Optimizing Parallel Programs with Explicit Synchronization

Arvind Krishnamurthy and Katherine Yelick
Computer Science Division
University of California, Berkeley *

Abstract: We present compiler analyses and optimizations for explicitly parallel programs that communicate through a shared address space. Any type of code motion on explicitly parallel programs requires a new kind of analysis to ensure that operations reordered on one processor cannot be observed by another. The analysis, based on work by Shasha and Snir, checks for cycles among interfering accesses. We improve the accuracy of their analysis by using additional information from post-wait synchronization, barriers, and locks.

We demonstrate the use of this analysis by optimizing remote access on distributed memory machines. The optimizations include *message pipelining*, to allow multiple outstanding remote memory operations, conversion of two-way to one-way communication, and elimination of communication through data re-use. The performance improvements are as high as 20-35% for programs running on a CM-5 multiprocessor using the Split-C language as a global address layer.

1 Introduction

Optimizing explicitly parallel shared memory programs requires new types of static analysis to ensure that accesses reordered on one processor cannot be observed by another. Intuitively, the parallel programmer relies on the notion of *sequential consistency*: the parallel execution must behave as if it were an interleaving of the sequences of memory operations from each of the processors [12]. If only the local dependencies within a processor are observed, the program execution might not be sequentially consistent [15]. To guarantee sequential consistency under reordering transformations, a new type of analysis called *cycle detection* is required [18].

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense monitored by the Office of Naval Research under contract DABT63-92-C-0026, by the Department of Energy under contract DE-FG03-94ER25206, and by the National Science Foundation. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

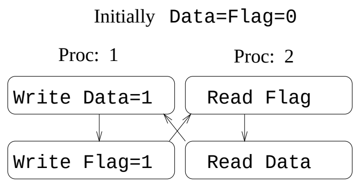

Figure 1: If the read of *Flag* returns 1, the read of *Data* should see the new value.

An example to illustrate sequential consistency is shown in Figure 1. The program is indeterminate in that the read of `Flag` may return either 0 or 1, and if it is 0, then the read to `Data` may return either 0 or 1. However, if 1 has been read from `Flag`, then 1 must be the result of the read from `Data`. If the two program fragments were analyzed by a sequential compiler, it might determine that the reads or writes could be reordered, since there are no local dependencies. If either pair of the accesses is reordered, the execution in which `Flag` is 1 and `Data` is 0, might result.

Even if the compiler does not reorder the shared memory accesses, reordering may take place at many levels in a multiprocessor system. At the processor level, a superscalar may issue an instruction as soon as all its operands are available, so writes to different locations might be issued in the order the values become available. Most processors have write buffers, which allow read operations to overtake write operations that have been issued earlier. In fact, on the SuperSparcs [20] the write-buffer itself is not guaranteed to be FIFO. Reordering may also take place at the network level in distributed memory multiprocessors, because some networks adaptively route packets to avoid congestion. Even if packets do not get reordered, two access sent to two different processors may be handled out of order, since latencies may vary. Also, on a machine like DASH [13], with hardware caching, writes do not wait for all invalidations to complete, so remote accesses might appear to execute in reverse-order. These architectural features usually come with support to ensure sequential consistency, such as a memory barrier or a write-buffer flush to enforce ordering between memory operations, or a test for completion of a remote operation. However, these new instructions must be inserted by the compiler. If a standard uniprocessor compiler is used for generating code, these special instructions would not be automatically inserted.

The cycle detection problem is to detect access cycles, such as the one designated by the figure-eight in Figure 1. In addition to observing local dependencies within a program, a compiler must ensure that accesses issued by a single processor in a cycle take place in order. Cycle detection is necessary for most optimizations involving code motion, whether the programs run on physically shared or distributed memory and whether they have dynamic or static thread creation. Cycle detection is not necessary for automatically parallelized sequential programs or data parallel programs with sequential semantics, because every pair of accesses has a fixed order, which is determinable at compile-time. The additional problem for explicitly parallel programs comes directly from the possibility of non-determinism, whether or not the programmer chooses to use it.

In spite of the semantic simplicity of deterministic programming models, for performance reasons many applications are written in an explicitly parallel model. As we noticed with our toy example, uniprocessor compilers are ill-suited to the task of compiling explicitly parallel programs, because they do not have information about the semantics of the communication and synchronization mechanisms. As a result, they either generate incorrect code or miss opportunities for optimizing communication and synchronization, and the quality of the scalar code is limited by the inability to move code around parallelism primitives [15].

