first half of the next iteration, it takes a page fault, which causes TreadMarks to fetch a diff from its neighbor. With m pages in a boundary column, the result is $4m(n-1)$ messages. In addition, there are another $2(n-1)$ messages at `Barrier(1)` that ends the first half of the iteration. Finally, there is consistency overhead for write detection during the second half of the iteration, including page faults, memory protection operations, and creating *twins* and *diffs*.

In a message passing version of Jacobi, whether hand-coded or compiler-generated, at the end of an iteration, each processor sends two messages, one to each of its neighbors, containing the boundary column to be used by that neighbor in the next iteration. It waits to receive the boundary columns from its neighbors, and proceeds with the next iteration. The result is only $2(n-1)$ messages per iteration for the message passing program.

Compiler analysis and transformation can virtually eliminate the extra overhead in the SDSM version of the program. Figure 5 shows the transformed program.

First, by examining the sections of data written by individual processors before `Barrier(2)` and read afterwards, the compiler recognizes that `Barrier(2)` can be replaced by a `Push`. The sections of data accessed are supplied as arguments to the `Push` run-time call (in reality, functions that will compute these per-processor sections are passed). In this case, the run-time will perform a point-to-point message exchange among neighboring processors after intersecting the sections of data read and written by the individual processors. The `Push` eliminates barrier overhead and pushes the data rather than requesting or pulling it.

Second, by examining the accesses during the second half of each iteration, the compiler can determine that between `Barrier(1)` and `Barrier(2)`, a processor writes *all* elements of the pages in its assigned section of the array, without reading the data. Hence, it inserts a `Validate` for that section with a `WRITE_ALL` argument, which causes the run-time *not* to make twins and diffs for these pages, eliminating consistency overhead.

The only extra overhead that now exists is `Barrier(1)`. This barrier cannot be eliminated due to the anti-dependence across it, and remains because shared memory semantics are assumed.

In this particular example, analysis is precise: the compiler can determine exactly what data is read or written as a function of the processor identifier. In such a case it is also possible for the compiler to directly generate a message passing program. As will be seen in Section III the performance of this strategy and ours are very similar. However, our methods can also be applied to applications for which the analysis cannot be made precise, or for which only some phases can be analyzed.

E.2 Moldyn

Moldyn is a molecular dynamics simulation. Its computational structure resembles the non-bonded force calculation in CHARMM [5], 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 6 illustrates the program structure of Moldyn, and the force computation subroutine.

Due to implementation limitations (no interprocedural analysis), the compiler inserts a Validate call at the beginning of ComputeForces. The compiler analyzes the access patterns for each statement in the subroutine. In this case, the access pattern consists of reads to `x`, the only shared array, through the `interaction_list` indirection array. The accesses to the indirection array are themselves regular and determinable at compile-time. Hence, the compiler can determine the section of the indirection array through which the shared array `x` is accessed. This information is conveyed through the `Validate` call.

The run-time system traverses the section of the indirection array supplied through the `Validate` call to determine the pages in `x` that will be accessed, or the `schedule`. This traversal is performed *only* if the indirection array has changed since the last time the `schedule` has been updated. Requests for the invalid pages in the `schedule` are then sent out, and the data is aggregated before being sent back to the requesting processor. This results in a reduced number of messages compared to the base system.

```
program moldyn

do step = 1, nsteps
  if (mod(step,UPDATE_INTERVAL) .eq. 0) then
    call build_interaction_list()
  endif
  ... ...
  call ComputeForces()
  ... ...
enddo

subroutine ComputeForces()

Validate(1, {x, 1, INDIRECT,
{interaction_list[1:2, 1:num_inter]}},READ,1)

do i = 1, num_inter
  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
```

Fig. 6. Transformed Moldyn program

III. Results

Our experimental environment is an 8-processor IBM SP/2 running AIX version 3.2.5. Each processor is a 66.7 MHz RS6000 thin node with 64 KBytes of data cache and 128 Mbytes of main memory. Interprocessor communication is accomplished over the IBM SP/2 high-performance two-level cross-bar switch, using IBM's MPL message passing layer. Unless indicated otherwise, all results are for 8-processor runs.

