Compile-time Synchronization Optimizations for Software DSMs

Hwansoo Han, Chau-Wen Tseng

Department of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Software distributed-shared-memory (DSM) systems provide a desirable target for parallelizing compilers due to their flexibility. However, studies show synchronization and load imbalance are significant sources of overhead. In this paper, we investigate the impact of compilation techniques for eliminating synchronization overhead in software DSMs, developing new algorithms to handle situations found in practice. We evaluate the contributions of synchronization elimination algorithms based on 1) dependence analysis, 2) communication analysis, 3) exploiting coherence protocols in software DSMs, and 4) aggressive expansion of parallel SPMD regions. We also found suppressing expensive parallelism to be useful for one application. Experiments indicate these techniques eliminate almost all parallel task invocations, and reduce the number of barriers executed by 66% on average. On a 16 processor IBM SP-2, speedups are improved on average by 35%, and are tripled for some applications.

1 Introduction

Experience has shown that distributed-memory parallel architectures (e.g., IBM SP-2, Cray T3D) come closest to achieving peak performance when programmed using a message-passing paradigm. Expert users are willing to expend time and effort to write message-passing programs for important applications. However, easier programming methods must be found for general users to make parallel computing truly successful. Compilers for languages such as High Performance Fortran (HPF) [15] provide a partial solution because they allow users to avoid writing explicit message-passing code, but HPF compilers currently only support a limited class of data-parallel applications.

One method for increasing the programmability of message-passing machines is to combine powerful shared-memory parallelizing compilers with software distributed-shared-memory (DSM) systems that provide a coherent shared address space in software. Scientists and engineers can write standard Fortran programs, rewriting a few computation-intensive procedures and adding parallelism directives where necessary. The resulting programs are portable since they can be run on large-scale parallel machines as well as low-end, but more pervasive multiprocessor workstations.

This research was supported by NSF CAREER Development Award #ASC9625531 in New Technologies. The IBM SP-2 and DEC Alpha Cluster were provided by NSF CISE Institutional Infrastructure Award #CDA9401151 and grants from IBM and DEC.

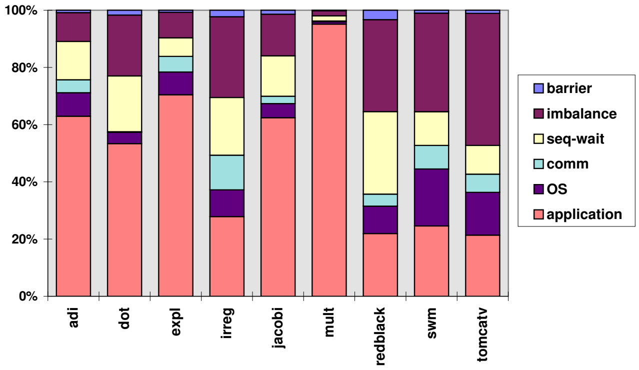

Figure 1. Breakdown of Total Execution Time (16 Processor SP-2)

Shared-memory parallelizing compilers are easy to use, flexible, and can accept a wide range of applications. Results from recent studies [3, 14] indicate they can approach the performance of current message-passing compilers or explicitly-parallel message-passing programs on distributed-memory machines. However, load imbalance and synchronization overhead were identified as sources of inefficiency when compared with message-passing programs.

Figure 1 categorizes execution time for several compiler-parallelized applications on a software DSM. Execution times is split into barrier overhead (time spent executing barrier code), load imbalance (idle time waiting for all processors to complete current parallel task), sequential wait (idle time waiting for next parallel task), communication cost, OS overhead, and application processing time. Measurements demonstrate idle time (caused by synchronization) can comprise a large portion of overall execution time across a range of sample applications. While not much time is spent explicitly executing barriers in these programs, the idle time induced by barriers comprises 37% of execution time on average. One possible reason is that while the compiler-generated code is balanced computationally, but the underlying DSM and OS effectively add random (and unequal) delays. It is clear from these measurements that reducing idle time caused by synchronization overhead is important for achieving good performance.

In this paper we investigate a number of compiler techniques for reducing synchronization overhead and load imbalance for compiler-parallelized applications on software DSMs. Our techniques are evaluated in a prototype system [14] using the CVM [12] software distributed-shared-memory (DSM) as a compilation target for the SUIF [8] shared-memory compiler. This paper makes the following contributions:

- empirical evaluation of compiler synchronization optimizations for software DSMs
- evaluating impact of communication analysis
- exploiting lazy release consistency to eliminate barriers guarding only anti-dependences
- reducing parallelism overhead by aggressively expanding SPMD regions to enclose time-step loops

We begin by introducing our compiler/software DSM framework, then describe each optimization technique. We present our prototype system, followed by experimental results. We conclude with a discussion of related work.

2 Background

2.1 SUIF Shared-Memory Compiler

SUIF is a optimizing and parallelizing compiler developed at Stanford [8]. It has been successful in finding parallelism in many standard scientific applications. Like most shared-memory parallelizing compilers, SUIF employs a *fork-join* programming model, where a single master thread executes the sequential portions of the program, assigning (forking) computation to additional worker threads when a parallel loop or task is encountered. After completing its portion of the parallel loop, the master waits for all workers to complete (join) before continuing execution. During the parallel computation, the master thread participates by performing a share of the computation as a worker. After each parallel computation worker threads spin or go to sleep, waiting for additional work from the master thread.

