Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs

Hwansoo Han, Chau-Wen Tseng, Pete Keleher

Dept. of Computer Science
University of Maryland
College Park, MD 20742

Abstract

Software distributed-shared-memory (DSM) systems provide an appealing target for parallelizing compilers due to their flexibility. Previous studies demonstrate such systems can provide performance comparable to message-passing compilers for dense-matrix kernels. However, synchronization and load imbalance are significant sources of overhead. In this paper, we investigate the impact of compilation techniques for eliminating barrier synchronization overhead in software DSMs. Our compile-time barrier elimination algorithm extends previous techniques in three ways: 1) we perform inexpensive communication analysis through local subscript analysis when using chunk iteration partitioning for parallel loops, 2) we exploit delayed updates in lazy-release-consistency DSMs to eliminate barriers guarding only anti-dependences, 3) when possible we replace barriers with customized nearest-neighbor synchronization. Experiments on an IBM SP-2 indicate these techniques can improve parallel performance by 20% on average and by up to 60% for some applications.

1 Introduction

It is generally agreed 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. However, users are willing to write message-passing programs only for a few important applications because it takes too much time and effort. Compilers for languages such as High Performance Fortran [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 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 the large-scale parallel machines as well as the low-end, but more pervasive multiprocessor workstations.

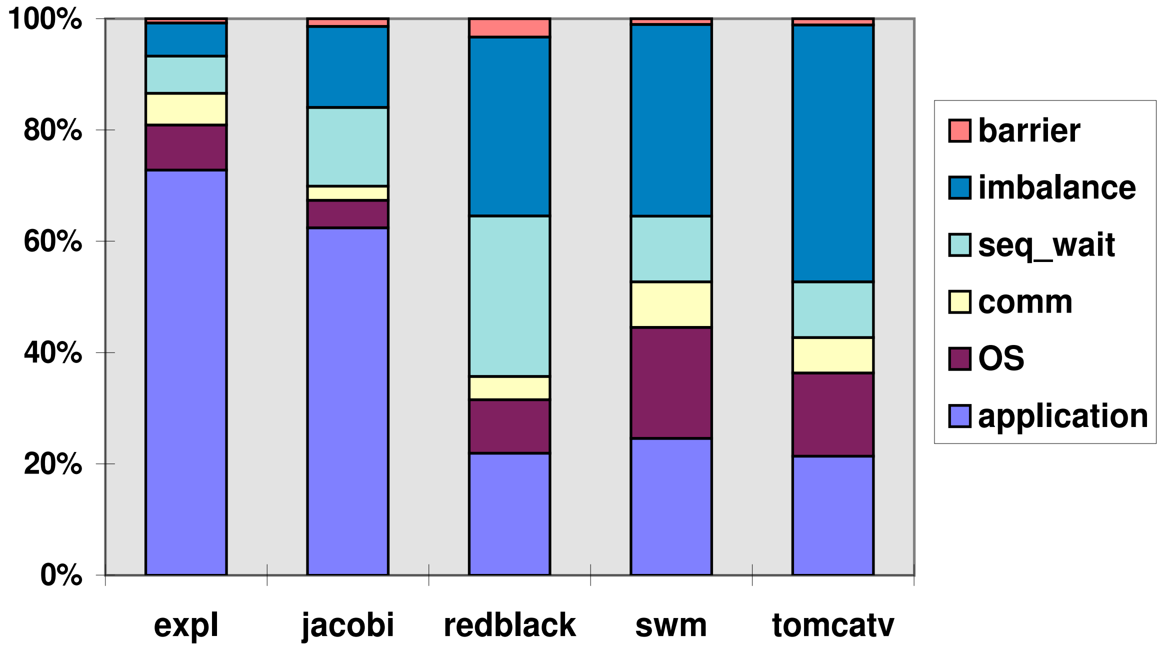

Figure 1: Breakdown of Execution Time (16 Processor SP-2)

Shared-memory parallelizing compilers are easy to use, flexible, and can accept a wide range of applications. Results from several recent studies [4, 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.

To determine sources of overhead, we instrumented five compiler-parallelized applications running on top of a software DSM. Figure 1 presents the results, with execution time summed up over all processors for each category. Total execution times is split into application processing time, OS overhead, communication cost, sequential wait (idle time waiting for next parallel task), load imbalance (idle time waiting for completion of current parallel task), and barrier overhead (time spent executing barrier code).