We present optimizations for multiprocessors with physically distributed memory and hardware or software support for a global address space. As shown in table 1, a remote reference on such a machine has a long latency [2][21][13]. However, most of this latency can be overlapped with local computation or with the initiation of more communication, especially on machines like the J-Machine and *T, with their low overheads for communication startup.

	CM-5	T3D	DASH
Remote Access	400	85	110
Local Access	30	23	26

Table 1: Access latencies for local and remote memory modules expressed in terms of machine cycles.

Three important optimizations for these multiprocessors are overlapping communication, eliminating round-trip message traffic, and avoiding communication altogether. The first optimization, *message pipelining*, changes remote read and write operations into their split-phase analogs, *get* and *put*. In a split-phase operation, the initiation of an access is separated from its completion [6]. The operation to force completion of outstanding split-phase operations comes in many forms, the simplest of which (called *sync* or *fence*) blocks until all outstanding accesses are complete. To improve communication overlap, *put*s and *get*s are moved backwards in the program execution and *sync*s are moved forward. The second optimization eliminates acknowledgement traffic, which are required to implement the *sync* operation for *put*s. A final optimization is the elimination of remote accesses by either re-using values of previous accesses or updating a remote value locally multiple times before issuing a write operation on the final value.

Cycle detection was first described by Shasha and Snir [18] and later extended by Midkiff, Padua, and Cytron to handle array indices [16]. In previous work, we showed that by restricting attention to Single Program Multiple Data (SPMD) programs, one could significantly reduce the complexity of cycle detection [11]. The primary contribution of this paper is improved cycle detection that makes use of synchronization information in the program. Shasha and Snir's analysis, when applied to real applications, discovers a huge number of spurious cycles, because cycles are detected between accesses that will never execute concurrently due to synchronization. We use synchronization analysis to eliminate these spurious cycles.

The rest of the paper is organized as follows. The source programming language is described in section 2. We present basic terminology in section 3 and a brief summary of Shasha and Snir's result in section 4. In section 5, we present our new algorithms that incorporate synchronization analysis, and in sections 6 and 7, we give code generation and optimizations for distributed memory machines. Section 8 estimates the potential payoffs of our approach by optimizing some application kernels. Related work is surveyed in section 9 and conclusions drawn in section 10.

2 Programming Language

Our analyses are designed for explicitly parallel shared memory programs. We have implemented them in a source-to-source transformer for a subset of Split-C [6].

Split-C is an explicitly parallel SPMD language for programming distributed memory machines using a global address space abstraction. The parallel threads interact through reads and writes on a shared address space that contains distributed arrays and shared objects accessible through wide pointers. The most important feature of Split-C as a target is its support for split-phase (or non-blocking) memory operations. Given pointers to global objects `src1` and `dest2`, and local values `src2` and `dest1` of the same type, the split-phase versions of read and write operations on the global objects are expressed as:

```
get(dest1, src1)
put(dest2, src2)
/* Unrelated computation */
sync();
```

In the first assignment statement, a `get` operation is performed on `src1`, and in the second, a `put` is performed on `dest2`. Neither of these operations are guaranteed to complete (the values of `dest1` and `dest2` are undefined) until after the `sync` statement. A *get* operation initiates the read of a remote location, but it does not wait for the value to be fetched. Similarly, a *put* operation does not wait for the acknowledgement that the write occurred on the remote processor. The `sync` operation delays the execution for previous non-blocking accesses to complete. On a distributed memory machine, the *get* and *put* operations are implemented using low-level messages sent across the interconnection network. Therefore, split-phase operations facilitate communication overlap, but the `sync` construct provides less control than one might want, because it groups all outstanding `puts` and `gets` from a single processor. Split-C also provides finer grained mechanisms in which a `sync` object, implemented by a counter, is associated with each memory operation. Examples of these are given in Section 6.