2.2 CVM Software DSM

CVM is a software DSM that supports coherent shared memory for multiple protocols and consistency models [12]. It is written entirely as a user-level library and runs on most UNIX-like systems. Its primary coherence protocol implements a multiple-writer version of *lazy release consistency*, which allows processors to delay making modifications to shared data visible to other processors until special synchronization [13]. Experiments show that lazy-release-consistency protocols generally cause less communication than release consistency [6]. Consistency information in CVM is piggybacked on synchronization messages. Multiple updates are also aggregated in a single message where possible.

2.3 SUIF/CVM Interface

SUIF was retargeted to generate code for CVM by providing a runtime interface based on CVM thread creation and synchronization primitives. Performance was improved by adding customized support for reductions, as well as a *flush update* protocol that at barriers automatically sends updates to processors possessing copies of recently modified shared data [14]. Compiler analysis needed to use the flush update protocol is much simpler than communication analysis needed in HPF compilers. The identities of the sending/receiving processors do not need to be computed at compile time, and the compiler does not need to be 100% accurate since the only effect is on efficiency, not correctness. Instead, the compiler only needs to locate data that will likely be communicated in a stable pattern, then insert calls to DSM routines to apply the flush protocol for those pages at the appropriate time.

2.4 Compile-time Barrier Elimination

The fork-join model used by shared-memory compilers is flexible and can easily handle sequential portions of the computation; however, it imposes two synchronization events per parallel loop as shown in Figure 2(A). First, a *broadcast barrier* is invoked before the parallel loop body to wake up available worker threads and provide workers with the address of the computation to be performed and parameters if needed. A *barrier* is then invoked after the loop body to ensure all worker threads have completed before the master can continue.

Measurements show synchronization overhead can significantly degrade performance. Barriers are expensive for many reasons. First, executing a barrier has some run-time overhead that typically grows quickly as the number of processors increases. Second, executing a barrier requires all processors to idle while waiting for the slowest processor; this effect results in poor processor utilization when processor execution times vary. Eliminating the barrier allows perturbations in task execution time to even out, taking advantage of the loosely coupled nature of multiprocessors. Barrier overhead goes up as the number of processors increases, since the interval between barriers decreases as computation is partitioned across more processors. Barriers are particularly expensive in software DSMs, since synchronization triggers communication of coherence data for lazy-release-consistency systems. The added cost of communication and OS support also increase the potential for load imbalance in software DSMs.

In previous work, we developed compiler algorithms for barrier elimination [27]. We first generate code for parallel loops using the *single-program, multiple-data* (SPMD) programming model found in message-passing programs, where all threads execute the entire program. Sequential computation is either replicated or explicitly guarded to limit execution to a single thread, while parallel computation is partitioned and executed across processors. By placing multiple loops in the same SPMD region, we make barriers explicit and provide opportunities for barrier elimination or replacement. The compiler attempts to make the parallel SPMD region as large as possible by merging adjacent regions, as well as outer loops whenever the entire loop body is in an SPMD region.

Compile-time barrier elimination is made possible by the observation that synchronization between a pair of processors is only necessary if they communicate shared data. If *data dependence analysis* can show two adjacent loop nests access disjoint sets of data, then the barrier separating them may be eliminated, as in Figure 2(B). If two loop nests accesses the same data, but *communication analysis* proves no remote data is accessed based on data and computation decomposition information, the barrier may also be eliminated, as between loops I and J in Figure 2(C). In addition, if communication analysis identifies simple interprocessor sharing patterns, the barrier may be replaced with less expensive forms of synchronization. In particular, if data is only shared between neighboring processors, the barrier may be replaced by nearest-neighbor synchronization, as between loops J and K in Figure 2(C). Communication analysis is enabled if the computation partition is known at compile time.

3 Reducing Synchronization at Compile-Time

In Section 4, we will examine the impact of barrier elimination algorithms on the performance of SUIF-parallelized programs for CVM. Here we discuss additional compiler techniques we found useful for reducing synchronization overhead in software DSMs.

```
{ Default Model }
DOALL I = 1,N
  ...
DOALL J = 1,N
  ...

⇓

broadcast
DO I = LB1,UB1
  ...
barrier
broadcast
DO J = LB2,UB2
  ...
barrier
```

(A)

```
{ Dependence
  Analysis }
DOALL I = 1,N
  A(I) =
DOALL J = N+1,2*N
  B(J) = A(J)

⇓

broadcast
DO I = LB1,UB1
  A(I) =
DO J = LB2,UB2
  B(J) = A(J)
barrier
```

(B)

```
{ Communication
  Analysis }
DOALL I = 1,N
  A(I) =
DOALL J = 1,N
  B(J) = A(J)
DOALL K = 1,N
  C(K) = B(K-1)

⇓

broadcast
DO I = LB1,UB1
  A(I) =
DO J = LB1,UB1
  B(J) = A(J)
sync_with_neighbors
DO K = LB1,UB1
  C(K) = B(K-1)
barrier
```

(C)

```
{ Lazy Release
  Consistency }
DOALL I = 1,N
  A(I) = B(I-1)
DOALL J = 1,N
  B(J) = C(J)

⇓

broadcast
DO I = LB1,UB1
  A(I) = B(I-1)
sync_first_time(T)
DO J = LB1,UB1
  B(J) = C(J)
barrier
```