Our measurements demonstrate synchronization overhead 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 load imbalance induced by frequent barriers comprises a large percentage of total execution time. 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 load imbalance 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. 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 [9] shared-memory compiler. This paper makes the following contributions:

- eliminate barriers by inexpensively detecting communication using local subscript analysis
- exploiting lazy release consistency to eliminate barriers guarding only anti-dependences
- replacing barriers with customized nearest-neighbor synchronization
- empirical evaluation of compiler synchronization optimizations for software DSMs

We begin by describing the 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

We introduce the two components of our prototype system, the SUIF shared-memory compiler and the CVM software DSM. We also review previous compiler techniques for reducing synchronization overhead.

2.1 SUIF Shared-Memory Compiler

SUIF is an optimizing and parallelizing compiler developed at Stanford [9]. 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 just like 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* [13], a derivation of *release consistency*. Release consistency allows a processor to delay making modifications to shared data visible to other processors until special *acquire* or *release* synchronization accesses occur. Lazy release consistency postpones the propagation of modifications further; updates to shared data do not have to be made visible to a processor until the next time that processor acquires a released synchronization variable. Experiments show that lazy-release-consistency protocols generally cause less communication than release consistency [7]. 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 run-time 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 two 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 synchronization overhead is particularly significant as the number of processors increases, since the interval between barriers decreases as computation is partitioned across more processors.

In previous work, we developed compiler algorithms for barrier elimination [25]. 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. Each SPMD region has an implicit broadcast barrier preceding the region and a barrier following the region.

{ Default Model }

```
DOALL I = 1,N
   ...
ENDDO
DOALL J = 1,N
   ...
ENDDO
   ⇓
broadcast barrier
DO I = LB1,UB1
   ...
ENDDO
barrier
broadcast barrier
DO J = LB2,UB2
   ...
ENDDO
barrier
```

(A)

{ Merge SPMD Regions }

```
DOALL I = 1,N
   ...
ENDDO
DOALL J = 1,N
   ...
ENDDO
   ⇓
broadcast barrier
DO I = LB1,UB1
   ...
ENDDO
barrier
DO J = LB2,UB2
   ...
ENDDO
barrier
```

(B)

{ Dependence Analysis }

```
DOALL I = 1,N
   A(I) =
ENDDO
DOALL J = N+1,2*N
   B(J) = A(J)
ENDDO
   ⇓
broadcast barrier
DO I = LB1,UB1
   A(I) =
ENDDO
DO J = LB2,UB2
   B(J) = A(J)
ENDDO
barrier
```

(C)

{ Communication Analysis }

```
DISTRIBUTE (BLOCK) :: A,B
DOALL I = 1,N
   A(I) =
ENDDO
DOALL J = 1,N
   B(J) = A(J)
ENDDO
   ⇓
broadcast barrier
DO I = LB1,UB1
   A(I) =
ENDDO
DO J = LB1,UB1
   B(J) = A(J)
ENDDO
barrier
```

(D)

{ Nearest-Neighbor Sync }

```
DISTRIBUTE (BLOCK) :: A,B
DOALL I = 1,N
   A(I) =
ENDDO
DOALL J = 1,N
   B(J) = A(J-1) + A(J+1)
ENDDO
   ⇓
broadcast barrier
DO I = LB1,UB1
   A(I) =
ENDDO
sync_with_neighbors
DO J = LB1,UB1
   B(J) = A(J-1) + A(J+1)
ENDDO
barrier
```

(E)

Figure 2: Optimization Examples

Next, we *merge* adjacent SPMD regions where possible, such as in the example shown in Figure 2(B). When two SPMD regions are merged, an explicit barrier is placed between the two regions, resulting in the elimination of a broadcast barrier. SPMD regions may also be expanded to enclose outer sequential loops when the body of the entire loop body is within a SPMD region. Merging SPMD regions will directly eliminate some barrier synchronization. Furthermore, by placing multiple loops in the same SPMD region, we make barriers explicit and provide opportunities for barrier elimination or replacement.

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(C). 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 in Figure 2(D). Finally, 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 shown in Figure 2(E).

3 Reducing Synchronization Overhead

Later in this paper, we will examine the impact of barrier elimination algorithms on the performance of SUIF-parallelized programs for CVM. In this section we discuss novel extensions to compiler techniques for reducing synchronization overhead.

3.1 Communication Analysis Using Local Subscripts

Previous communication analysis relied on compile-time information on the data and computation decomposition selected for a program to precisely determine whether interprocessor communication takes place [25]. We find that an alternative communication analysis technique based on *local subscript analysis* can yield good results with much less complex analysis. The Local subscript analysis algorithm is shown in Figure 3.