Split-C also provides a `store` operation, which is a variant of the `put` operation. A `store` operation generates a write to a remote memory location, but does not acknowledge when the write operation completes. It exposes the efficiency of one-way communication in those cases where the communication pattern is well understood. By transforming a `put` to a `store`, we not only reduce network contention by

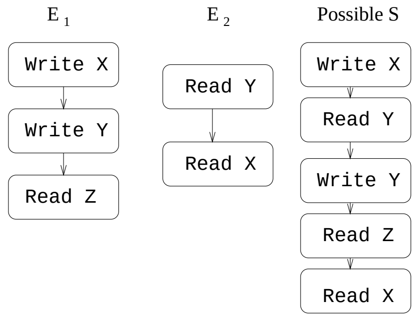

Figure 2: The figure shows a shared memory execution and one possible total order on operations. Executions in which the Write to Y appears to happen before the Write to X would not be legal.

reducing the number of packets but also reduce the processing time spent by the processors in generating and handling the acknowledgement traffic.

The source language differs from Split-C in two key aspects. First, the source language does not provide split-phase operations; all accesses to shared memory are blocking. This design choice stems from the observation that split-phase operations are good for performance but hard to use. Second, the global address space abstraction is provided in the source language only through a distributed array construct and through shared scalar values; the global address space cannot be accessed through global pointers, which are supported by Split-C. Disallowing global pointers allows us to implement our analysis techniques without full-blown pointer alias analysis. However, there are no restrictions imposed on the use of pointers into the local address space. The type system prevents the creation of local pointers into the global address space, and this allows us to ignore local memory accesses and local pointers in our analysis. Our source language design allows us to write meaningful parallel programs with optimized local computational kernels and global phases that communicate using either shared variables or distributed arrays.

3 Sequential Consistency

A parallel execution E on n processors is given by n sequences of instruction executions $E_1, \ldots, E_n$. We consider two types of executions: a *shared memory* execution allows only atomic reads and writes to shared variables; a *weak memory* execution may contain split-phase as well as atomic operations.

Given a processor execution $E_i = a_1, \ldots, a_m$, we associate with E_i a total order $a_1 < a_2 < \ldots < a_m$. In a shared memory execution, reads and writes happen atomically in the order they are issued, with no predetermined ordering between accesses issued by different processors. Therefore, the *program execution order*, E, is the union of these E_i's. An execution E is *sequentially consistent* if there exists a total order S of the operations in E, i.e., $E \subseteq S$, such that S is a correct sequential execution where the reads must return the value of the most recent preceding write [12]. For example, in Figure 2, if the read to Y returns a new value written by E_1, then the read of X must also return the value written by E_1[1].

[1] This program might come from a case in which Y is acting as a "presence flag" for the value being written into X.

System Contract 1 *Given a shared memory program, a correct machine must produce only sequentially consistent executions for that program.*

This contract does not specify the behavior of programs with `put` and `get` or other non-blocking memory operations. In order to extend the system contract for programs with weak memory accesses, rather than relying on a particular instruction set with non-blocking memory operations and synchronizing accesses, we use a more general framework proposed by Shasha and Snir [18]. A *delay set* D specifies some pairs of memory accesses as being ordered, which says that the second operation must be delayed until the first one is complete. For example, in Figure 3, E_1 specifies the accesses issued by a processor, and D_1 specifies the delay constraints for executing the accesses. A `sync` operation, which is one particular mechanism for expressing delay constraints, could be introduced to prevent the `get` operation from being initiated before the `puts` complete. In general, given an execution E, D is a subset of the ordering given by E, i.e., $D \subseteq E$. A weak memory execution given by E and D is *weakly consistent* if there exists a total order S of the accesses in D, i.e., $D \subseteq S$, such that S is a correct sequential execution.

System Contract 2 *Given a weak memory program, a correct machine must produce only weakly consistent executions for that program.*

E is a dynamic notion based on a particular execution. During compilation, we approximate E by the *program order* P, defined as the transitive closure of the n program control flow graphs, one per processor. The compiler computes a delay set D, which is a subset of P. We say that a delay set D is *sufficient* for P if, on any machine that satisfies the second system contract, all possible executions of P are sequentially consistent. Note that if we take D to be P, which means that we block on every remote memory access, it forces our machine to produce a sequentially consistent execution. Our goal during program analysis is to find a much smaller D that still ensures sequential consistency.

