# Compiler Optimizations for Eliminating Barrier Synchronization

Chau-Wen Tseng

Computer Systems Laboratory
Stanford University
Stanford, CA 94305-4070
*tseng@cs.stanford.edu*

## Abstract

This paper presents novel compiler optimizations for reducing synchronization overhead in compiler-parallelized scientific codes. A hybrid programming model is employed to combine the flexibility of the fork-join model with the precision and power of the single-program, multiple data (SPMD) model. By exploiting compile-time computation partitions, communication analysis can eliminate barrier synchronization or replace it with less expensive forms of synchronization. We show computation partitions and data communication can be represented as systems of symbolic linear inequalities for high flexibility and precision. These optimizations has been implemented in the Stanford SUIF compiler. We extensively evaluate their performance using standard benchmark suites. Experimental results show barrier synchronization is reduced 29% on average and by several orders of magnitude for certain programs.

## 1 Introduction

Parallel machines with shared address spaces and coherent caches provide an attractive source of computation power for scientists and engineers because they are easy to program. Examples of such machines include the Kendall Square Research KSR-1, Convex Exemplar, and SGI Challenge, all of which provide commercial parallelizing compilers. In many cases these compilers can automatically detect parallelism and generate efficient code for small numbers of processors, achieving respectable speedups with little or no additional programmer effort.

Unfortunately, parallelizing compilers are by no means infallible. While compilers can usually detect available loop-level parallelism for inner loops, they tend to be weak in analyzing large sections of code enclosed by outer loops. Interprocedural parallelization has proven successful in identifying large parallel loops for some programs [8, 15], but in most cases compilers will identify many parallel inner loops rather than a few large parallel outer loops.

When the amount of computation in a parallel loop (also known as *granularity*) is small, parallel speedup can be significantly limited due to barrier synchronization overhead [10]. Barriers are expensive for two reasons. First, executing a barrier has some run-time overhead that typically grow 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 [16]. Eliminating the barrier allows small perturbations in task execution time to even out, taking advantage of the loosely coupled nature of multiprocessors. Barrier synchronization overhead is particularly significant when attempting to use many processors, since the interval between barriers decreases as computation is partitioned across more processors.

To ameliorate the problem of excessive synchronization overhead in compiler-parallelized codes, we have developed new compiler algorithms to either eliminate barrier synchronization or replace it with less expensive forms of synchronization. Our optimization is based on three factors: encapsulating parallel regions in *single-program, multiple-data* (SPMD) sections, compile-time partitioning of computation, and communication analysis similar to that performed by compilers for message-passing machines. The contributions of this paper include:

- Novel algorithms for using communication analysis to eliminate or replace barrier synchronization in compiler-parallelized programs.
- Lessons from implementing the optimization in the SUIF compiler [28] using systems of symbolic linear inequalities.
- Extensive evaluation using a large number of standard benchmarks suites, averaging 29% reduction in number of barriers executed.

We expect the importance of reducing synchronization overhead to increase with advancing technology, since the gap between uniprocessor performance and inter-processor latency continues to widen. It is particularly important to reduce barrier synchronization when compiling programs for software distributed-shared memory (DSM) systems [12], since software barrier costs are dramatically higher than the barriers on current multiprocessors architectures.

The rest of this paper presents approaches for eliminating barrier synchronization, an overview of our algorithm, some implementation details, and experimental results. We conclude with a discussion of related work.

## 2 Optimization Approaches

### 2.1 Fork-Join and SPMD Programming Models

Traditional shared-memory parallelizing compilers employ 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 is encountered. After completing its portion of the parallel loop, the master waits for all workers to complete (join) before continuing execution.

This research was supported in part by ARPA contract DABT63-94-C-0054 and an NSF CISE postdoctoral fellowship in Experimental Science.

In *Proceedings of the 5th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, Santa Barbara, CA, July 1995

```
{ Parallel Loop }
DOALL I = 1,N
   A(I) = . . .
ENDDO
   ⇓
broadcast
BK = N/NPROCS
LB = MAX(1, (MYID*BK)+1)
UB = MIN(N, (MYID+1)*BK)
DO I = LB,UB
   A(I) = . . .
ENDDO
barrier
```

```
{ Nested Parallel Loop }
DO I = 1,N
   DOALL J = 1,N
      A(J,I) = A(J,I-1)
   ENDDO
ENDDO
   ⇓
DO I = 1,N
   broadcast
   DO J = LB,UB
      A(J,I) = A(J,I-1)
   ENDDO
   barrier
ENDDO
```

Figure 1: Implicit Synchronization Overhead

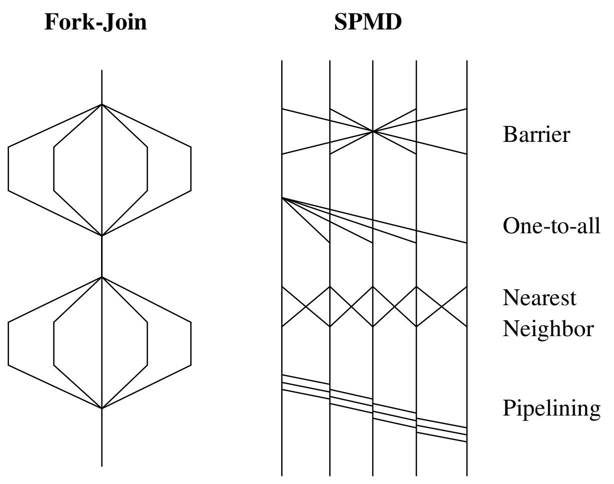


Figure 2: Fork-join vs. SPMD Models

The fork-join model is flexible and can easily handle sequential portions of the computation; however, it imposes two synchronization events per parallel loop. Figure 1 shows how a parallel loop is typically translated by the back end of a compiler. First, a *broadcast* is inserted before the loop body to wake up available worker threads. A *barrier* is then inserted after the loop body to ensure all worker threads have completed before the master can continue. Between the broadcast and the barrier threads execute computation in parallel. The local processor ID may be used to distinguish threads; in the figure it is used to calculate local loop bounds for each thread. Barrier costs for the fork-join model can be high, especially when parallel loops are nested in outer sequential loops.

In comparison, compilers for message-passing machines generate code according to a *single-program, multiple-data* (SPMD) model, where all threads execute the entire program [18]. Sequential computation is either replicated or explicitly guarded to limit execution to a single thread. When a parallel loop is encountered, threads compute their portion of the computation based on their processor IDs. Synchronization only needs to be inserted where inter-processor data dependences actually occur; however, processors may be forced to remain idle for long periods of time during sequential portions of the computation. The compiler also requires extensive analysis that may limit the range of applications supported.

Figure 2 illustrates the difference between the fork-join and SPMD programming models. In the figure, time proceeds from top to bottom. The tall vertical lines represent threads, the short lines crossing between them represent synchronization. In the fork-join model, worker threads are awakened by the master and synchronized with a barrier after each parallel region. In the SPMD model, threads are always active and can synchronize in a number of ways, including barriers, one-to-all, nearest-neighbor, and pipelining.

Our compiler attempts to blend the advantages of both approaches through a hybrid strategy first suggested by Cytron *et al.* [11]. Sequential parts of the program are executed by a single master thread, as in traditional shared-memory compilers. Parallel loops, however, are combined to form larger parallel regions that can be treated as small SPMD programs. Within each SPMD region, the compiler can improve upon traditional compilers by exploiting less expensive forms of synchronization. By applying communication analysis, barrier synchronization may be also be eliminated outright.