We assume dependence analysis has already eliminated all array reference pairs between loop nests proven to access disjoint memory locations. Local subscript analysis relies on the compiler selecting a consistent chunk partitioning of parallel loop iterations. Consistent iteration partitioning of loops with identical loop bounds will then always assign the same loop iterations to each processor. If any subscript pair containing the parallel loop index is identical, each processor will only access local data. If the subscripts differ by only a constant, then each processor will access remote data a constant number of processors away. Since constant differences in subscripts are usually small, processors will end up accessing remote data on neighboring processors, allowing barriers to be replaced by nearest-neighbor synchronization.

Local subscript analysis is designed to complement full communication analysis. It has two advantages, efficiency and applicability. Because it only relies on local symbolic information, local subscript analysis can quickly eliminate simple array reference pairs that do not require synchronization. More expensive communication analysis may be applied if the local test fails. Local subscript analysis can also potentially be applied in more cases than standard communication analysis, since it relies only on local program information and a consistent chunk iteration partitioning. Communication analysis, in comparison, relies on knowing the compile-time computation and data decomposition. Additionally, local subscript analysis only needs to ensure a

```
for each pair of parallel loop nests
    if loop bounds are identical
        for each dependence crossing loop nests
            examine array reference pair at endpoints
            find subscript pairs containing parallel loop index
            if any subscript pair is identical
                no barrier needed for this dependence
            else if any subscript pair differs by constant
                sync_neighbor needed for this dependence
            else
                barrier needed for this dependence
        if barrier needed for any dependence
            insert barrier
        else if sync_neighbor needed for any dependence
            insert nearest-neighbor synchronization
        else
            no barrier needed between loop nests
    else
        insert barrier
```

Figure 3: Local Subscript Analysis Algorithm

subscript expression pair containing the parallel loop index is identical. Communication analysis must calculate what data is communicated, and can be disabled by a single complex array subscript. For instance, in Figure 4(A), local subscript analysis does not need to fully analyze $f_1()$ or $f_2()$ before eliminating the barrier, unlike communication analysis. Similarly, local subscript analysis can determine that only nearest-neighbor synchronization is needed even if some subscripts are too complex to analyze, as shown in Figure 4(B).

3.2 Exploiting Lazy Release Consistency

A second enhancement to our compile-time barrier elimination algorithm is 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.

To see how the compiler can use this property, consider the example shown in Figure 4(c). 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

{ Local Subscript Analysis }

```
DOALL J = 1,N
  DO I = 1,N
    A(f1(I),f2(J)) =
  ENDDO
ENDDO
DOALL J = 1,N
  DO I = 1,N
    B(...) = A(f1(I),f2(J))
  ENDDO
ENDDO
  ⇓
broadcast barrier
DO J = LB1,UB1
  DO I = 1,N
    A(f1(I),f2(J)) =
  ENDDO
ENDDO
DO J = LB1,UB1
  DO I = 1,N
    B(...) = A(f1(I),f2(J))
  ENDDO
ENDDO
barrier
```

(A)

{ Local Subscript Analysis 2 }

```
DOALL J = 1,N
  DO I = 1,N
    A(f1(I),J) =
  ENDDO
ENDDO
DOALL J = 1,N
  DO I = 1,N
    B(...) = A(f1(I),J-1)
  ENDDO
ENDDO
  ⇓
broadcast barrier
DO J = LB1,UB1
  DO I = 1,N
    A(f1(I),J) =
  ENDDO
ENDDO
sync_with_neighbors
DO J = LB1,UB1
  DO I = 1,N
    B(...) = A(f1(I),J-1)
  ENDDO
ENDDO
barrier
```

(B)

{ Lazy Release Consistency }

```
DO TIME =
  DOALL J = 1,N
    DO I = 1,N
      A(I,J) = B(I,J-1)+B(I,J+1)
    ENDDO
  ENDDO
  DOALL J = 1,N
    DO I = 1,N
      B(I,J) = A(I,J)
    ENDDO
  ENDDO
ENDDO
  ⇓
broadcast barrier
DO TIME =
  DO J = LB1,UB1
    DO I = 1,N
      A(I,J) = B(I,J-1)+B(I,J+1)
    ENDDO
  ENDDO
  sync_first_time
  DO J = LB1,UB1
    DO I = 1,N
      B(I,J) = A(I,J)
    ENDDO
  ENDDO
  sync_with_neighbors
ENDDO
barrier
```

(C)

Figure 4: Advanced Optimization Examples

B on another processor will not be made visible until the two processors synchronize. Since there are no other loop-independent dependences between the two loop nests, barrier synchronization between them is not always required.