(D)

```
{ Default Model }
DO TIME
  DOALL I = 1,N
    ...
  S = 0
  DOALL J = 1,N
    S = S + A(I)

⇓

DO TIME
  broadcast
  DO I = LB1,UB1
    ...
  barrier
  S = 0
  broadcast
  init_sum(S')
  DO J = LB2,UB2
    S' = S' + A(I)
  reduce_sum(S,S')
  barrier
```

(E)

```
{ Reduction
  Initializations }
DO TIME
  S = 0
  DOALL I = 1,N
    S = S + A(I)
  ...

⇓

broadcast
DO TIME
  init_sum(S')
  DO I = LB1,UB1
    S' = S' + A(I)
  reduce_sum_init(S,S',0)
  ...
  barrier
```

(F)

```
{ Control Flow }
DO TIME
  DOALL I = 1,N
    ...
  IF (C) THEN
    DOALL J = 1,N
      ...
  ELSE
    DOALL K = 1,N
      ...

⇓

broadcast
DO TIME
  DO I = LB1,UB1
    ...
  IF (C) THEN
    DO J = LB1,UB1
      ...
  ELSE
    DO K = LB1,UB1
      ...
  barrier
```

(G)

```
{ Procedure Calls }
DO TIME
  CALL P()
  CALL Q()

PROCEDURE P()
  DOALL I = 1,N
    ...
PROCEDURE Q()
  DOALL J = 1,N
    ...

⇓

broadcast
DO TIME
  CALL P()
  CALL Q()
  barrier

PROCEDURE P()
  DO I = LB1,UB1
    ...
PROCEDURE Q()
  DO J = LB1,UB1
    ...
```

(H)

Figure 2. Optimization Examples

3.1 Communication Analysis

Our implementation of communication analysis in SUIF differs somewhat from previous work [27]. Instead of relying on actual data/computation partitions provided in a language such as HPF, we rely on knowledge of the simple computation partitioning strategy applied by the SUIF compiler to efficiently detect most cases of interprocessor communication. Consistent iteration partitioning of loops with identical loop bounds will always assign the same loop iterations to each processor. If any subscript pair containing the parallel loop index is identical, each processor will access only local data. If the subscripts differ by a sufficiently small constant, then each processor will access remote data only from neighboring processors. We find local communication analysis is sufficient to detect interprocessor communication in our applications [9].

In addition, we discover taking advantage of nearest-neighbor synchronization requires implementing a customized synchronization routine which sends a single message to each neighboring processor upon arrival, and continues as soon as messages are received from all neighboring processors [9]. The system reduces contention at the master processor, allows load imbalance to smooth out between parallel regions, and improves the ability of the software DSM to piggyback coherence data on synchronization messages.

3.2 Exploiting Lazy Release Consistency

We discovered an enhancement to the barrier elimination algorithm made possible by the underlying semantics of lazy-release-consistency software DSMs. In a traditional shared-memory model, synchronization is needed between two loop nests if two processors access shared data, with at least one of the processors performing a write to the shared data. However, in a lazy-release-consistency software DSM, if the shared reference is a read in the first loop and a write in the second loop, no synchronization is needed because writes from the second loop will not become visible to the read in the first loop until synchronization is encountered. In other words, anti-dependences (write-after-read) do not need to be synchronized.

The one exception is when the processor performing a read does not yet possess a copy of the shared data, since it may retrieve a copy of the data with the new values. For scientific computations where iterative computations are the rule, this is rarely the case. In fact, for regular computations, the compiler can prove this situation may only occur the first time the barrier is encountered. We can thus improve performance by replacing a barrier protecting only anti-dependences with a special execute-once barrier which synchronizes processors only the first time it is invoked at each point in the program. For irregular adaptive applications, the compiler will need execute-after-change barriers which are executed whenever access pattern changes may cause a processor to access new shared data for the first time.

To see how the compiler can exploit delayed updates in a lazy-release-consistency system, consider the example shown in Figure 2(D). The first loop nest reads nonlocal values of B which are defined in the second loop nest. The cross-processor dependence caused by B is thus a loop-independent anti-dependence. Normally, synchronization is needed to ensure the old values of B are read before the new values of B are written. However, with lazy release consistency the software DSM guarantees that new values of B on another processor will not be made visible until the two processors synchronize. Since there are no other cross-processor dependences between the two loop nests, synchronization between them is not required, except possibly the first time the loops are executed. The compiler can thus replace the barrier with a special execute-once barrier.

Since anti-dependences may be ignored, the algorithm for inserting barrier synchronization becomes similar to the algorithm for *message vectorization* [11]. The level of the deepest true/flow cross-processor dependence becomes the point where synchronization must be inserted to prevent data races. Synchronization at lower loop levels can be replaced by execute-once barriers.

3.3 Aggressive Expansion of Parallel SPMD Regions

Previous techniques were implemented in the SUIF compiler and applied to programs. Barriers were eliminated, with fair improvements in performance. However, measurements indicate that significant idle time remains, leading us to investigate additional techniques for reducing synchronization. Several obstacles were discovered which prevent the compiler from merging parallel loops into the same parallel SPMD region, a prerequisite to analyzing communication and eliminating barriers. In particular, a number of program artifacts prevent the compiler from expanding parallel SPMD regions to enclose the outer time-step loop. In this section, we present techniques for overcoming these problems.

Reduction Initializations Most computations in time-step loops are parallel, and can be merged into the same parallel SPMD region. The most common obstacle encountered in the benchmarks was initialization statements for *reductions*. Reductions are commutative operations such as SUM or MAX which may be computed in parallel using initialized private variables, then accumulated into a single global variable using runtime routines. Typically the global variable is first initialized to a base value, such as zero for SUM and MINREAL for MAX. An example of how reductions are parallelized by the compiler is shown in Figure 2(E), where a SUM reduction over the elements of A is parallelized by calculating partial sums using the private variable S', then accumulating each processor's contribution into the global variable S using *reduce_sum*().