The minimum roundtrip time using send and receive for the smallest possible message is 365 μseconds, including an interrupt.[1] The time for a remote 4Kbyte page fetch is 1054 μseconds. In TreadMarks, the minimum time to acquire a free lock is 427 μseconds. The minimum time to perform an 8-processor barrier is 893 μseconds. Under AIX 3.2.5, the time for both page faults and memory protection operations is a linear function of the page number and the number of pages in use. For instance, the memory protection operation time can vary between 18 and 800 μseconds with 2000 pages in use.

[1]Although substantially faster round-trip times are possible if interrupts are disabled, interrupts are required to implement lock and page requests in TreadMarks. For XHPF and CHAOS, interrupts were disabled.

Application	Data set size	Time (secs)
Jacobi - 4Kx4K	4096x4096	288.3
Jacobi - 1Kx1K	1024x1024	17.7
3D-FFT - 6x6x6	$2^6 \times 2^6 \times 2^6$	9.5
3D-FFT - 5x6x5	$2^5 \times 2^6 \times 2^5$	2.3
Shallow - 1Kx1K	1024x1024	74.8
Shallow - 1Kx.5K	1024x512	36.9
IS - 23-19	$N = 2^{23}, B_{max} = 2^{19}$	91.2
IS - 20-15	$N = 2^{20}, B_{max} = 2^{15}$	3.9
Gauss - 2Kx2K	2048x2048	3344.8
Gauss - 1Kx1K	1024x1024	271.5
MGS - 2Kx2K	2048x2048	449.3
MGS - 1Kx1K	1024x1024	56.4
Tomcatv - 1.4Kx1.4K	1400x1400	25.9
Tomcatv - 1Kx1K	1024x1024	14.5
Grid - 2Kx2K	2000x2000	382.2
Grid - 1.5Kx1.5K	1500x1500	215.0
Moldyn - 20 iter	16384	267.2
Moldyn - 11 iter	16384	467.3
NBF - 64x1024	64x1024	78.3
NBF - 64x1000	64x1000	76.5

TABLE I

Applications, data set sizes, and uniprocessor execution times

We separate our results in terms of regular and irregular applications. Our aim is to compare performance against state-of-the-art compiler techniques currently available to optimize performance for these types of applications.

A. Overall Results for Regular Applications

We used eight Fortran programs: IS and 3D-FFT from the NAS benchmark suite [4], the Shallow benchmark from the National Center for Atmospheric Research, Tomcatv from the SPEC benchmark suite [9], Grid from Applied Parallel Research, Inc., and Jacobi, Gauss, and Modified Gramm-Schmidt (MGS), three locally developed benchmarks. For each application, we use two data set sizes to illustrate any effects from changing the computation to communication ratio, as well as due to false sharing. Table I describes the data set sizes and the corresponding uniprocessor execution times.[2] Uniprocessor execution times were obtained by removing all synchronization from the TreadMarks programs; these times were used as the basis for the speedup figures.

We present the performance of these applications in three different versions:

1. The base TreadMarks program executing with the base TreadMarks run-time system – `Tmk`.
2. The compiler-optimized TreadMarks program executing with the augmented TreadMarks run-time system –

[2]All measurements for Tomcatv and Grid were made on 120 MHz thin nodes.

Opt-Tmk.

3. A message passing version automatically generated by the Forge XHPF compiler [3] from Applied Parallel Research, Inc. (APR) – XHPF.

The results for the XHPF compiler are provided in order to compare performance against a commercial parallelizing compiler for data-parallel programs.

Figure 7 shows the speedups achieved for all applications using the three different environments. The numbers for the compiler-optimized TreadMarks version reflect the gains achieved by the most sophisticated level of analysis possible for each application. There are no entries for IS using XHPF in the figure. XHPF cannot parallelize IS because of an indirect access to the main array in the computation.