The one case where synchronization is needed for anti-dependences 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. Our implementation of nearest-neighbor synchronization solves this problem by invoking a global barrier the first time it is invoked at each location in the program. 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 is not needed.

Finally, in Figure 4(c) a cross-processor true/flow dependence (read-after-write) exists which does need synchronization. It is the dependence between definitions of B in the second loop nest and reads of nonlocal values of B in the first loop nest. This dependence is carried by the outer TIME loop, since the endpoints of the dependence occur on different iterations of the TIME loop. The compiler normally inserts a barrier as the last statement of the TIME loop, but local subscript analysis can show only nearest-neighbor synchronization is needed.

3.3 Customized Nearest-Neighbor Synchronization

At some barriers, the compiler can detect communication only takes place between neighboring processors [25]. To take advantage of this information, we implemented a customized routine for nearest-neighbor synchronization (where each processor has either zero, one, or two neighbors) directly in CVM. The routine sends a single message to each neighboring processor upon arrival, and continues as soon as messages are received from all neighboring processors. In comparison, for normal global barriers all processors send a single message to the barrier master, which broadcasts a reply once all processors have checked in.

The customized nearest-neighbor synchronization has several advantages over standard global barriers. The most important is that nearest-neighbor synchronization allows at least some of the induced load imbalance to smooth out before it delays all processors. However, this benefit usually occurs only if there are multiple nearest-neighbor synchronization events invoked in sequence. If nearest-neighbor synchronization and global barriers are executed in alternating sequence, opportunities to smooth out load imbalance are lessened. In addition, the serial bottleneck of the barrier master is avoided. This is not a large advantage for the size of the system that we are currently evaluating, but should be significant for larger systems. Finally, common messages can be used to carry both synchronization and data, because both flow only between neighbors.

4 Experimental Results

4.1 Applications

We evaluated the performance of our compiler/software DSM interface with five programs shown in Table 1. The "Granularity" column refers to the average length in seconds of a parallelized loop. Except where indicated, numbers below refer to the larger data set for each application. `expl`, and `redblack` are dense stencil kernels typically found in iterative PDE solvers. `jacobi` is a stencil kernel combined with a convergence test that checks the residual value using a `max` reduction. `swm` and `tomcatv` are programs from the SPEC benchmark suite containing a mixture of stencils and reductions. We used the version of `tomcatv` from APR whose arrays have been transposed to improve data locality.

All applications were originally written in Fortran, and typically contain an initialization section followed by iterations of a time-step loop. Statistics and timings are collected after the initialization section. Optimized versions of each program were automatically generated by the SUIF compiler. The compiler analyzed but was unable to apply synchronization optimizations to

Name	Description	Problem Sizes		Granularity (secs)	
		Small	Large	Small	Large
expl	Explicit Hydrodynamics (Livermore 18)	256^2	512^2	0.06	0.34
jacobi	Jacobi Iteration w/Convergence Test	512^2	1024^2	0.06	0.91
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

four other programs, indicating that not all programs may benefit from the techniques presented in this paper.

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 messages between nodes require around 160 μsecs, but round-trip messages used by the software DSM to communicate nonlocal data can take from 900 to 1400 μsecs. Both communication and computation are slow, but the communication/computation ratio should be similar to newer systems. Since the impact of synchronization optimizations is proportional to the ratio (more impact when communication costs are relatively high), our experimental results should also apply to newer parallel architectures.

In our experiments, CVM [12] applications written in Fortran 77 were automatically parallelized by the Stanford SUIF parallelizing compiler version 1.1.2 [9], 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]. These enhancements reduce communication overhead by exploiting communication patterns in iterative scientific computations to combine communication for multiple nonlocal accesses with barrier synchronization messages. Without such communication optimizations, reducing synchronization overhead would be even more important, since uneven communication overhead would introduce additional load imbalance between processors.