### 2.2 Scalar and Array Dependence Analysis

Determining exactly which barriers can be safely eliminated in an SPMD region is a non-trivial task. In the simplest case, without analysis two adjacent parallel loops may be merged into a single region by separating them with a barrier, as shown in first example in Figure 3. The number of barriers executed is unchanged after merging, but the number of parallel regions is reduced, eliminating start up overhead such as broadcasts to the worker threads.

Traditional parallelizing compilers perform scalar data-flow and array data-dependence analysis to track data access patterns [23]. Scalar data-flow analysis treats each variable as a monolithic object; array data-dependence analysis is more precise in that it tries to determine which array elements are accessed. If scalar or array dependence analysis can determine variables or memory locations accessed in one region of the program are not accessed in another region, the barrier separating the two regions may be eliminated.

To see how scalar and array dependence analysis may be applied, consider Figure 3. In the second program the first loop defines A while the second loop defines B and references C. Scalar analysis is sufficient to determine neither loop defines values used by the other loop, allowing the interleaving barrier to be eliminated. In comparison, for the third program in Figure 3 array data-dependence analysis is needed to determine that the two loops define and access disjoint elements of the array A. Scalar analysis is insufficient because there are accesses to A in both loops.

Dependence analysis, however, may not be sufficiently precise to eliminate barrier synchronization. For instance, consider the examples in Figure 4. The first program contains two adjacent parallel loops, with the second loop accessing values of array A defined in the first loop nest. In the second program, there is a dependence carried on the outer J loop caused by repeated accesses to the same elements of array A between loop iterations. A barrier is needed following the I loops in both programs because dependence analysis cannot eliminate the possibility of data races between definitions and uses of A. Though dependence analysis fails for these two examples, synchronization costs may be reduced further through communication analysis.

### 2.3 Communication Analysis

*Communication analysis* is a set of techniques used to track the flow of data between processors. It was first developed in compilers for

{ No Analysis }

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

{ Scalar Analysis }

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

{ Dependence Analysis }

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

Figure 3: Precision of Analysis

{ Adjacent Parallel Loops }

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

{ Nested Parallel Loop }

```
DO J = 1,M
   DOALL I = 1,N
      A(I) = F(A(I))
   ENDDO
ENDDO
   ⇓
broadcast
DO J = 1,M
   DO I = LB1,UB1
      A(I) = F(A(I))
   ENDDO
ENDDO
barrier
```

Figure 4: Consistent Computation Partitioning

{ Nearest-Neighbor }

```
DOALL I = 1,N
   A(I) =
ENDDO
DOALL J = 1,N
   B(J) = A(J-1) + A(J+1)
ENDDO
   ⇓
broadcast
DO I = LB,UB
   A(I) =
ENDDO
sync_with_neighbors
DO J = LU,UB
   B(J) = A(J-1) + A(J+1)
ENDDO
barrier
```

{ Producer-Consumer }

```
DO I = 1,N
   DOALL J = I+1,N
      A(J) = F(A(I))
   ENDDO
ENDDO
   ⇓
broadcast
DO I = 1,N
   IF (I > 1)
      wait_counter
   DO J = LU,UB
      A(J) = F(A(I))
      IF (J == I+1)
         increment_counter
   ENDDO
ENDDO
barrier
```

Figure 5: Replacing Barriers

distributed-memory machines in order to identify where messages must be inserted into a program to access data on other processors [18]. We discovered communication analysis to be quite useful for reducing synchronization on shared-memory machines as well. If it can identify the producers and consumers of all data shared between two regions to be identical (*i.e.,* the same processor), then data movement is local and no synchronization is necessary.

A *computation partition* describes how work is partitioned across processors. It is usually expressed by how iterations of parallel loops are assigned to processors. In order to fully take advantage of communication analysis, the computation partition must be computed at compile-time in a globally consistent manner to reduce communication. Otherwise communication analysis will simply identify sources of data movement, instead of opportunities for eliminating synchronization overhead.

Computation-oriented approaches for selecting computation partitions that reduce communication are known as *affinity scheduling*, where loop iterations accessing the same data are assigned to the same processor where possible [6, 13, 21]. The computation partition is usually determined locally through simple heuristics. Alternative data-oriented approaches are based on *automatic data decomposition* techniques; they strive for a global partition to co-locate data and computation. The computation partition may be computed simultaneously with the data partition [5] or derived from the data decomposition [7, 14] using the *owner-computes* rule [18], where each processor is assigned work that results in values for its local data. For this paper we assume the compiler partitions computation using global automatic data decomposition techniques, since such methods have been extensively studied and are more precise.

Once consistent compile-time computation partitions are selected, communication analysis can be used to eliminate synchronization for the code in Figure 4. In the first program, a barrier is not needed between the I and J loops if the computation is partitioned so that each processor receives the iterations from both loops that accesses the same elements of A. In the second example, the barrier can be moved outside of the J loop because the compiler detects that the same iterations of the inner I loop are assigned to each processor on each instance of the outer J loop. In both cases, communication analysis succeeds where scalar and array data dependence analysis fails because it is able to take the consistent compile-time computation partition into account.

Finally, by identifying the producer and consumer of inter-processor data movement, communication analysis is able to enforce necessary synchronization with inexpensive pair-wise synchronization instead of global barriers. For instance, consider the two examples in Figure 5. We assume that the computation is partitioned so that processors receive blocks of contiguous loop iterations. In the first case communication analysis determines that each processor needs only elements of A from neighboring processors. In the second case the compiler finds the J loop consumes one element of A defined on the previous iteration of the I loop. By inserting synchronization in the form of counters, the compiler can enforce the produce-consumer relationship, eliminating the need for barriers following the parallel J loop.

## 3 Synchronization Optimization Algorithm

Now that we have presented an overview how synchronization may be reduced, we describe compiler algorithms to eliminate barrier synchronization. Our optimization algorithm is composed of three major components: constructing SPMD regions, eliminating unneeded barriers, and replacing needed barriers with less expensive forms of pair-wise synchronization based on counters.

### 3.1 Constructing SPMD Regions

The first step for the compiler is to construct SPMD regions, setting the stage for applying communication analysis. The goal is to create SPMD regions that are sufficiently large to enable optimizations, but does not include excessive amounts of sequential, replicated, or guarded code. The basic construction algorithm is very simple. Initially, each parallel loop in the program is placed in its own SPMD region. Larger SPMD regions may be constructed by following two rules:

- Adjacent SPMD regions may be merged into a single SPMD region by inserting a barrier between the two original regions.
- If the all statements in the loop body are in a single SPMD region, the region may be expanded to include the loop by inserting a barrier as the last statement in the loop body.

The rules are applied repeatedly until no further expansion of SPMD regions is possible. Intuitively, each SPMD region grows until it encounters sequential computation that the compiler determines will definitely require barrier synchronization.

The SPMD regions constructed by the basic algorithm are conservative; no computations are replicated and processor utilization in the region is maximized because no sequential computation is enclosed. Larger SPMD regions may be created by allowing two additional types of of statements to be included:

- *Replicated computations*—statements whose execution can be replicated across processors. The most common case involves assignments to *privatizable* variables. These assignments may need to be *finalized* following the SPMD region [15, 27].
- *Guarded computations*—statements that must be protected by explicit guard expressions to ensure they are executed by the appropriate processor(s) [18]. Synchronization may also be needed for the guarded computation.