Compiler optimization achieves substantial execution time improvements in comparison to the base TreadMarks, ranging from 0% to 59%.[3] For programs for which base TreadMarks achieves relatively good speedups (Jacobi, Shallow, Gauss, Tomcatv, Grid, and MGS), the execution time improvements are moderate: 0% to 18%. For the two programs (IS and 3D-FFT) for which base TreadMarks performs poorly compared to XHPF, execution time improvements are quite large, ranging from 48% to 59%. These gains are mainly due to communication aggregation, and elimination of consistency overhead. The execution times achieved by the compiler-optimized shared memory programs are within 0-9% of XHPF (except for Tomcatv with the 1Kx1K dataset, where cache effects result in the XHPF version showing significant performance degradation).

The compiler-optimized version of Jacobi (from our example in Figure 5) shows a 10-16% improvement in execution time over the base TreadMarks and is within 8% of the execution times of the XHPF version. For the 4096x4096 data set, Jacobi derives most of its improvement from communication aggregation, because of a significant reduction in the number of messages (5-fold). For the 1024x1024 data set, communication aggregation does not improve execution time, because the boundary rows are exactly one page. Eliminating `Barrier(2)` through the use of a `Push` provides most of the benefit. With a smaller data set, the cost of the barrier becomes proportionally higher, and hence its elimination results in some improvement in running time (10%). Correspondingly, in comparison to XHPF, while the performance of the 4096x4096 data set is similar, there is a slight drop in performance for the 1024x1024 data set. This is because of the extra `Barrier(1)`, which was not eliminated.

Performance gains for 3D-FFT for the larger problem size come mainly from communication aggregation and twin/diff creation elimination. The gains from the smaller problem size, however, also come from the elimination of data communication due to false sharing by the use of the `Push` directive (an additional 11%). The `Push` directive only updates those sections of data specified as being read by the processor, thereby resulting in reduced data communication in the presence of false sharing.

IS has a migratory access pattern. The use of diffs in TreadMarks results in extra data communicated due to the *diff accumulation* [24] problem - that of multiple overlapping diffs being communicated due to multiple processors successively modifying the same data. With the compiler-based directives, this overhead can be eliminated. The performance gains of ∼50% in comparison to Tmk for Opt-Tmk come from the above optimization (reduced data communication) in addition to communication aggregation.

Shallow, Gauss, Tomcatv, and MGS benefit mainly from communication aggregation. There are also some additional gains from combining synchronization and data transfer when the amount of data transferred is small. The performance of all three versions of Grid is similar due to the high computation to communication ratio resulting in near perfect speedups in all cases.

B. Overall Results for Irregular Applications

In the case of the irregular applications, we compare the compiler-optimized TreadMarks programs (Opt-Tmk) with the hand-coded CHAOS (inspector-executor based [29]) programs (CHAOS), as well as the base TreadMarks programs (Tmk). Our intent in presenting the CHAOS performance numbers is to compare performance with state-of-the-art compiler technology for irregular applications. The compiler-optimized TreadMarks programs include optimizations for both regular and irregular access patterns. Figure 8 presents the speedups at 8 processors for two programs, Moldyn from CHARMM [5] and NBF from the GROMOS benchmark [12], both molecular dynamics simulation kernels. Table I presents the sequential execution time and data set sizes used. In the case of Moldyn, we vary the frequency with which the indirection array is recomputed. In the case of NBF, we vary the data set size to introduce false sharing.

For Moldyn (from which our example in Figure 6 is taken), our optimized system is 11% faster than base TreadMarks, a result of an almost 5-fold reduction in the number of messages due to communication aggregation. Our optimized system is also up to 23% faster than CHAOS, depending on the frequency with which the indirection array is updated. The cost of access pattern computation (the inspector), which in our case consists of traversing the indirection array, is lower than in the inspector-executor approach. In the inspector-executor approach, global communication of data schedules is required since the communication is not request-response in nature.