Program	Doalls in program original number	Doalls in program % eliminated	Barriers in program original number	% eliminated merge	% eliminated depend	% eliminated comm	% eliminated lazy	% replaced comm	% replaced lazy
expl	3	67	6	33	33	33	50	33	17
jacobi	2	50	4	25	25	25	50	25	–
redblack	4	75	8	38	63	63	63	13	13
swm	11	36	22	18	27	27	27	–	–
tomcatv	8	63	16	31	31	62	62	–	–
Average	5.6	58	11.2	29	36	42	50	14	6.0

Table 2: Static Measurement of Synchronization Optimizations

Program	Doalls executed original number	Doalls executed % eliminated	Barriers executed by program original number	% eliminated merge	% eliminated depend	% eliminated comm	% eliminated lazy	% replaced comm	% replaced lazy
expl	60	67	120	33	33	33	50	33	17
jacobi	40	50	80	25	25	25	50	25	–
redblack	80	75	160	38	63	63	63	13	13
swm	265	33	530	17	33	33	33	–	–
tomcatv	140	71	280	36	36	71	71	–	–
Average	117	59	234	30	38	45	53	14	6.0

Table 3: Dynamic Measurement of Synchronization Optimizations

4.3 Effectiveness of Compiler Synchronization Optimizations

First, we examine the effectiveness of compiler algorithms in eliminating synchronization. Table 2 displays the number of parallel loops (doalls) and barriers found in each program at compile time, and the percentage eliminated by different levels of optimization. Table 3 presents the same information for parallel loops and barriers executed dynamically at run time for each application. The first two columns for "doalls" indicate the number of parallel loops executed in the original program and the percentage reduction by merging doalls into the same parallel region. 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 SPMD region, "depend" eliminates barriers in parallel regions based on data dependence analysis, "comm" performs communication analysis using local subscript analysis to eliminate barriers, "lazy" eliminates barriers guarding only anti-dependences. Communication analysis may also replace barriers with nearest-neighbor synchronization, Optimizations are cumulative.

We see from both tables that the compiler is effective at eliminating parallel loops and barriers encountered during compilation, with roughly similar benefits for the number of parallel loops and barriers actually executed by the application at run time. Examining the run-time measurements in Table 3, we find the compiler is quite successful in discovering parallel loops which may be merged into a single SPMD region, eliminating on average 59% of parallel invocations and 30% 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 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 anti-dependences. Applying all optimizations, on average 53% of all barrier executions are eliminated in these five programs, with 6% of barriers replaced by nearest-neighbor synchronization.

4.4 Impact of Compiler Optimizations on Synchronization Overhead

In order to evaluate synchronization optimizations in more detail, we instrumented CVM to directly measure barrier overhead (time spent executing barrier code), sequential wait (idle time waiting for next parallel task), and load imbalance (idle time waiting for completion of current parallel task). We found the actual time spent in barrier routines is small. Instead, most of the overhead was caused by sequential wait and load imbalance. Table 4 shows sequential wait and load imbalance as a percentage of overall execution time. The measurements are for 16 processor SP-2 runs with small and large data sets. Average sequential wait and load imbalance are 14% and 27% of execution time for large data sets, and 23% and 31% for small data sets respectively. Load imbalance takes up less percentage of execution time with large data sets, because the larger amount of data on individual processors gives the greater chance to smooth out load imbalance. We also measured the impact of synchronization optimizations on reducing idle time (sequential wait + load imbalance). Table 4 displays the percentage reduction in idle time after applying different levels of optimization. Figure 5 graphically presents the same information for the large data sets. We see that optimizations can significantly reduce idle time for some of the applications studied (72% decrease for `redblack` with small data set). Average load imbalances decrease by 28%, 32%, and 34% with large data sets and by 34%, 37%, and 43% with small data sets for three optimization levels, respectively.