Unfortunately, initializations to global variables cannot be included in parallel SPMD regions without additional synchronization, and can prevent the compiler from enlarging SPMD regions. For the example code in Figure 2(E), the initial assignment to S prevents the compiler from merging the SPMD regions for the parallel I and J loops, requiring parallel tasks to be invoked twice for each time step. Once reduction initializations are discovered to be a problem, the compiler can be easily extended to search for reduction initializations. If initializations are found, they may be eliminated by calling versions of runtime routines which automatically initialize global variables used to accumulate reduction results. After eliminating reduction initializations, the compiler can create larger parallel SPMD regions, including the time-step loop, as in Figure 2(F). Since the time-step loop is sequential, a barrier must be inserted into the body of the loop to synchronize between iterations.

Control Flow In addition to procedure calls, we also found control flow to inhibit the ability of the compiler to enlarge parallel SPMD regions. Figure 2(G) presents an example where parallel loops occur in both the *then* and *else* branches of an IF statement. The compiler can be extended to merge the entire IF statement into the parallel SPMD region, inserting synchronization where necessary if the conditional expression for the IF statement accesses global values defined within the loop. Once the IF statement has been put in a parallel SPMD region, the compiler is able to enlarge the SPMD region to enclose the entire time-step loop.

Procedure Calls Another obstacle preventing the compiler from expanding parallel SPMD regions are procedure calls, as shown in Figure 2(H). If the compiler can perform interprocedural analysis (or inlining) to determine that a procedure call contains only parallel (or guarded sequential) code, it can modify the procedure body and safely expand parallel SPMD regions to enclose procedure calls. Once the SPMD region encloses the procedure calls, the compiler can merge the entire time-step loop into the SPMD region.

For our benchmark programs, these three techniques for aggressively enlarging parallel SPMD code were quite successful. All programs were able to include their time-step loops within a single large parallel SPMD region, significantly reducing parallelism startup and barrier synchronization overhead.

3.4 Suppressing Parallelism

Finally, we found one additional technique to be useful. Most loop nests in the benchmark programs were parallelized in a consistent manner. As a result, each processor performs its portion of the computation using the same slice of data from each global array. Computations can thus be parallelized efficiently, with little interprocessor communication.

We found in our experiments that occasionally parallel loops (typically dealing with initializations or boundary conditions) are encountered which operate on different slices of data than the main computation loops. Parallelizing these loops incurs significant communication between processors, and reduces the efficiency of the flush update coherence protocol of the underlying software DSM. The compiler needs to perform communication analysis to determine when parallelizing a loop results in excessive interprocessor communication. When such loops are detected, the compiler can improve actual overall performance by executing such loops sequentially on the master processor. Not much performance is lost since such loops are usually singly-nested loops not containing much computation.

4 Experimental Results

This section presents our experimental results. We discuss our experimental environment, present our overall results, and discuss the effect of our compiler optimizations.

4.1 Applications

We evaluated the performance of our compiler/software DSM interface with programs shown in Table 1. The "Granularity" column refers to the average length in seconds of a parallelized loop in the original program. Except where indicated, numbers below refer to the larger data set for each application. `adi`, `expl`, and `redblack` are dense stencil kernels typically found in iterative PDE solvers. `dot` calculates the dot product of two vectors. `jacobi` is a stencil kernel combined with a convergence test that checks the residual value using a MAX reduction. `irreg` performs a similar computation, but over a randomly generated irregular mesh. `mult` multiples two matrices. `swm` and `tomcatv` are programs from the SPEC benchmark suite containing a mixture of stencils and reductions. We use the version of `tomcatv` from APR in which arrays are transposed to improve data locality for parallel execution.

All applications were originally written in Fortran, and contain an initialization section followed by the main computation enclosed in an sequential time-step loop. The main computation is thus repeated on each iteration of the time-step loop. Statistics and timings are collected after the initialization section and the first few iterations of the time-step loop, in order to more closely match steady-state execution.

4.2 Experimental Environment

We evaluated our optimizations on an IBM SP-2 with 66MHz RS/6000 Power2 processors operating AIX 4.1. Nodes are connected by a 120 Mbit/sec bi-directional Omega switch capable of a sustained bandwidth of approximately 40 Mbytes per second. Simple RPCs on the SP-2 require 160 μsecs. Nonlocal accesses causing page misses require 2-3 messages and take from 1 to 1.3 μsecs to fetch remote data.

In our experiments, CVM [12] applications written in Fortran 77 were automatically parallelized by the Stanford SUIF parallelizing compiler version 1.1.2 [8], with close to 100% of the computation in parallel regions. A simple chunk scheduling policy assigns contiguous iterations of equal or near-equal size to each processor, resulting in a consistent computation partition that encourages good locality. The resulting C output code was compiled by `g++` version 2.7.2 with the -O2 flag, then linked with the SUIF run-time system and the CVM libraries to produce executable code on the IBM SP-2. Customized support for reductions and the *flush update* protocol were used to improve overall performance [14].