Putting replicated computations in an SPMD region increases the amount of computation that must be performed. Putting guarded computations in an SPMD region incurs the cost of evaluating guard expressions and also results in lower processor utilization. As a result, these statements should be put in an SPMD region only when necessary to enable merging, such as when they lie between two parallel loops or are the only statements in a loop body not in an SPMD region. The prototype implementation is conservative in that it only places replicated computations in SPMD regions.

To see how the SPMD region construction algorithm works, consider the fragment of *tomcatv* displayed in Figure 6. Both of the the parallel I loops are initially placed in a separate SPMD region. As the entire body of the J loop is in an SPMD region, the region grows to also include the J loop, inserting a barrier as the last statement in the body of the loop. Finally, the SPMD regions containing the first I loop and the J loops are merged into a single region, with a barrier inserted between the two loops. Note how the SPMD region construction algorithm inserts explicit barriers to take the place of implicit barriers following parallel loops.

### 3.2 Eliminating Barrier Synchronization

After SPMD regions are constructed, each SPMD region contains groups of statements separated by barriers. A barrier separating two groups can be eliminated if the compiler can determine that there are no race conditions between the code separated by the barrier. We describe how the compiler tests for communication between two references, then show how it is used in the barrier elimination algorithm.

#### 3.2.1 Testing For Communication

The compiler can test for inter-processor communication at several levels of precision. As described earlier, scalar data-flow and array data-dependence analysis are sufficient in some cases, but communication analysis is more precise. Our compiler first eliminates potential sources of communication using scalar and array dependence analysis, then applies communication analysis.

In our compiler, local definitions and nonlocal accesses are both represented by systems of symbolic linear inequalities [1]. Using systems of inequalities provides greater flexibility and precision. They are particularly suited for synchronization optimizations since we need to know 1) whether communication is present, and 2) the identity of the producer and consumer processors. The actual array elements transferred between processors is not needed, avoiding problems with precise merging of systems of inequalities.

To test two references for communication, symbolic linear inequalities are built up consisting of array subscript expressions for both references and the computation partition. When using the owner-computes rule, the computation partition may be derived by composing the inequalities representing the array, loops, data decomposition of the array, and the array subscript expression for the left-hand side (LHS) reference. We add the constraint that producer and consumer processors are different. For the right-hand side (RHS) reference, we also rename the processor variables and some or all of the loop index variables. This renaming allows us to distinguish between the producer and consumer processors and the loop iterations accessing the communicated data.

We can also add additional constraints to specify when the communication takes place. When testing for *loop-independent* (*i.e.,* on the same loop iteration) communication between adjacent groups of statements, we hold the outer loop variables constant as symbolics, renaming only loop index variables for deeper loops. When testing for *loop-carried* communication (*i.e.,* across different loop iterations) of statements nested in a loop, we also rename the loop index variable for the loop and add the constraint that its value differs on the producer and consumer processors for the iterations accessing the data being communicated.

Before attempting to solve the system of symbolic linear inequalities, we sort the variables into the following scan order: symbolics, processors, loop index variables, and array indices. We then determine whether the resulting system of inequalities is consistent by scanning the system using Fourier-Motzkin elimination [2, 3]. If a solution to the system exists, inter-processor data movement takes place and must be synchronized. The solution calculated by the scan can then be used to calculate the identities of the communication processors, the nonlocal data accessed, and the loop iterations where the data is defined [1].

#### 3.2.2 Elimination Algorithm

Given a test for interprocessor communication, the compiler uses the following greedy algorithm to eliminate barriers in an SPMD region:

1. Starting at the head of the deepest loop containing a barrier, construct a list of variable references and definitions for each group of statements separated by a barrier. Combine references to the same variable when precise.
2. If two groups of statements are separated by a barrier at the same loop level, compare the list of variables defined in the first group against the list of variables references in the second group, and vice versa. If not using the owner-computes rule to partition computation, also compare the list of variables defined in the first group against those defined in

the second group. These correspond to true, anti, and output dependences between the two regions [23].

3. Test for loop-independent communication between the two groups at the current loop nesting level. If communication does not exist then the barrier may be eliminated. Combine the two group of statements into a single group, merging the lists of variable definitions and references. Repeat from beginning.

4. If loop-independent communication does take place, leave barrier in place and continue to the next barrier. Repeat from beginning, starting with the group of statements after the barrier.

5. If all barriers at the current loop level or deeper have been eliminated except for the barrier at the end of loop body, compare all variables defined in loop against list of variables references. Test for *loop-carried* communication at the enclosing loop. If no such communication takes place, the barrier may be eliminated. Propagate lists of variable definitions and references upward to the loop.

6. After loop body is analyzed, repeat from beginning, starting with with statements at the level of the enclosing loop. Do not eliminate barriers separating loops containing barriers. Loops not containing barriers may be treated in the standard manner as large compound statements using the variable lists propagated in the previous step.

The algorithm works its way through SPMD regions in a bottom-up manner, eliminating barriers in the order they are encountered. Assuming the test for communication is precise, if all barriers separating adjacent groups of statements can be eliminated the algorithm will do so. The algorithm will also eliminates the maximum number of barriers when only a small numbers of barriers are required. However, the algorithm may not achieve the optimal result if the number of statement groups and barriers required are both large. When dealing with barriers in loop nests, the algorithm will eliminate all barriers from a nest if it is possible. However, if the loop contains one or more required barriers, the algorithm is not always able to detect the barrier at the end of the loop body may be eliminated.

### 3.2.3 Tomcatv Example

Consider how the barrier elimination algorithm works for the fragment of *tomcatv* displayed in Figure 6. Recall that barriers were inserted after both parallel I loops while constructing the SPMD region. The barrier elimination algorithm would begin by attempting to eliminate the deeper nested barrier. Statements in the body of the second I loop are traversed to build up a list of definitions and references to RX and RY. Dependence analysis cannot disprove communication because data dependences exist, carried on the J loop.

Communication analysis is then invoked. The compiler finds that the automatic data decomposition phase has partitioned RX and RY into blocks of contiguous rows. Applying the owner-computes rule, the compiler assigns each processor the chunk of I loop iterations that compute values for its local data. Systems of linear inequalities are built by combining the computation partition and the definition and use of each array. We also constrain values of J, the outer loop, to differ for the producer and consumer processors, but allow the value of I to range freely between its bounds. Figure 7 shows an example set of inequalities constructed to test for communication between the references RX(I,J) and RX(I,J+1). Applying Fourier-Motzkin elimination to these systems of equations, we find that no solution exists. All communication is localized since the producer and consumer processors are identical. We can thus eliminate the barrier nested in the J loop. The lists of variables are propagated upwards to the outer J loop.

```
DOALL I = 2,N-1
   RX(I,M) = RX(I,M)*D(I,M)
   RY(I,M) = RY(I,M)*D(I,M)
ENDDO
{ implicit barrier }
DO J = M-1,1,-1
   DOALL I = 2,N-1
      RX(I,J) = 𝓕(RX(I,J),RX(I,J+1))
      RY(I,J) = 𝓕(RY(I,J),RY(I,J+1))
   ENDDO
   { implicit barrier }
ENDDO
```