4 Shasha and Snir's Algorithm

A violation of sequential consistency occurs when the "happens before" relation, which is $E \cup S$, contains a cycle. An example is shown in Figure 1. In this case, the figure-eight formed by the arrows is the cycle that violates sequential consistency. All violations are due to such cycles, although in general the cycles may extend over more than two processors and involve as many as $2n$ accesses. The cross-processor edges in the cycles are conflicting (read-write or write-write) accesses to the same variable by two different processors. We define the *conflict set* C to be a conservative approximation to these interferences: C contains all unordered pairs of shared memory operations a_1, a_2, such that a_1 and a_2 are issued by different processors, both access (or could access) the same shared variable, and at least one is a write operation.

Shasha and Snir proved that there exists a minimum delay set, $D_{S\&S}$, that can be defined by considering cycles in $P \cup C$. The primary idea is that if $P \cup C$ does not contain any cycles (as in Figure 4), then $E \cup S$ would not contain any cycles since $P \cup C$ is a conservative compile-time superset of $E \cup S$. We reformulate their result with the following definitions.

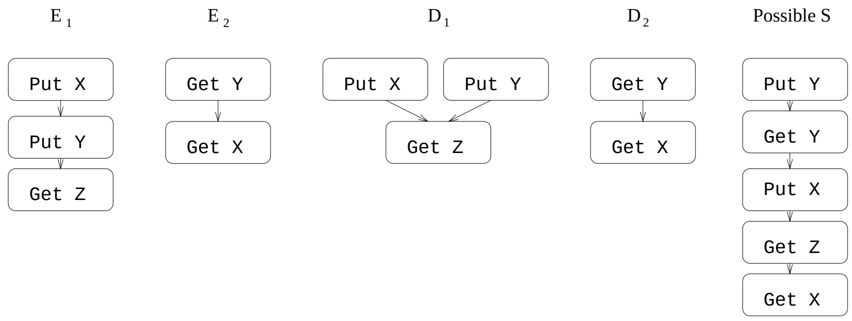

Figure 3: A weak memory execution with D containing some of the edges in E. Since the `puts` in E_1 are not ordered, they may execute in either order. However, an execution in which the `get` of Z appears to happen before either one of the `puts` would be illegal.

Definition 1 *A path* $[a_1, \ldots, a_m] \in P \cup C$ *is a* simple path*, if for any access* a_i *in the path, if* a_i *is an access on processor* P_k*, then the following hold:*

1. *If* a_{i+1} *is also on* P_k*, then for all other accesses* a_j $(j \neq i$ *and* $j \neq i+1)$*,* a_j *is not on* P_k*.*
2. *If* a_{i-1} *and* a_{i+1} *exist* $(i \neq 1$ *and* $i \neq n)$ *and* $[a_{i-1}, a_i] \in C$ *and* $[a_i, a_{i+1}] \in C$*, then for all* $j \neq i$*,* a_j *is not on* P_k*.*

Thus, except for the end-points of the path, a simple path is one that visits each processor at most once, with at most two accesses per processor during a visit. A special case of simple paths points to a potential violation of sequential consistency.

Definition 2 *Given an edge* $[a_m, a_1]$ *in some* P_k*, a path* $[a_1, \ldots, a_m] \in P \cup C$ *is called a* back-path*, for* $[a_m, a_1]$ *if* $[a_1, \ldots, a_m]$ *is a simple path.*

Shasha and Snir define a particular delay set, denoted here $D_{S\&S}$, which is sufficient and in some sense minimal.

Definition 3 $D_{S\&S} = \{[a_i, a_j] \in P | [a_i, a_j]$ *has a back-path in* $P \cup C\}$*.*

Theorem 1 *[18]* $D_{S\&S}$ *is sufficient.*

The minimality results on $D_{S\&S}$ says that given straight-line code without explicit synchronization, if a pair of accesses in $D_{S\&S}$ is allowed to execute out of order (i.e., is omitted from the delay set when the program is run), there exists a weakly consistent execution of that program that is not sequentially consistent. This notion of minimality is not as strong as one would like, because it ignores the existence of control structures and synchronization constructs that prevent certain access patterns. In the section that follows, we extend this framework to include synchronization analysis, which is critical in reducing spurious delay edges in real parallel programs.

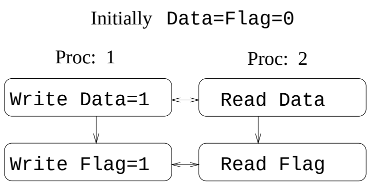

Figure 4: Example of a parallel program that does not require any delay constraints.