To separate the effects of inspector computation, for NBF, we do not include the time to execute the inspector in the measured computation. In this case, our optimized system is no worse than 14% slower than CHAOS, and is up to 38% faster than the base TreadMarks system. If we include the execution time of the inspector, our approach is faster than CHAOS by up to 20% for 10 iterations of the program loop. Changing the data set from 64x1024 to 64x1000 introduces false sharing, resulting in the two

[3]Percentage improvements are calculated by the formula (*base* − *opt*) ÷ *base*.

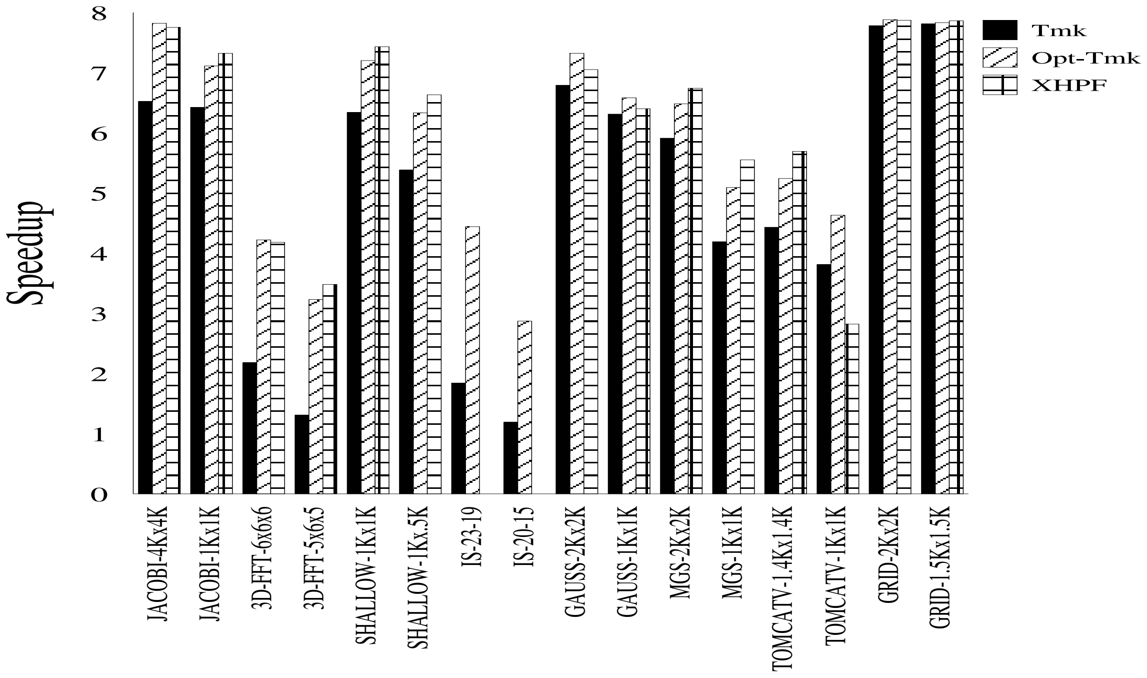

Fig. 7. Speedup (at 8 processors) for TreadMarks, Compiler-Optimized Version of TreadMarks, and XHPF. The IS bar is missing for XHPF because it cannot parallelize IS.