All synchronization optimizations were implemented in the SUIF compiler, except for expanding SPMD regions across procedure boundaries (due to lack of interprocedural analysis). For our experiments, procedures in time-step loops were inlined by hand. We also manually transformed the time-step loop in `tomcatv`

Name	Description	Problem Sizes		Granularity (secs)	
		Small	Large	Small	Large
adi	ADI Fragment (Livermore 8)	32K	64K	0.31	0.63
dot	Dot Product (Livermore 3)	256K	512K	0.10	0.28
expl	Explicit Hydrodynamics (Livermore 18)	256^2	512^2	0.06	0.34
irreg	Irregular Solve Over Mesh	500K	1000K	0.06	0.12
jacobi	Jacobi Iteration w/Convergence Test	512^2	1024^2	0.06	0.91
mult	Matrix Multiply (Livermore 21)	300^2	400^2	1.83	4.33
redblack	Red-Black Successive-Over-Relax.	512^2	1024^2	0.01	0.14
swm	Shallow Water Model (SPEC)	512^2	750^2	0.10	0.20
tomcatv	Vector Mesh Generation (SPEC)	256^2	512^2	0.04	0.15

Table 1. Applications

Program	Barrier Elimination/Replacement				Aggressive SPMD Expansion				Suppress parallel loops
	dep anal.	comm anal.	lazy RC	nearest neighbor	time step	reduc init	control flow	proc call	
adi					√				
dot					√	√			
expl		√	√	√	√				
irreg					√	√		√	
jacobi		√	√	√	√	√			
mult					√				
redblack	√	√		√	√				
swm	√				√		√	√	√
tomcatv		√			√	√	√		

Table 2. Applicability of Optimizations

Program	Doalls executed			Barriers executed by program							
	orig	% eliminated		orig	% eliminated					% replaced	
	num	merge	aggr	num	merge	dep	comm	lazy	aggr	comm	lazy
adi	20	–	95	40	–	–	–	–	45	–	–
dot	100	–	99	200	–	–	–	–	49	–	–
expl	60	67	98	120	33	33	33	50	82	33	17
irreg	80	75	99	160	50	50	50	50	61	–	–
jacobi	40	50	98	80	25	25	25	50	73	25	–
mult	10	–	90	20	–	–	–	–	40	–	–
redblack	80	75	99	160	38	63	63	63	86	13	13
swm	265	33	99	530	17	33	33	33	66	–	–
tomcatv	140	71	99	280	36	36	71	71	92	–	–
Average	88	41	97	177	22	27	31	35	66	8	3

Table 3. Dynamic Measurement of Synchronization Optimizations

since it was formed with a backwards goto (which SUIF did not recognize).

4.3 Effectiveness of Optimizations

First, we examine the optimizations applied to each benchmark code, shown in Table 2. The first four columns indicate the type of barrier elimination algorithm successfully applied: dependence analysis, communication analysis, exploiting lazy release consistency, and barrier replacement with nearest-neighbor synchronization. The next four columns show whether aggressive SPMD expansion was able to encompass the time-step loop, and whether the compiler was required to handle reduction initializations, control flow, or procedure calls. Finally, the last column indicates whether parallelism for loops were suppressed for efficiency. As we can see, compile-time barrier elimination was successful for many programs, and aggressive SPMD expansion was always able to move the time-step loop inside the parallel SPMD region using the techniques described in the paper.

We now examine the effectiveness of compiler algorithms in eliminating synchronization. Table 3 displays the number of parallel loops (doalls) and barriers executed dynamically by each program at run time, and the percentage eliminated by different levels of optimization. The first three columns for "doalls" indicate the number of parallel loops executed in the original program, the percentage eliminated by merging adjacent doalls into the same parallel SPMD region, and the percentage eliminated by aggressively expanding SPMD regions to include the outer time-step loop. The remaining columns show the number of barriers executed by the original program, followed by the percentage eliminated or replaced by nearest-neighbor synchronization for different levels of optimization.

The compiler optimization levels are as follows: "merge" measures the effect of merging adjacent parallel loops into a single parallel region, "dep" eliminates barriers in parallel regions based on data dependences, "comm" performs communication analysis using local subscript analysis to eliminate barriers, "lazy" eliminates barriers guarding only anti-dependences, "aggr" aggressively expands parallel SPMD regions to include the outer time-step loop. Communication analysis may also replace barriers with nearest-neighbor synchronization. Optimizations are cumulative. Dashes are used to mark programs when no optimizations have been applied.

We see from Table 3 that the compiler is effective at reducing the number of parallel tasks and barriers actually executed by the application at run time. We find the compiler is quite successful in discovering adjacent parallel loops which may be merged into a single parallel region, eliminating on average 88% of parallel invocations and 22% of barriers executed. Dependence analysis alone is only able to eliminate barriers in two programs, `redblack` and `swm`, but the improvement in `redblack` is significant. Communication analysis can eliminate barriers in `tomcatv` and eliminate/replace barriers in `expl`, `jacobi`, and `redblack`. Detecting barriers guarding only anti-dependences, the compiler can eliminate more barriers outright in `expl` and `jacobi`. The number of replaced barriers goes down in `expl` and `jacobi`, since the compiler can prove some nearest-neighbor barriers guard only