Figure 6: Tomcatv Fragment

| | |
|---|---|
| ARRAY | $1 \le rx_1 \le n$<br>$1 \le rx_2 \le n$ |
| LOOPS | $1 \le j \le m-1$<br>$2 \le i \le n-1$ |
| LHS | $rx_1 = i$<br>$rx_2 = j$ |
| RHS | $rx_1 = i'$<br>$rx_2 = j' + 1$ |
| DATA DECOMP (BLOCK, *) | $b * nprocs = n$<br>$b * p \le rx_1 < b * (p+1)$ |
| COMP PARTITION | ARRAY $\wedge$ LOOPS $\wedge$ DATA DECOMP $\wedge$ LHS (PROJECT AWAY $rx_1, rx_2$) |
| COMMUNICATION | COMP PARTITION $\wedge$ LHS $\wedge$ RHS $\wedge$ $p < p'$<br>COMP PARTITION $\wedge$ LHS $\wedge$ RHS $\wedge$ $p > p'$ |

Figure 7: Inequalities for References to RX in Tomcatv

A similar process takes place to determine whether the barrier following the first I loop may be eliminated. This time we compare two different pairs of definition and reference lists against each other, and allow values both I and J to range freely between their loop bounds. Applying Fourier-Motzkin elimination we find that the new systems are also inconsistent, allowing the barrier to be eliminated. As a result, communication analysis determines this entire portion of *tomcatv* can be executed without synchronization.

## 3.3 Replacing Barriers with Counters

Even after communication analysis identifies inter-processor data movement, it may still be possible to lower synchronization cost by replacing barriers with less expensive forms of point-to-point synchronization such as counters. Counters are similar to event synchronization [20] but are more flexible; they enforce synchronization constraints as follows. Processors defining (producing) values can increment a counter, and processors accessing (consuming) the values wait until the counter is incremented to the proper value. Counters are desirable alternatives for two reasons. They incur less overhead than barriers, and only processors actually consuming data must wait for a counter.

Excessive use of counters is not efficient, since a single barrier can take the place of a large number of counters. Fortunately, our compiler does not need to insert counters for each nonlocal access or loop iteration. Instead, it takes advantage of compile-time computation partitioning to insert at most one counter between each pair of processors. Counters are especially useful for supporting many regular communication patterns found in scientific applications.

Examples include one-to-all, nearest-neighbor, and pipelining as pictured in Figure 2.

The algorithm for used for replacing barriers is a simple extension to the barrier elimination algorithm. When the compiler detects communication between statements separated by a barrier, it normally moves on. To determine whether the barrier may be replaced, the compiler tests the communication. If it fits simple patterns allowing efficient synchronization with counters, the barrier is replaced with the appropriate counter. Statements separated by barrier are merged into same group before continuing.

### 3.3.1 Replacement Test

To detect simple communication patterns that may be efficiently enforced with counters, the compiler examines the systems of inequalities corresponding to the data communicated between processors. Recall that these inequalities are produced by the barrier elimination test. The identity of the producer and consumer processors may be read directly from each system and used to identify the following communication patterns:

- *Nearest-neighbor* occurs when the identity of the producer is equal that of the consumer, plus or minus a constant. This pattern corresponds to a shift on a message-passing machine. The sign of the constant is the direction of the message.
- *One-to-many* occurs when the producer is a symbolic or actual constant. This pattern corresponds to a broadcast.
- *Many-to-one* occurs when the consumer is a symbolic or actual constant. This pattern corresponds to a gather.

To test whether a barrier may be replaced, the compiler scans the list of data communicated across the barrier for communication patterns. The barrier is replaced only if all the data sections can be classified as a simple pattern or combination of several patterns.

When the barrier separates adjacent groups of statements, it is replaced by calls to the *increment_counter* and *wait_counter* routines from the run-time system. Guards are placed around each call to ensure the producers increment the counter while consumers wait for counters to be incremented before proceeding. For barriers at the end of the loop body, guarded calls to *increment_counter* are inserted at the bottom of the loop body, while guarded calls to *wait_counter* are inserted at the head of the loop. This arrangement prevents processors from waiting for counter increments unless they actually execute the iteration requiring data.

To reduce idle processors, the calls to *increment_counter* may be moved back in the code, closer to the computation producing the data to be communicated. Similarly, the calls to *wait_counter* may be moved forward in the code, closer to the computation that actually consumes the nonlocal data. When loop nests are encountered during this code motion, the inequalities representing the nonlocal data may contain information on which loop iterations produced the data. If only the first few iterations produce nonlocal data, it may be profitable to move the call to *increment_counter* inside the loop, to be executed as soon as the last value needed by other processors is produced.

### 3.3.2 LU Decomposition Example

To see how barrier replacement works, consider the LU Decomposition example in Figure 8. The entire loop nest is put into a single SPMD region, with a barrier inserted after the parallel J loop. Automatic data decomposition selects a column-cyclic distribution for the array A to improve locality and load balance. Each processor is thus assigned iterations of J in a round-robin manner. Communication analysis determines that there is communication across the

```
DO K = 1,N
  DOALL J = K+1,N
    A(K,J) = A(K,J) / A(K,K)
    DO I = K+1,N
      A(I,J) = A(I,J) - A(K,J) * A(I,K)
    ENDDO
  ENDDO
ENDDO

  ⇓

broadcast
SENDER = 0
DO K = 1,N
  IF (K ≠ 1) AND (MYID ≠ SENDER) THEN
    wait_counter
  SENDER = MOD(K, NPROCS)
  COUNT = COUNT + 1
  DO J = LB, UB
    A(K,J) = A(K,J) / A(K,K)
    DO I = K+1,N
      A(I,J) = A(I,J) - A(K,J) * A(I,K)
    ENDDO
    IF (J == K+1) AND (MYID == SENDER) THEN
      increment_counter
  ENDDO
ENDDO
barrier
```

Figure 8: LU Decomposition

| | |
|---|---|
| ARRAY | $1 \le a_1 \le n$<br>$1 \le a_2 \le n$ |
| LOOPS | $1 \le k \le n$<br>$k+1 \le j \le n$<br>$k+1 \le i \le n$ |
| LHS | $a_1 = i$<br>$a_2 = j$ |
| RHS | $a_1 = i'$<br>$a_2 = k'$ |
| DATA DECOMP (*, CYCLIC) | $a_2 = vp$<br>$(vp \bmod nprocs = p)$ |
| COMP DECOMP | ARRAY $\wedge$ LOOPS $\wedge$ DATA DECOMP $\wedge$ LHS<br>(PROJECT AWAY $a_1, a_2$) |
| COMMUNICATION | COMP DECOMP $\wedge$ LHS $\wedge$ RHS $\wedge$ $p < p'$<br>COMP DECOMP $\wedge$ LHS $\wedge$ RHS $\wedge$ $p > p'$ |
| SCAN OUTPUT | FOR $k$ = 1 TO $n$<br>$k'$ = $k$+1<br>$vp$ = $k'$<br>FOR $vp'$ = $k$+2 TO $n$<br>$j$ = $k'$<br>$j'$ = $vp'$<br>FOR $i$ = $k$+2 TO $n$<br>$i'$ = $i$<br>$a'_1$ = $k'$<br>$a'_2$ = $i$<br>$a_1$ = $j'$<br>$a_2$ = $a'_2$ |

Figure 9: Inequalities for References to A in LU Decomposition

iterations of K, the outermost loop, between the *lhs* reference A(I,J) and the *rhs* reference A(I,K). The inequalities constructed are shown in Figure 9.