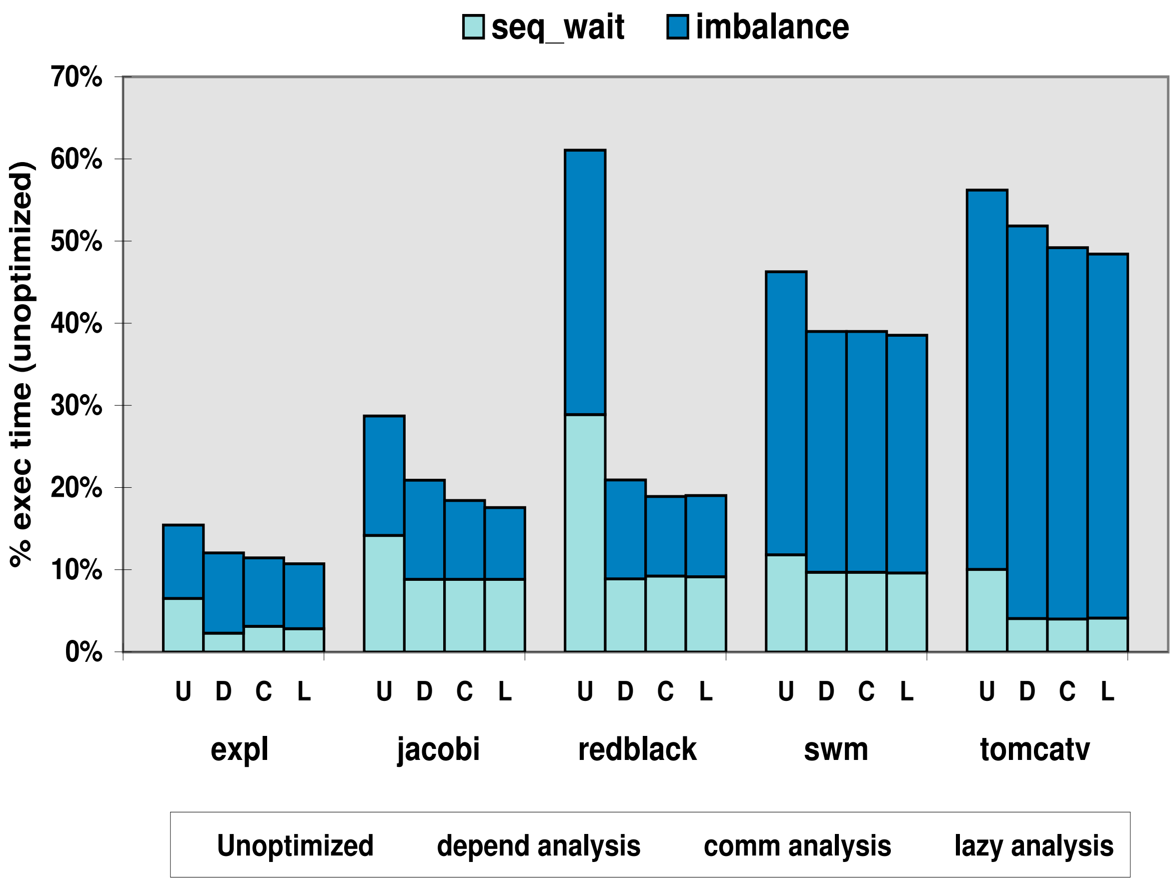

Figure 5: Effect of Optimizations on Idle Time (16 Processors)

		expl		jacobi		redblack		swm		tomcatv		Average	
		sm	lg	sm	lg	sm	lg	sm	lg	sm	lg	sm	lg
% Exec time (unoptimized)	barrier	2.1	0.7	2.6	1.3	4.0	3.2	1.5	1.0	1.8	1.1	2.4	1.5
	seq_wait	20	6.5	27	14	36	29	15	12	18	10	23	14
	imbalance	15	8.9	24	15	36	32	32	34	47	46	31	27
% Decrease in idle time	depend	37	22	31	27	67	66	17	16	18	7.7	34	28
	comm	36	26	37	36	72	69	17	16	23	12	37	32
	lazy	48	31	52	39	72	69	17	17	25	14	43	34

Table 4: Effect of Optimizations on Idle Time (16 Processors)

4.5 Impact of Compiler Optimizations on Program Performance

Figure 6 displays the impact of synchronization optimizations on application performance on the SP-2, for both small and large data sets. For each graph, the Y-axis measures improvement over unoptimized programs, the X-axis presents three optimized versions of each program for both 8 and 16 processor runs: "depend analysis" merges parallel loops into common SPMD regions and eliminates barriers based on data dependences, "comm analysis" uses local subscript analysis to eliminate barriers or replace them with nearest-neighbor synchronization, "lazy analysis" eliminates barriers guarding only anti-dependences. Except for `redblack` and `swm`, performance for "depend analysis" is the same as that for simply merging adjacent parallel loops into the same parallel region. Optimizations are cumulative.

Table 6 displays speedups and percentage improvements due to optimizations in detail. Performance for applications cover a broad range. For 16 processor runs with large data sets, average improvement from synchronization optimizations is 13% for dependence analysis, 17% for communication analysis, and 18% when also eliminating barriers using lazy release consistency. As expected, optimizations have greater impact for smaller data sets and more processors, since synchronization overhead is more significant. For small data set runs on 16 processors, average improvements increase to 20%, 26%, and 28% respectively. For 8 processor runs improvements from synchronization optimizations drop to 8%, 11%, and 11% with large data sets and 13%, 18%, and 19% with small data sets for the three optimization levels, respectively.