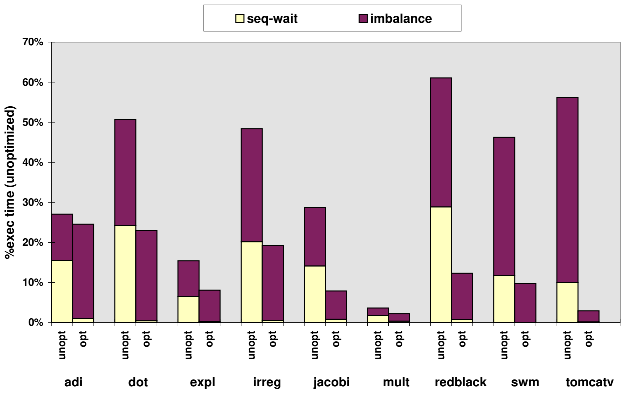

Figure 3. Impact of Optimizations on Idle Time (16 Proc SP-2)

Program		orig	dep anal.	comm anal.	lazy RC	aggr SPMD	suppress parallel
		Speedup					
adi	sm	7.8	7.8	7.8	7.8	8.4	8.4
	lg	10.4	10.4	10.4	10.4	10.7	10.7
dot	sm	4.8	4.8	4.8	4.8	7.6	7.6
	lg	7.2	7.2	7.2	7.2	10.4	10.4
expl	sm	5.5	6.5	6.7	7.1	7.9	7.9
	lg	12.2	12.8	13.2	13.3	13.7	13.7
irreg	sm	3.0	4.0	4.0	4.0	4.6	4.6
	lg	4.7	5.9	5.9	5.9	7.0	7.0
jacobi	sm	5.3	6.5	6.9	7.8	9.9	9.9
	lg	11.5	12.7	13.0	13.3	15.2	15.2
mult	sm	21.5	21.5	21.5	21.5	22.1	22.1
	lg	22.8	22.8	22.8	22.8	23.2	23.2
redblack	sm	1.2	2.6	2.8	2.8	3.7	3.7
	lg	3.7	6.4	6.7	6.7	7.9	7.9
swm	sm	1.2	1.3	1.3	1.3	1.5	3.3
	lg	1.7	1.9	1.9	1.9	2.4	4.6
tomcatv	sm	1.2	1.3	1.6	1.6	5.0	5.0
	lg	2.6	2.7	3.0	3.0	9.6	9.6
Average	sm	5.4	5.9	6.0	6.2	7.5	7.7
	lg	8.2	8.8	9.0	9.0	10.8	11.0

Table 4. Impact of Optimizations on Speedup (16 Processors)

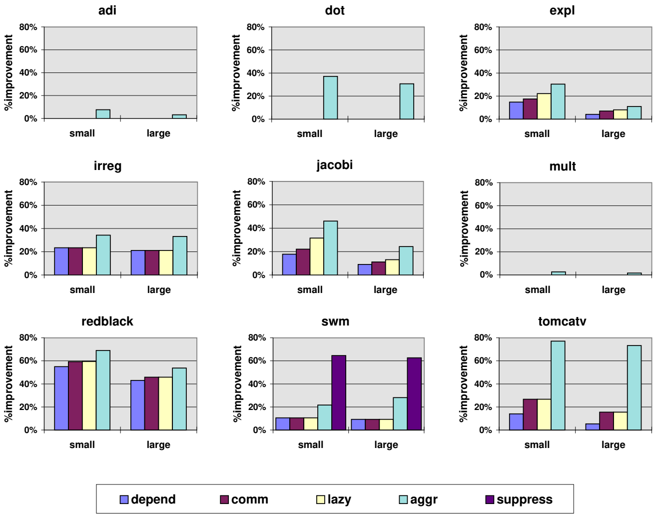

Figure 4. Impact of Optimizations on Speedup (16 Proc SP-2)

anti-dependences. Aggressively enlarging parallel SPMD regions until the time-step loop is enclosed eliminates 97% of all parallel loop invocations, and eliminates an additional 30% of barriers executed. Applying all optimizations, on average 66% of all barrier executions are eliminated, with 3% of barriers replaced by nearest-neighbor synchronization.

4.4 Impact of Compiler Optimizations on Idle Time

In order to evaluate synchronization optimizations, we instrumented CVM to directly measure sequential wait (idle time waiting for next parallel task), and load imbalance (idle time waiting for completion of current parallel task). As previously noted, the actual time spent in barrier routines is small. Instead, most of the overhead was caused by sequential wait and load imbalance.

Figure 3 graphically presents the reduction in idle time after applying optimizations for programs using large data sets on a 16 processor SP-2. The Y-axis represents idle time as a percentage of total execution time. Along the X-axis we present measurements for the original and optimized versions of each program.

Despite the fact most computation is parallel, measurements show sequential wait is a major source of overhead for software DSMs, possibly due to the long latency of interprocessor communication. Fortunately, optimizations can significantly reduce the amount of idle time for the applications studied. Aggressively expanding SPMD regions to include time-step loop appears to be successful in eliminating virtually all sequential-wait idle time in the test suite. Load imbalance is also significantly reduced after optimization. On average, total idle time is decreased from 37% to 12% of total execution time.

4.5 Impact of Compiler Optimizations on Speedups

We examine next how reductions in idle time affect application speedups. Table 4 displays speedups (both small and large data sets) on a 16-processor SP-2 for each application with different levels of optimization. As before, "dep" eliminates barriers in parallel regions based on data dependences, "comm" performs communication analysis using local analysis to eliminate barriers, "lazy" eliminates barriers guarding only anti-dependences, "aggr" aggressively expands parallel SPMD regions to include the outer time-step loop, and "suppress" sequentializes parallel loops with excessive communication. Optimizations are cumulative.