Note that virtual processors ($vp$) are used to represent cyclic distributions. The logical processor ID is calculated using the modulo function; this relation is implicit and is not part of the system of inequalities. Because the compiler is attempting to calculate communication across iterations of the outer K loop, its indices are scanned before the processors. The scan output is consistent, so the barrier cannot be eliminated outright.

The barrier replacement algorithm is now invoked, with the results of the scan as input. It discovers that the communication has a one-to-many pattern, since the processor producing the data has a symbolic constant value ($vp = k'$). The barrier may thus be replaced by a counter operation. In addition, inequalities show that the value of J also has a constant value ($j = k', k' = k + 1$), indicating all the values communicated are defined on the iteration J=K+1. When counters are introduced in place of the barrier, the compiler thus arranges to have the counter incremented as soon as the producer processor completes the J loop iteration with value equal to K+1. The resulting program, shown in Figure 8, exhibits good overlap of communication and synchronization. By incrementing the counter as soon as the first column of the submatrix is reduced, other processors do not have to wait for the producer processor to complete all of its work for the current iteration of the outer K loop.

## 4 Experimental Results

To evaluate the effectiveness of the techniques described in this paper, we have implemented them in the Stanford SUIF parallelizing compiler [28]. The inputs to the compiler are sequential Fortran or C programs. Data-flow, symbolic, and reduction analysis is performed, followed by a parallelism and locality analysis phase identifies and optimizes loop-level parallelism [29]. A global decomposition pass attempts to find a mapping of the computation and data that maximizes inter-processor locality [5]. Synchronization optimizations are applied next, followed by a data-reorganization pass to reduce cache conflicts [4]. A code-generation pass then generates the parallel (SPMD) program, converting bodies of parallel loops and parallel regions into parallel procedures. The output is translated to C, compiled by the native C compiler, and linked with a run-time library that provides support for thread creation, synchronization (barriers, locks and counters) and reductions. The run-time library supports the Stanford DASH, KSR-1, and SGI Challenge multiprocessors.

The applications used in our study consist of standard scientific codes taken from the PERFECT, SPEC, and NAS benchmark suites; their characteristics are listed in Table 1. We measured the effectiveness of optimizations both statically at compile time and dynamically at run time. Three applications (*ora, buk, and embar*) were excluded from our study because the compiler was not able to automatically detect appreciable amounts of parallelism. Two applications (*SDS, wave5*) were excluded from the run-time evaluation because of difficulties in executing the compiler output.

### 4.1 Static Measures

Table 2 presents compile-time statistics on the effect of the synchronization optimizations. We present optimization results for each program, and also summarize in two ways. For *Total* we treat all the statistics as if they came from large program, adding up raw numbers before calculating percentages. For *Average* we compute average statistics over all programs. By providing both sets of summaries we account for both the effect of large programs and many small programs.

| Program | Lines | Description |
|---|---|---|
| PERFECT BENCHMARKS | | |
| APS (adm ) | 6105 | Pseudospectral Air Pollution |
| CSS (spice) | 18521 | Circuit Simulation |
| LGS (qcd ) | 2102 | Quantum Chromodynamics |
| LWS (mdg) | 1238 | Molecular Dynamics of Water |
| MTS (track) | 3735 | Missile Tracking |
| OCS (ocean) | 4343 | 2D Ocean Simulation |
| NAS (bdna) | 3980 | Molecular Dynamics of DNA |
| SDS (dyfesm) | 7608 | Structural Dynamics |
| SMS (md3d) | 2812 | Depth Migration |
| SRS (arc2d) | 3965 | 2D Fluid Flow Solver |
| TFS (flo52) | 3285 | Transonic Inviscid Flow |
| TIS (trfd) | 485 | Electron Integral Transform |
| WSS (spec77) | 3885 | Weather Simulation |
| SPEC92 BENCHMARKS | | |
| doduc | 5334 | Thermohydraulical Modelization |
| fpppp | 2718 | 2 Electron Integral Derivative |
| hydro2d | 4392 | Navier-Stokes |
| mdljdp2 | 4316 | Molecular Dynamics |
| mdljsp2 | 3885 | Molecular Dynamics |
| nasa7 | 1105 | NASA Ames Fortran Kernels |
| su2cor | 2514 | Quantum Physics |
| swm256 | 487 | Shallow Water Model |
| tomcatv | 195 | Mesh Generation |
| wave5 | 7628 | Maxwell's Equations |
| NAS BENCHMARKS | | |
| appbt | 4457 | Block-Tridiagonal PDE Solver |
| applu | 3417 | Lower-Upper triangular PDE Solver |
| appsp | 3516 | Scalar-Pentadiagonal PDE Solver |
| cgm | 855 | Sparse Conjugate Gradient |
| fftpde | 773 | 2D Fast Fourier Transform |
| mgrid | 676 | Multigrid Solver |
| MISCELLANEOUS | | |
| adi256 | 131 | 2D ADI Integration |
| erle64 | 580 | 3D Tridiagonal Solver |
| linpackd | 791 | Gaussian Elimination with Pivoting |
| lu256 | 59 | LU Decomposition |
| shallow | 167 | Shallow Water Model |
| simple | 1346 | 2D Lagrangian Hydrodynamics |

Table 1: Scientific Applications in Study

The two main metrics we use to evaluate the effectiveness of our synchronization optimizations are the number of SPMD regions and barriers. Reducing the number of SPMD regions reduces the overhead associated with starting a parallel task, including the cost of broadcasting the function pointer. The advantages of eliminating barriers has been discussed. The first two columns, *Base* and *Opt*, contrasts the number of parallel SPMD regions in the base and optimized versions of each program. In the base (unoptimized) program, each SPMD region represents one parallel loop. We see that by applying synchronization optimizations the compiler can reduce the number of SPMD regions for almost all programs, sometimes significantly.

The next four columns measures optimizations on adjacent parallel loops. *% Regions Merged* refers to the percent of adjacent parallel loops that have been merged into a single SPMD region. *% Barriers Elim (Dep)* shows the percent of barriers between adjacent loops eliminated using scalar or array dependence analysis. *% Elim (Comm)* and *% Repl (Comm)* list the total percent of barriers eliminated or replaced with counters using communication analysis.