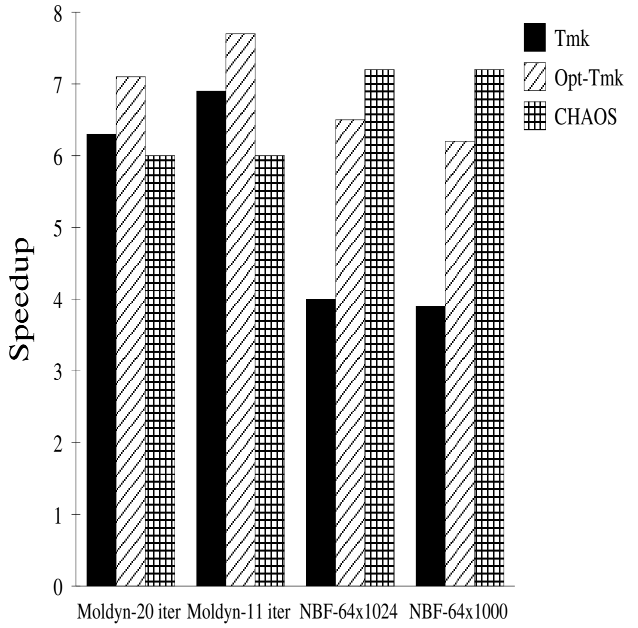

Fig. 8. Speedup (at 8 processors) for TreadMarks, Compiler Optimized Version of TreadMarks, and CHAOS.

TreadMarks versions sending more data than CHAOS.

Our compile-time optimizations successfully reduce the number of messages used during program execution, making performance comparable to a system such as CHAOS. The advantage of our approach increases as the frequency of changes to the indirection array increases. Its disadvantage is the potential for false sharing overhead when the data set is small or has poor spatial locality.

IV. Applicability to Other Platforms

While the experimental results presented here are specific to the TreadMarks SDSM system, the techniques described generalize to other SDSM systems such as Cashmere [30], home-based lazy release consistency (HLRC) [34], or Shasta [28]. In these systems, each coherence unit has a home where modifications are collected or where directory information is maintained. While careful placement of the home can result in a prefetching effect, such placement using purely run-time information does not capture either phase changes or complex access patterns, and can result in additional overhead. The compiler-provided access information can be used to optimize the migration/placement of the home. Write-first accesses, something the run-time has no knowledge of, can avoid data communication merely by changing the current home. The benefits of communication aggregation and consistency overhead elimination continue to apply in such systems, although the run-time mechanisms will differ. For virtual memory-based systems such as Cashmere and HLRC, memory protection operations are eliminated. Also, the `Push` interface can avoid extra data communication as a result of false sharing. For variable-grain instrumentation-based systems such as Shasta, the instrumentation overhead can be further reduced.

Our experimental results have also been presented in the context of a fairly high-latency communication subsystem. If a low-latency network were to be used, the benefits of aggregation would shift from being purely due to a reduction in the number of messages, to being able to overlap communication with computation.

Our compiler framework was implemented for explicitly parallel programs. However, the general principle is also applicable to automatic parallelization with the SDSM system as the target. The access pattern information can be folded into the shared memory parallelization directives. These directives identify all data races, and hence perform a similar function to the synchronization in the explicitly parallel programs in terms of identifying the appropriate points at which to supply the access pattern information. This information can be utilized by the run-time, not only to optimize communication, but also to balance load [15].

Several recent proposals for hardware shared memory machines include a message passing subsystem designed in part to allow applications to take advantage of bulk data transfer [20], [21]. Woo et al. [33] evaluate one such design in the context of the Flash system. While Woo et al. focus on establishing the magnitude of the performance benefits of bulk data transfer with hardware-based shared memory, we have explored in addition ways for the compiler to automate the use of the bulk data transfer facility in a software shared memory environment. The same access pattern information can be used in a hardware shared memory environment to exploit the bulk transfer features. The information can also be used for optimal page placement and re-mapping in machines such as the Origin-2000.

V. Related Work

Mowry et al. [25] examine the effect of combining prefetching and multithreading in a software DSM system. Their prefetching strategy involves fetching data in advance of synchronization operations. Our strategy involves leveraging the program synchronization in order to reduce redundant messages, as well as eliminating consistency overhead where possible.

Jeremiassen et al. [16] present a static algorithm for computing per-process memory references to shared data in coarse-grained parallel programs. We use a similar analysis in terms of processor identifiers in order to replace a barrier with a `Push`.