Figure 4 displays impact of compiler optimizations in a different manner, by calculating the percentage improvements in speedups due to each optimization. For each graph, the Y-axis measures improvement over unoptimized programs, the X-axis presents optimized versions of each program for both small and large data sets. Improvements from "suppress" (sequentializing expensive parallel loops) are reported only for `swm`, the one code where the optimization was applied. Except for `redblack` and `swm`, performance for "dep" is the same as that for simply merging adjacent parallel loops into the same parallel region.

Both the table on actual speedups and graph of percentage improvements indicate the impact of optimizations is significant, but varies over applications. For 16 processor runs with large data sets, average improvements from synchronization optimizations are 8.1% for dependence analysis, 9.9% for communication analysis, 10.5% when eliminating barriers based on lazy release consistency, 32.1% after aggressively expanding SPMD regions, and 35.1% by suppressing expensive parallelism. As expected, optimizations have greater impact for smaller data sets, since synchronization overhead is more significant. For small data set runs on 16 processors, average improvements are 9.9%, 12.2%, 14.9%, 40.4%, and 44% respectively.

Detailed analysis of the results for each application show some programs (`adi, dot, mult`) have few barriers to begin with, and are only affected by aggressive expansion of SPMD regions. Depending on whether parallel tasks have sufficient granularity, improvements range from low (`adi, mult`) to high (`dot`). The remaining applications benefit moderately from compile-time synchronization optimizations. Most optimizations affect only a few applications, but aggressive SPMD region expansion is useful for all applications and can result in significant improvements. Despite its complexity, aggressive SPMD transformations appear definitely worth incorporating in an optimizing compiler for software DSMs. Suppressing expensive parallel loops is applicable only to `swm`, but achieves major performance improvements and is also worth implementing.

Overall, synchronization overhead appears much greater for software DSMs than for shared-memory multiprocessors. Optimizations to reduce synchronization thus achieve better results for software DSMs than previously measured for shared-memory systems, even when fewer barriers are eliminated.

5 Related Work

Before studying methods for eliminating barrier synchronization, researchers investigated efficient use of data and event synchronization, where *post* and *wait* statements are used to synchronize between data items [26] or loop iterations [16]. Researchers compiling for fine-grain data-parallel languages sought to eliminate barriers following each expression evaluation [10, 21, 22]. Simple data dependence analysis can be used to reduce barrier synchronization by orders of magnitude, greatly improving performance. For barriers separating statements on the same loop level, Hatcher and Quinn use a two-dimensional radix sort to find the minimal number of barriers [10]. Philippsen and Heinz find the minimal number of barriers with an algorithm based on topological sort; they also attempt to minimize the amount of storage needed for intermediate results [21].

Eliminating barriers in compiler-parallelized codes is more difficult. Cytron *et al.* were the first to explore the possibilities of exploiting SPMD code for shared-memory multiprocessors [4]. They concentrated on safety concerns and the effect on privatization. O'Boyle and Bodin [20] present techniques similar to local communication analysis. They apply a classification algorithm to identify data dependences that cross processor boundaries, then apply heuristics based on max-cut to insert barrier synchronization and satisfy dependence. Stoher and O'Boyle [25] extend this work, presenting an optimal algorithm for eliminating barrier synchronization in perfectly nested loops.

There has been a large amount of research on software DSMs [1, 6, 19, 24, 30]. More recently, groups have examined combining compilers and software DSMs. Viswanathan and Larus developed a two-part predictive protocol for iterative computations for use in the data-parallel language C** [29]. Chandra and Larus evaluated combining the PGI HPF compiler and the Tempest software DSM system [2]. Results on a network of workstations connected by Myrinet indicates shared-memory versions of dense matrix programs achieve performance close to the message-passing codes generated. Granston and Wishoff suggest a number of compiler optimizations for software DSMs [7]. These include tiling loop iterations so computation is on partitioned matching page boundaries, aligning arrays to pages, and inserting hints to use weak coherence. Mirchandaney *et al.* propose using *section locks* and *broadcast barriers* to guide eager updates of data and reductions based on multiple-writer protocols [18].

Dwarkadas *et al.* applied compiler analysis to explicitly parallel programs to improve their performance on a software DSM [5]. By combining analysis in the ParaScope programming environment with TreadMarks, they were able to compute data access patterns at compile time and use it to help the runtime system aggregate communication and synchronization. Cox *et al.* conducted an experimental study to evaluate the performance of TreadMarks as a target for the Forge SPF shared-memory compiler from APR [3]. Results show that SPF/TreadMarks is slightly less efficient for dense-matrix programs, but outperforms compiler-generated message-passing code for irregular programs. They also identify opportunities for the compiler to eliminate unneeded barrier synchronization and aggregating messages in the shared-memory programs. Many of their suggestions are implemented in the SUIF/CVM system and are evaluated in this paper.

Rajamony and Cox developed a performance debugger for detecting unnecessary synchronization at run-time by instrumenting all loads and stores [23]. In the SPLASH application Water, it was able to detect barriers guarding only anti and output dependences that may be eliminated by applying odd-even renaming. In comparison, SUIF at compile time eliminates many barriers guarding only anti-dependences. Lu *et al.* found that software DSMs can also efficiently support irregular applications when using compile-time analysis to prefetch index arrays at run time [17].

Tzeng and Kongmunvattana improve the efficiency of barriers for software DSMs [28]. Our measurements show actual time spent in barrier routines is small compared to load imbalance caused by barriers, so improving barrier efficiency is likely to have only minor impact on application performance.