| Program | SPMD Regions Base | SPMD Regions Opt. | Adjacent Loops % Regions Merged | Adjacent Loops % Barriers Elim (Dep) | Adjacent Loops % Barriers Elim (Comm) | Adjacent Loops % Barriers Repl (Comm) | Nested Loops % Regions Merged | Nested Loops % Barriers Elim (Comm) | Nested Loops % Barriers Repl (Comm) | Largest SPMD Region # Loop Nests | Largest SPMD Region Barriers Inserted |
|---|---|---|---|---|---|---|---|---|---|---|---|
| PERFECT BENCHMARKS | | | | | | | | | | | |
| APS | 120 | 119 | 1 | 0 | 1 | 0 | 11 | 10 | 0 | 2 | 0 |
| CSS | 34 | 34 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| LGS | 18 | 17 | 6 | 6 | 6 | 0 | 6 | 6 | 0 | 2 | 0 |
| LWS | 36 | 35 | 3 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 1 |
| MTS | 36 | 31 | 14 | 11 | 11 | 0 | 11 | 11 | 0 | 3 | 0 |
| NAS | 87 | 75 | 14 | 6 | 6 | 0 | 1 | 0 | 0 | 4 | 3 |
| OCS | 42 | 36 | 14 | 5 | 10 | 2 | 19 | 14 | 0 | 2 | 0 |
| SDS | 64 | 58 | 9 | 6 | 9 | 0 | 8 | 6 | 0 | 2 | 0 |
| SMS | 53 | 42 | 21 | 13 | 13 | 0 | 55 | 32 | 0 | 2 | 0 |
| SRS | 116 | 46 | 60 | 12 | 21 | 0 | 16 | 13 | 0 | 16 | 13 |
| TFS | 61 | 43 | 30 | 15 | 15 | 2 | 30 | 26 | 0 | 5 | 2 |
| TIS | 23 | 19 | 17 | 9 | 9 | 0 | 35 | 9 | 0 | 3 | 1 |
| WSS | 188 | 114 | 39 | 21 | 31 | 1 | 13 | 7 | 0 | 9 | 2 |
| SPEC92 BENCHMARKS | | | | | | | | | | | |
| doduc | 193 | 67 | 65 | 65 | 65 | 0 | 1 | 1 | 0 | 27 | 0 |
| fpppp | 10 | 9 | 10 | 10 | 10 | 0 | 0 | 0 | 0 | 2 | 0 |
| hydro2d | 86 | 60 | 30 | 20 | 20 | 2 | 15 | 15 | 0 | 5 | 1 |
| mdljdp2 | 15 | 14 | 7 | 0 | 7 | 0 | 0 | 0 | 0 | 2 | 0 |
| mdljsp2 | 14 | 13 | 7 | 0 | 7 | 0 | 0 | 0 | 0 | 2 | 0 |
| nasa7 | 24 | 16 | 33 | 4 | 33 | 0 | 12 | 12 | 0 | 7 | 0 |
| su2cor | 26 | 25 | 4 | 4 | 0 | 0 | 4 | 4 | 0 | 2 | 1 |
| swm256 | 12 | 10 | 17 | 0 | 0 | 17 | 0 | 0 | 0 | 3 | 0 |
| tomcatv | 9 | 5 | 44 | 33 | 33 | 0 | 56 | 56 | 0 | 5 | 1 |
| wave5 | 173 | 156 | 10 | 1 | 6 | 1 | 18 | 17 | 0 | 3 | 0 |
| NAS BENCHMARKS | | | | | | | | | | | |
| appbt | 54 | 40 | 26 | 2 | 9 | 11 | 6 | 6 | 0 | 4 | 0 |
| applu | 45 | 36 | 20 | 4 | 7 | 13 | 9 | 9 | 0 | 4 | 0 |
| appsp | 58 | 44 | 24 | 2 | 3 | 10 | 5 | 5 | 0 | 4 | 0 |
| cgm | 11 | 10 | 9 | 9 | 9 | 0 | 0 | 0 | 0 | 2 | 0 |
| fftpde | 20 | 18 | 10 | 10 | 10 | 0 | 10 | 10 | 0 | 2 | 0 |
| mgrid | 14 | 10 | 29 | 14 | 29 | 0 | 21 | 21 | 0 | 2 | 0 |
| MISCELLANEOUS | | | | | | | | | | | |
| adi256 | 9 | 2 | 78 | 44 | 67 | 0 | 33 | 22 | 0 | 6 | 1 |
| erle64 | 129 | 45 | 65 | 48 | 59 | 0 | 61 | 61 | 0 | 7 | 0 |
| lu256 | 2 | 2 | 0 | 0 | 0 | 0 | 50 | 0 | 50 | 1 | 0 |
| linpackd | 22 | 21 | 5 | 5 | 5 | 0 | 9 | 9 | 0 | 2 | 0 |
| shallow | 9 | 4 | 56 | 0 | 0 | 56 | 0 | 0 | 0 | 3 | 0 |
| simple | 26 | 20 | 23 | 15 | 12 | 8 | 31 | 31 | 0 | 3 | 1 |
| SUMMARIES | | | | | | | | | | | |
| *Total* | 1839 | 1296 | 29.53 | 17.29 | 21.21 | 1.90 | 15.93 | 13.54 | 0.05 | | |
| *Average* | 52.54 | 37.03 | 22.83 | 11.27 | 14.85 | 3.52 | 15.57 | 11.81 | 1.43 | 4.29 | 0.77 |

Table 2: Optimization Results: Static Counts

| Program | SPMD Regions Base | SPMD Regions % Elim | Barriers % Elim (Dep) | Barriers % Elim (Comm) | Barriers % Repl (Comm) | Granularity Base (msec) | Granularity % Incr (Dep) | Granularity % Incr (Comm) |
|---|---|---|---|---|---|---|---|---|
| PERFECT BENCHMARKS | | | | | | | | |
| APS | 2738515 | 2 | 0 | 0 | 0 | 0.08 | 0 | 0 |
| CSS | 138361 | 0 | 0 | 0 | 0 | 0.02 | 0 | 0 |
| LGS | 1642498 | 12 | 0 | 12 | 0 | 0.03 | 0 | 14 |
| LWS | 49609915 | 0 | 0 | 0 | 0 | 0.04 | 0 | 0 |
| MTS | 626688 | 24 | 20 | 20 | 0 | 0.03 | 25 | 25 |
| NAS | 22813 | 33 | 0 | 0 | 0 | 9.28 | 0 | 0 |
| OCS | 323412 | 16 | 0 | 15 | 0 | 1.21 | 0 | 18 |
| SMS | 2082960 | 97 | 0 | 41 | 0 | 0.79 | 0 | 70 |
| SRS | 251924 | 98 | 0 | 87 | 0 | 8.81 | 0 | 643 |
| TFS | 410119 | 93 | -7 | 87 | 0 | 1.46 | -7 | 672 |
| TIS | 7116557 | 2 | 0 | 2 | 0 | 0.05 | 0 | 2 |
| WSS | 5584322 | 76 | 12 | 22 | 1 | 0.12 | 14 | 30 |
| SPEC92 BENCHMARKS | | | | | | | | |
| doduc | 34636 | 8 | 1 | 8 | 0 | 0.03 | 1 | 9 |
| fpppp | 61474 | 0 | 0 | 0 | 0 | 0.02 | 0 | 0 |
| hydro2d | 82593 | 93 | 0 | 90 | 1 | 0.16 | 0 | 972 |
| mdljdp2 | 16507 | 0 | 0 | 0 | 0 | 0.00 | 0 | 0 |
| mdljsp2 | 16507 | 0 | 0 | 0 | 0 | 0.00 | 0 | 0 |
| nasa7 | 177233 | 49 | 47 | 49 | 0 | 10.48 | 90 | 97 |
| su2cor | 5715 | 0 | 0 | 0 | 0 | 9.40 | 0 | 0 |
| swm256 | 362 | 1 | 0 | 0 | 1 | 348.57 | 0 | 1 |
| tomcatv | 102256 | 100 | 0 | 100 | 0 | 4.38 | 0 | 33648 |
| NAS BENCHMARKS | | | | | | | | |
| appbt | 680041 | 61 | 0 | 30 | 3 | 0.06 | 0 | 48 |
| applu | 1203381 | 6 | 4 | 4 | 1 | 0.09 | 4 | 6 |
| appsp | 250081 | 14 | 0 | 1 | 12 | 0.71 | 0 | 16 |
| cgm | 2358 | 0 | 0 | 0 | 0 | 1.54 | 0 | 0 |
| fftpde | 344095 | 25 | 0 | 25 | 0 | 0.41 | 0 | 33 |
| mgrid | 1635 | 90 | -8 | 90 | 0 | 16.89 | -8 | 874 |
| MISCELLANEOUS | | | | | | | | |
| adi256 | 2059 | 100 | 0 | 99 | 0 | 3.73 | 0 | 18627 |
| erle64 | 9105 | 100 | -1 | 100 | 0 | 0.50 | -1 | 53465 |
| linpackd | 1014520 | 0 | 0 | 0 | 0 | 0.45 | 0 | 0 |
| lu256 | 257 | 99 | 0 | 0 | 99 | 13.38 | 0 | 8500 |
| shallow | 9 | 56 | 0 | 0 | 50 | 232.80 | 0 | 100 |
| simple | 1887201 | 76 | 0 | 76 | 0 | 2.48 | 0 | 324 |
| SUMMARIES | | | | | | | | |
| *Total* | 76440109 | 12.86 | 1.19 | 6.89 | 0.16 | 0.21 | 1.21 | 7.58 |
| *Average* | 2316367 | 40.27 | 2.04 | 29.04 | 5.07 | 20.24 | 3.57 | 3581 |