4.6 Impact of Nearest-Neighbor Synchronization on Performance

Using communication analysis, SUIF was able to replace two barriers with nearest-neighbor synchronization in `expl`, and one barrier each in `jacobi` and `redblack`. The performance improvements due to replacing barriers are shown in Table 5. As with all synchronization optimizations, impact is heightened for smaller data sizes. Performance improvements are due to a combination of better load balancing and fewer messages; the amount of data communication that is piggybacked on synchronization messages tripled for some applications when using nearest-neighbor barriers. Nonetheless, the overall impact on performance is limited. The primary reason is that many of the barrier synchronizations that are prime candidates to be replaced by nearest-neighbor synchronization can be eliminated instead. However, nearest-neighbor synchronization may prove more important when using more than 16 processors.

Execution time	expl		jacobi		redblack		swm		tomcatv		Average	
(depend vs. comm)	sm	lg	sm	lg	sm	lg	sm	lg	sm	lg	sm	lg
% Improve	3.5	2.4	5.5	2.5	9.3	5.0	–	–	–	–	3.7	2.0

Table 5: Impact of Nearest-Neighbor Barriers (16 Processors)

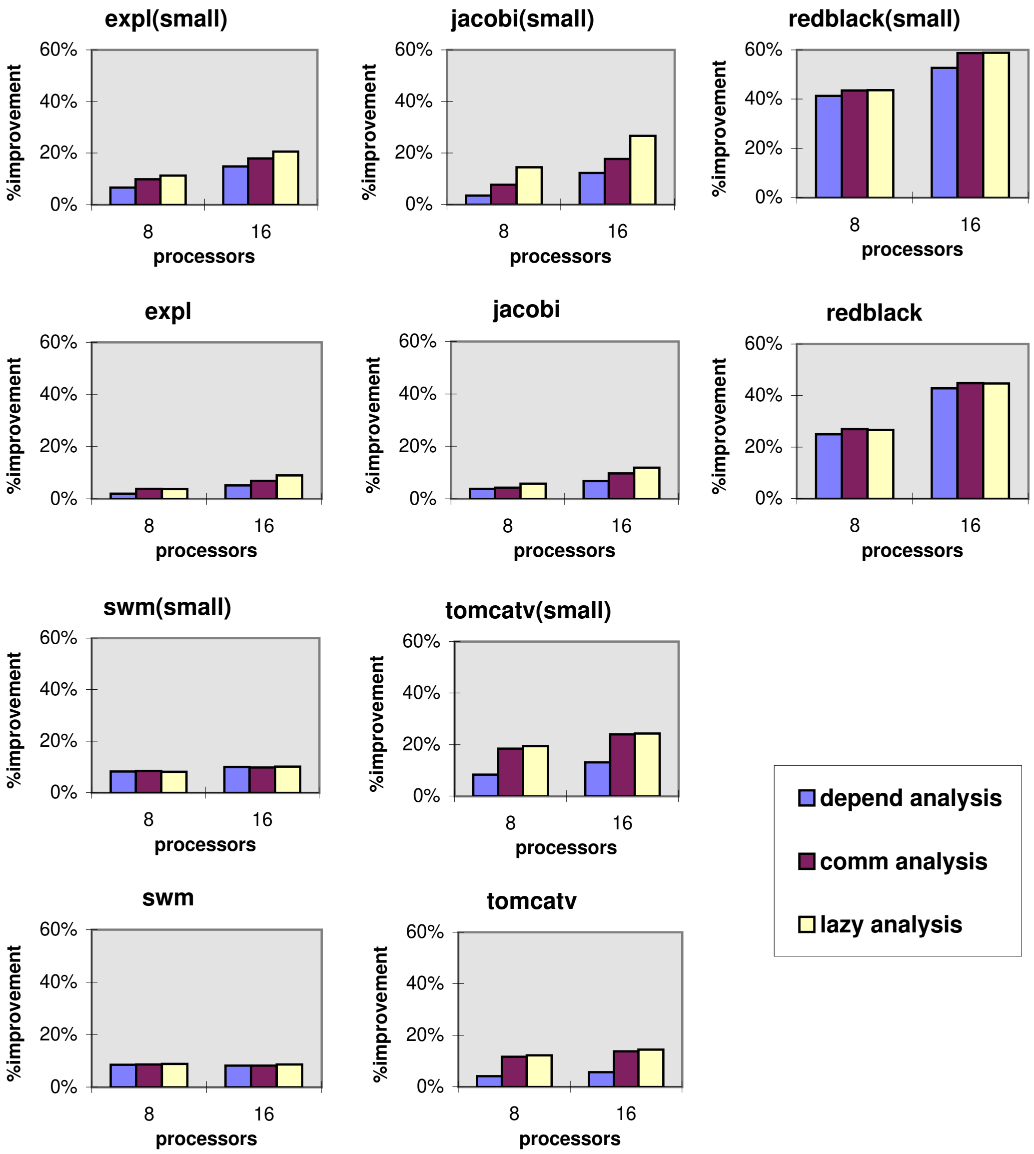

Figure 6: Impact of Barrier Elimination (8 & 16 Processors)