Mukherjee et al. [26] 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 [27]. They conclude that TSM is not competitive with CHAOS, while XSM achieves performance comparable to CHAOS after introducing several special-purpose protocols. In our work, we use a fairly straight-forward compiler to optimize the shared memory programs, rather than relying on hand-coded special-purpose protocols.

Keleher and Tseng [18] describe a run-time interface and compile-time system that couples the compiler and the run-time in a manner similar to our system. Their interface and implementation are, however, more run-time intensive. Chandra and Larus [7] also describe a combined compiler and run-time system that is similar in spirit to our system, but in the context of fine-grained software shared memory.

VI. Conclusion

We have described an integrated compile-time/run-time approach for executing regular and irregular computations on distributed memory machines. This approach is based on a modified software distributed shared memory layer, and fairly simple compile-time support. Our compiler computes data access summaries using regular section analysis and feeds that information to the TreadMarks run-time SDSM system. Improvements in execution time range from 0 to 59% on an 8-processor IBM SP/2 in comparison to the base run-time system for the applications analyzed. The combination of static prediction of shared memory accesses by the compiler with dynamic detection of accesses by the run-time allows the combined system to approach the performance of compiler-generated message passing (within 9% of XHPF for regular programs, and up to 23% better than CHAOS for irregular programs). It does so without incurring the programming difficulties of message passing or the limitations on automatic parallelization of data-parallel programs for message passing targets. A combined compile-time run-time system of this nature retains the ease of programming of shared memory, while exploiting the message passing capabilities of the underlying hardware.

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, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Applied Parallel Research. *FORGE High Performance Fortran User's Guide*, version 2.0 edition.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. Technical Report 103863, NASA, July 1993.

[5] 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.

[6] 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.

[7] S. Chandra and J. R. Larus. Optimizing communication in hpf programs for fine-grain distributed shared memory. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, June 1997.

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

[9] K. M. Dixit. The spec benchmarks. *Parallel Computing*, pages 1195–1209, 1991.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] S. Ioannidis and S. Dwarkadas. Compiler and run-time support for adaptive load balancing in software distributed shared memory systems. In *Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, May 1998.

[16] T.E. Jeremiassen and S. Eggers. Computing per-process summary side-effect information. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Fifth Workshop on Languages and Compilers for Parallelism*, pages 175–191, August 1992.

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

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

[19] 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.

[20] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B. Lim. Integrating message-passing and shared-memory: Early experience. In *Proceedings of the 1993 Conference on the Principles and Practice of Parallel Programming*, May 1993.

[21] J. Kuskin and D. Ofelt et al. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

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

[23] H. Lu, A.L. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the 6th Symposium on the Principles and Practice of Parallel Programming*, pages 48–56, June 1997.

[24] 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.

[25] T.C. Mowry, C.Q.C. Chan, and A.K.W. Lo. Comparative evaluation of latency tolerance techniques for software distributed shared memory. In *Proceedings of the Fourth High Performance Computer Architecture Symposium*, February 1998.

[26] 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 ACM Symposium on the Principles and Practice of Parallel Programming*, July 1995.

[27] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[28] D.J. Scales, K. Gharachorloo, and C.A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the 7th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 174–185, October 1996.

[29] S. D. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and compile-time support for adaptive irregular problems. In *SuperComputing*, 1994.

[30] R. Stets, S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M.L. Scott. Cashmere-2l: Software coherent shared memory on a clustered remote-write network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 170–183, October 1997.

[31] 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.

[32] 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.

[33] S.C. Woo, J.P. Singh, and J.L. Hennessy. The performance advantages of integrating block data transfer in cache-coherent multiprocessors. In *Proceedings of the 6th Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 219–231, October 1994.

[34] Y. Zhou, L. Iftode, and J.P. Singh. Performance evaluation of two home-based lazy release consistency protocols for shared virtual memory systems. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, October 1996.