Table 3: Optimization Results: Dynamic Counts

We see that about 11% of barriers can be eliminated without communication analysis. Though we do not present the actual numbers, scalar analysis is sufficient for eliminating almost all these barriers. For all the programs in the study, data dependence analysis could only eliminate four barriers (in *SRS, TFS*, and *hydro2d*) not eliminated by scalar analysis. In contrast, communication analysis eliminates on average an additional 3.6% of barriers separating adjacent parallel loops, and can replace another 3.5% more with nearest-neighbor synchronization.

The next three columns measures optimizations on parallel loops nested in sequential loops. *% Regions Merged* refers to the percent of times an SPMD region was expanded to include an outer sequential loop. *% Barriers Elim (Comm)* and *Repl (Comm)* list the percent of barriers in the nested loops eliminated or replaced with counters using communication analysis. We see the compiler is fairly successful, enlarging parallel SPMD regions around 15% of all parallel loops. Communication analysis is 76% successful in eliminating barriers resulting from such merges, resulting in an overall 11% elimination rate. It can only replace nested barriers with counters in one case, a one-to-all communication in LU decomposition.

The final two columns display statistics on the largest SPMD region created for each program. The columns *# Loop Nests* and *Barriers Inserted* show the size of the largest SPMD region created and the number of barriers inserted during the merge that could not be eliminated with communication analysis. For most programs, regions are fairly small. Even more interesting is that enclosed barriers can be usually eliminated, since only four programs have multiple barriers in their largest SPMD region. This fact implies that the greedy algorithm used in the compiler will usually find the minimal number of barriers needed to separate adjacent parallel loops.

### 4.2 Dynamic Measures

To better determine the performance impact we look at dynamic measures. Table 3 presents the dynamic count of parallel SPMD regions, barriers, and counters during program execution. The first two columns present information on SPMD regions. Because there is one barrier and parallel loop per SPMD region in the unoptimized program, the column *Base* presents the number of parallel loops and barriers executed in the original program. The *% Elim* column displays the percentage of parallel loop invocations that have been eliminated. We see that the average is quite high at 40%. The total value is lower at about 13%, showing that the compiler is less successful at merging SPMD regions for larger programs.

The next three columns display the percentage of barrier invocations that are eliminated or replaced by counters. The *% Elim (Dep)* column shows the percent of barriers eliminated by scalar and array data dependence analysis. *% Elim (Comm)* and *% Repl (Comm)* tally the total percentage of barriers eliminated or replaced using communication analysis. We see that scalar and dependence analysis is successful only for a few programs. Communication analysis is significantly more effective, on average eliminating 29% of all barriers and replacing another 5% with counters. However, the total reduction in barriers is only 7%, indicating there are several programs with high barrier overhead where synchronization optimizations were less effective.

On three programs (*mgrid, TFS, erle64*), the number of barriers increases noticeably when only dependence analysis is applied. This increase is a minor side effect of merging nested parallel loops. Recall that a barrier is inserted at the end of the loop body after merging with an outer sequential loop. If this barrier is not eliminated, it causes an extra barrier execution, since it is unnecessarily executed on the last iteration of the loop. However, the cost is likely to be small compared to the reduction in parallel invocation overhead.

The last column presents a new metric based on *granularity*, the size of a parallel region. The column *Base* presents the average granularity of parallel tasks for each program in milliseconds, as measured on a DECstation 5000/260. We calculated the average granularity by taking the total time spent in parallel regions of the program when the program is executed on one processor, divided by the number of barriers encountered. The columns *% Incr (Dep)* and % Incr (Comm) show the percentage increase in the average granularity of each parallel region. The results show communication analysis is significantly more effective than scalar and array dependence analysis in increasing the granularity of parallelism. With communication analysis, granularity is increased by a factor of 36 on average and by several orders of magnitude for some programs.

### 4.3 Speedups

Finally, we consider the effect on speedups for some programs on the Stanford DASH shared-memory multiprocessor. The baseline versions of each program produced by SUIF include both data and computation transformation optimizations [4]. The only difference between the optimized and base versions of each program is whether synchronization optimizations were applied. Clock routines were used to measure elapsed time between the start and end of the computation, excluding time spent by the program to initially create threads and initialize data arrays. Performance is measured as speedups over the best sequential version of each program. The results are shown in Figure 10.

Two programs significantly benefit from synchronization optimizations. *Tomcatv* contains a group of doubly nested loops generating a two-dimensional mesh. Several of the parallel loops are inner loops requiring nested barriers. With synchronization analysis, all nested barriers can be eliminated. *Lu256* contains a triply nested loop performing LU decomposition. The compiler is able to replace a nested barrier with a counter, reducing synchronization overhead and improving processor utilization.

Two other programs, *erle64* and *shallow*, experience less improvement despite significant decrease in synchronization. This behavior demonstrates that once sufficient granularity of parallelism is achieved, further reductions in synchronization overhead has less impact. However, note that for all programs the effect of synchronization optimization grows as the number of processors increase. We expect that with enough processors, even *erle64* and *shallow* will significantly benefit from eliminating barrier synchronization.

### 4.4 Summary and Future Extensions

We found our synchronization optimizations to successfully eliminate a moderate amount of barrier synchronization in the programs tested. For a few programs, the improvements were dramatic. It is interesting to note that our optimizations performed better on the cleanly-written short programs than on the larger benchmarks. For instance, *shallow* is an inlined and cleaned-up version of *swm256* from the SPEC benchmarks. The computation pattern is virtually identical, but the compiler was successful in eliminating synchronization only in *shallow*. This suggests that procedures and messy code (*e.g.,* control flow, scalar symbolics) can significantly retard the analysis required to perform synchronization optimizations.