		expl		jacobi		redblack		swm		tomcatv		Average	
		sm	lg	sm	lg	sm	lg	sm	lg	sm	lg	sm	lg
Speedup	unoptimized	5.7	13	6.0	12	1.3	4.0	1.4	2.2	1.3	3.1	3.1	6.7
	depend	6.6	13	6.8	13	2.7	6.9	1.5	2.4	1.5	3.3	3.8	7.7
	comm	6.9	14	7.3	13	3.0	7.2	1.5	2.4	1.7	3.6	4.1	7.9
	lazy	7.1	14	8.2	13	3.1	7.2	1.5	2.4	1.7	3.6	4.3	8.1
% Improve	depend	15	5.1	12	6.8	53	43	5.2	5.0	13	5.6	20	13
	comm	18	6.9	18	9.7	59	45	9.8	8.2	24	14	26	17
	lazy	21	9.0	27	12	59	45	10	8.6	24	14	28	18

Table 6: Impact of Barrier Elimination (16 Processors)

4.7 Discussion

To summarize, our experiments indicate compile-time synchronization optimizations are indeed useful for reducing idle time and improving overall program performance. For the five programs examined, the majority of the improvements result from applying previously developed synchronization optimizations, such as merging adjacent parellel regions and eliminating barriers using dependence or communication analysis. The novel synchronization optimizations targeting software DSMs described in this paper are useful, but have a lesser impact on performance.

Our main contribution is thus demonstrating existing synchronization optimizations can significantly improve performance on software DSMs (by about 20%) even though they reduce the amount of synchronization only by about 50%. On hardware DSMs, due to efficient communication mechanisms, order of magnitude reductions in synchronization were needed before noticable performance improvements would result. The sensitivity to load imbalance in software DSMs also encourages the development of new synchronization optimizations.

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 [24] or loop iterations [16]. Researchers compiling for fine-grain data-parallel languages sought to eliminate barriers following each expression evaluation [10, 20, 21]. 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 [20].

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 [5]. They concentrated on safety concerns and the effect on privatization. In previous work [25] we presented techniques to eliminate or lessen synchronization based on communication analysis used by distributed-memory compilers to calculate explicit communication [11].

O'Boyle and Bodin [19] present techniques similar to local subscript 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 [23] 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, 7, 18]. 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** [27]. Chandra and Larus evaluated combining the PGI HPF compiler and the Tempest software DSM system [2, 3]. 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 [8]. 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 [17].

Dwarkadas *et al.* applied compiler analysis to explicitly parallel programs to improve their performance on a software DSM [6]. 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 [4]. 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 [22]. 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.

Tzeng and Kongmunvattana improve the efficiency of barriers for software DSMs [26]. 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.

6 Conclusions

In this paper we investigate ways to improve the performance of shared-memory parallelizing compilers targeting software DSMs. We present techniques for reducing synchronization overhead based on compile-time elimination of barriers. Our algorithm extends previous techniques by 1) inexpensively performing communication analysis using local subscript analysis by exploiting chunk iteration partitioning, 2) exploiting delayed updates in lazy-release-consistency software DSMs to eliminate barriers guarding only anti-dependences, 3) replacing barrier synchronization with customized nearest-neighbor synchronization. Experiments on an IBM SP-2 indicate these techniques on average eliminate 50% of all barriers executed and improve parallel performance by 10-30%, depending on data set size and number of processors. Synchronization optimizations become more important as the number of processors grows. By reducing the synchronization overhead of compiler-parallelized programs on software DSMs, we believe that we are contributing to our long-term goal: effectively running applications that are too complex to be compiled directly to message-passing code.

References

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

[2] S. Chandra and J.R. Larus. HPF on fine-grain distributed shared memory: Early experience. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996.

[3] S. Chandra and J.R. 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.

[4] 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, April 1997.

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

[6] 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, October 1996.

[7] S. Dwarkadas, P. Keleher, A.L. 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.

[8] 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.

[9] 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, December 1995.

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

[18] 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.

[19] 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.

[20] 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, February 1995.

[21] 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, August 1993.

[22] R. Rajamony and A.L. 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)*, February 1996.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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