This paper extends our previous work on synchronization optimizations. We earlier presented communication analysis based on compile-time computation partition as a means of eliminating or replacing barrier synchronization [27]. Communication analysis techniques were similar to those used by distributed-memory compilers to calculate explicit communication [11]. Results were presented on a shared-memory multiprocessor.

Here we use communication analysis for a simple static chunk partitioning of loop iterations, rather than more sophisticated partitioning based on global automatic data decompositions. We appear to eliminate approximately the same barriers despite the decrease in precision. However, we needed to suppress small parallel loops in `swm` not present with automatic data decomposition. In addition, we present additional optimizations exploiting features of software DSMs such as lazy release consistency, as well as evaluate synchronization optimization on software DSMs. We find synchronization overhead is significantly higher for software DSMs, making optimizations more important.

We also earlier described compiler and run-time techniques for obtaining improved performance for compiler-parallelized programs on software DSMs [14]. Run-time enhancements such as the flush update protocol and support for reductions were helpful, but much synchronization overhead remained. More recently, we also presented techniques for reducing synchronization in software DSMs, including local communication analysis, customized nearest-neighbor synchronization [9]. Experiments found significant idle time remained, so in this paper we focused on additional optimizations, particularly techniques for aggressive expansion of SPMD regions. As a result, performance has improved.

6 Conclusions

In this paper we investigate ways to improve the performance of shared-memory parallelizing compilers targeting software DSMs. Software DSMs are appealing compilation targets, but experience indicate achieving good performance automatically is not simple and requires sophisticated compiler and run-time support. In this paper we present techniques for reducing synchronization overhead based on compile-time elimination of barriers. Our algorithm 1) exploits delayed updates in lazy-release-consistency software DSMs to eliminate barriers guarding only anti-dependences, and 2) aggressively expands parallel SPMD regions using a number of techniques for handling reduction initializations, control flow, and procedure calls, and 3) suppresses expensive parallel loops.

Experiments on a 16 processor IBM SP-2 indicate these techniques on average eliminate 66% of all barriers executed and improve parallel performance on average by 35%, and by much more for some programs. By reducing the synchronization overhead of compiler-parallelized programs on software DSMs, we are contributing to our long-term goal: making it easier for scientists and engineers to take advantage of the benefits of parallel computing.

7 Acknowledgements

The authors are grateful to Pete Keleher and Kritchal Thitikamol for providing CVM and valuable assistance on this paper.

References

[1] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 152–164, Oct. 1991.

[2] S. Chandra and J. Larus. Optimizing communication in HPF programs for fine-grain distributed shared memory. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[3] A. Cox, S. Dwarkadas, H. Lu, and W. Zwaenepoel. Evaluating the performance of software distributed shared memory as a target for parallelizing compilers. In *Proceedings of the 11th International Parallel Processing Symposium*, Geneva, Switzerland, Apr. 1997.

[4] R. Cytron, J. Lipkis, and E. Schonberg. A compiler-assisted approach to SPMD execution. In *Proceedings of Supercomputing '90*, New York, NY, Nov. 1990.

[5] S. Dwarkadas, A. Cox, and W. Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, Boston, MA, Oct. 1996.

[6] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 244–255, May 1993.

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

[8] M. Hall, S. Amarasinghe, B. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, Dec. 1995.

[9] H. Han, C.-W. Tseng, and P. Keleher. Reducing synchronization overhead for compiler-parallelized codes on software DSMs. In *Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing*, Minneapolis, MN, Aug. 1997.

[10] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.

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

[12] P. Keleher. The relative importance of concurrent writers and weak consistency models. In *16th International Conference on Distributed Computing Systems*, Hong Kong, May 1996.

[13] 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.

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

[15] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[16] Z. Li. Compiler algorithms for event variable synchronization. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[17] H. Lu, A. Cox, S. Dwarkadas, R. Rajamony, and W. Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[18] R. Mirchandaney, S. Hiranandani, and A. Sethi. Improving the performance of DSM systems via compiler involvement. In *Proceedings of Supercomputing '94*, Washington, DC, Nov. 1994.

[19] S. Mukherjee, S. Sharma, M. Hill, J. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[20] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared virtual memory systems. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, Barcelona, Spain, July 1995.

[21] M. Philippsen and E. Heinz. Automatic synchronization elimination in synchronous FORALLs. In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, Feb. 1995.

[22] S. Prakash, M. Dhagat, and R. Bagrodia. Synchronization issues in data-parallel languages. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, Aug. 1993.

[23] R. Rajamony and A. Cox. A performance debugger for eliminating excess synchronization in shared-memory parallel programs. In *Proceedings of the Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Feb. 1996.

[24] D. Scales and K. Gharachorloo. Design and performance of the Shasta distributed shared memory protocol. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[25] E. Stohr and M. O'Boyle. A graph based approach to barrier synchronisation minimisation. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[26] P. Tang, P. Yew, and C. Zhu. Compiler techniques for data synchronization in nested parallel loops. In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[27] C.-W. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[28] N.-F. Tzeng and A. Kongmunvattana. Distributed shared memory systems with improved barrier synchronization and data transfer. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, Vienna, Austria, July 1997.

[29] G. Viswanathan and J. Larus. Compiler-directed shared-memory communication for iterative parallel computations. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, Nov. 1996.

[30] Y. Zhou, L. Iftode, J. Singh, K. Li, B. Toonen, I. Schoinas, M. Hill, and D. Wood. Relaxed consistency and coherence granularity ind DSM systems: A performance evaluation. In *Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.