Our prototype optimizer does not include two important techniques, interprocedural analysis and array privatization. *Interprocedural analysis* can in many cases locate large parallel loops, reducing synchronization overhead significantly by itself [8, 15]. We believe that there are many programs where communication analysis can reduce synchronization overhead further, particularly as the number of processors increases. Interprocedural analysis can

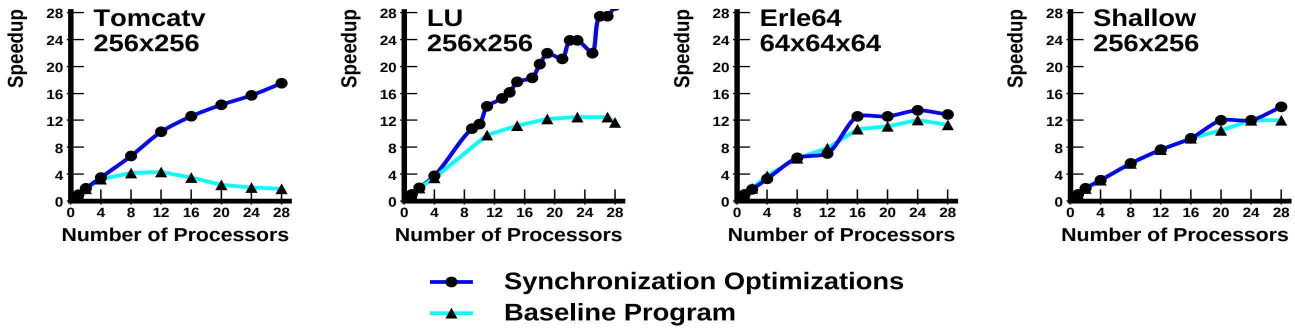


Figure 10: Program Speedups

enhance synchronization optimizations for these programs by creating larger SPMD regions and providing more precise symbolic information for communication analysis.

A second technique, *array privatization*, identifies opportunities when processors may be assigned private copies of an array to eliminate storage-related dependences [27]. Communication analysis is improved because privatized arrays do not cause communication, making it more likely barriers may be eliminated. In the future we plan to integrate our synchronization optimizer into the SUIF interprocedural framework and evaluate its effectiveness. We also intend to experiment with heuristics for constructing SPMD regions that are more aggressive about including control flow and guarded code, as well incorporate run-time scheduling techniques for dynamic load balancing.

## 5 Related Work

The techniques used to eliminate or lessen synchronization in our compiler are similar to those used by distributed-memory compilers to calculate explicit communication [1, 18]. However, distributed-memory compilers do not consider synchronization explicitly. Instead, the entire program is placed in a single parallel SPMD region, and the compiler inserts calls to nonblocking *send* and blocking *receive* routines that implicitly synchronize processors.

In comparison, our compiler has a simpler job when compiling for shared-address architectures because it does not have to explicitly manage every nonlocal data access. Instead, it can rely on barriers or counters to synchronize a large number of disparate nonlocal accesses at once. The compiler can also restrict its analysis to smaller regions of the program and selectively perform analysis to compute sending and receiving processors in situations where it is profitable.

Researchers have looked at the problem of exploiting parallelism through the use of data and event synchronization, where *post* and *wait* statements are used to synchronize between data items [26] or loop iterations [20]. In comparison, we do not attempt to extract fine-grain parallelism through synchronization. Instead, we try to expand the granularity of loop-level parallelism found by parallelizing compilers; we also reduce overhead by only synchronizing once between each pairs of processors.

Researchers compiling for SIMD languages such as Fortran 90 and Modula-2* have been faced with a programming model that assumed the existence of barriers following each expression evaluation [17, 24, 25]. Simple data dependence analysis can be used to reduce barrier synchronization by orders of magnitude, greatly improving performance. In comparison, we begin with parallel loops discovered by parallelizing compilers, where the remaining barriers are significantly harder to remove.

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 [17]. 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 [24]. Both algorithms require first computing potential interprocessor communication between all statements, the most expensive step in our optimizer. In comparison, our greedy algorithm incrementally calculates potential communication, inserting barriers as soon as communication is discovered between two groups. In practice we usually achieve optimality since SPMD regions are small and require few, if any, barriers.

Cytron *et al.* were the first to explore the possibilities of exploiting SPMD code for shared-memory multiprocessors [11]. They concentrated on safety concerns and the effect on privatization. In comparison, we focus on exploiting compile-time computation partition and communication analysis.

Bodin *et al.* [9] are exploring techniques similar to ours. 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. Good results are reported for two programs, *shallow* and *tred2*. In comparison, we apply communication analysis to precisely identify data that must be moved between processors, based on techniques from distributed-memory compilers [1, 2, 3, 18]. This information allows us to eliminate more barriers; it also enables the use of counters to satisfy producer-consumer synchronization.

Finally, a number of researchers have developed techniques for analyzing synchronization in explicitly parallel programs [19, 22]. By focusing on compiler-parallelized programs, our analysis is simpler and we expect more precise because it is performed on the original sequential source.

## 6 Conclusion

We presented a new compiler optimization for reducing synchronization overhead for compiler-parallelized scientific codes. Elements include a hybrid fork-join/SPMD programming model, globally consistent compile-time computation partitions, and communication analysis similar to that performed by compilers for distributed-memory machines. Our compiler optimization has been implemented in the Stanford SUIF compiler; we extensively evaluate its performance using standard benchmark suites. Communication analysis proved to be key in eliminating barriers or replacing them with less expensive forms of synchronization such as counters. Results show on average 29% of the barrier synchronization executed at run time are eliminated; some programs achieve dramatic reductions. We expect our synchronization optimization to complement other compiler optimizations [4]; it will become increasingly important as microprocessors outspeed interconnection network and multiprocessor workstations become common.

## 7 Acknowledgements

We gratefully acknowledge Monica Lam and other members of the Stanford SUIF compiler group for providing the software infrastructure upon which our implementation is based. In particular, we wish to thank Jennifer Anderson for developing the SUIF passes responsible for computing data decompositions and generating SPMD code. We are also indebted to Saman Amarasinghe for providing tools to manipulate systems of symbolic linear inequalities.

## References

[1] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[2] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Williamsburg, VA, April 1991.

[3] C. Ancourt and F. Irigoin. Automatic code distribution. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, Vienna, Austria, July 1992.

[4] J. Anderson, S. Amarasinghe, and M. Lam. Data and computation transformation for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

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

[6] B. Appelbe and B. Lakshmanan. Optimizing parallel programs using affinity regions. In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[7] B. Bixby, K. Kennedy, and U. Kremer. Automatic data layout using 0-1 integer programming. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, August 1994.

[8] W. Blume et al. Polaris: The next generation in parallelizing compilers,. In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[9] F. Bodin, L. Kervella, and M. O'Boyle. Synchronization minimization in a SPMD execution model. Technical Report TR-94-863, INRIA, September 1994.

[10] D. Chen, H. Su, and P. Yew. The impact of synchronization and granularity on parallel systems. In *Proceedings of the 17th International Symposium on Computer Architecture*, Seattle, WA, May 1990.

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

[12] S. Dwarkadas, P. Keleher, A. Cox, and W. Zwaenepoel. An evaluation of software distributed shared memory for next-generation processors and networks. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[13] J. Fang and M. Lu. An iteration partition approach for cache or local memory thrashing on parallel processing. In *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, Santa Clara, CA, August 1991.

[14] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[15] M. W. Hall, S. Amarasinghe, and B. Murphy. Interprocedural analysis for parallelization: Design and experience. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, San Francisco, CA, February 1995.

[16] R. v. Hanxleden and K. Kennedy. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

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

[18] 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.

[19] T. Jeremiassen and S. Eggers. Static analysis of barrier synchronization in explicitly parallel systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Montreal, Canada, August 1994.

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

[21] E. Markatos and T. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[22] S. Midkiff and D. Padua. Compiler generated synchronization for DO loops. In *Proceedings of the 1986 International Conference on Parallel Processing*, St. Charles, IL, August 1986.

[23] D. Padua and M. J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[24] 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.

[25] 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.

[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] P. Tu and D. Padua. Automatic array privatization. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[28] R. Wilson et al. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[29] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.