5 Using Synchronization Information

The delay set computation described in the previous section does not analyze synchronization constructs and is therefore overly conservative. It is correct to treat synchronization constructs as simply conflicting memory accesses, but this ignores a valuable source of information, since synchronization creates mutual exclusion or precedence constraints on accesses executed by different processors. For example, Figure 5 shows two program segments that access the variables X and Y. The delay set $D_{S\&S}$ contains edges between these access, prohibiting overlap. However, if synchronization behavior is taken into account, the delays are seen to be unnecessary, because the `post` will be delayed for the writes to complete and the reads cannot be started until the `wait` completes.

In this section, we modify the delay set computation to incorporate synchronization analysis for three constructs: post-waits, barriers, and locks. Our synchronization analysis presumes that synchronization constructs can be matched across processors, and we describe runtime to ensure this dynamically. This analysis only helps if the programmer uses the synchronization primitives provided by the language. If the programmer builds synchronization using reads and writes, we would not be able to detect the synchronization. In this case our algorithm is still correct, but we would not be able to prune the delay set.

5.1 Analyzing Post-Wait Synchronization

Post-wait synchronization is used for producer-consumer dependencies. The consumer executes a `wait`, which blocks until the producer executes a `post`. This creates a strict precedence between the operations before the `post` and the operations after the `wait`[2]. We start by considering two examples that use post-wait synchronization, and then present the modified delay set construction. In our discussion, we use a precedence relation R that captures the *happens-before* property of accesses.

Definition 4 *A* precedence relation R *is a set of ordered pairs of accesses,* $[a_1, a_2]$*, such that* a_1 *is guaranteed to complete before* a_2 *is initiated.*

Consider the computation of the delay set for the program in Figure 5. $D_{S\&S}$ is $\{[a_1, a_2], [a_2, a_3], [a_1, a_3], [a_4, a_5], [a_5, a_6], [a_4, a_6]\}$, which will force completion of a_1 before the

[2] In our analysis, we assume that it is illegal to post more than once on an event variable.

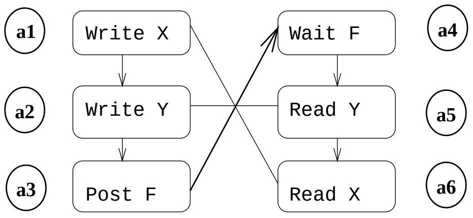

Figure 5: The post-wait synchronization enables the ordering of conflict edges, which results in fewer back-paths.

initiation of a_2 and a_5 before a_6. The semantics of post-wait synchronization require a precedence edge from a_3 to a_4, which eliminates one direction of the conflict edge between a_3 and a_4 and leads to a smaller delay set. Starting with the delay set $\{[a_2, a_3], [a_1, a_3], [a_4, a_5], [a_4, a_6]\}$, we can order the other conflict edges $[a_1, a_6]$ and $[a_2, a_5]$ by transitivity, and thus destroy the remaining back-paths.

As this example illustrates, the delay set and precedence relation will be computed through a process of refinement. Initially, the precedence relation contains only those edges that directly link a `post` and a `wait`. We then create an initial delay set D_1 with those edges from $D_{S\&S}$ that involve at least one synchronization construct. This says that some delay edges—those involving synchronization—are more fundamental than others. Once D_1 is computed, the precedence relation R is expanded to include the transitive closure of itself and D_1. The example provides two key insights into how we could use synchronization information. First, by providing a directionality to a conflict edge, we impose more restrictions on the interleaving of accesses from different processors, which results in a smaller delay set. Second, the precedence relation R serves as the catalyst for initiating the refinement process.

The process of invalidating a back-path does not always involve directing a conflict edge. R could be used for removing certain accesses that are not qualified to appear in back-paths, and thus decrease the number of back-paths that we discover. In Figure 6, there are simple paths from a_3 to a_1 and from a_6 to a_4. Furthermore, a_3 and a_4 are synchronization accesses, so $[a_1, a_3]$ and $[a_4, a_6]$ belong to the initial delay set D_1. This information, when combined with the precedence edge $[a_3, a_4]$, implies that a_1 precedes a_6 for any execution of the program. Since a simple-path to a_1 corresponds to a runtime execution where all the accesses in the sequence execute before a_1, a_6 will never occur in a simple-path to a_1. We therefore remove a_6 while determining the existence of simple-paths to a_1. Removal of a_6 destroys the simple-path from a_2 to a_1, which otherwise

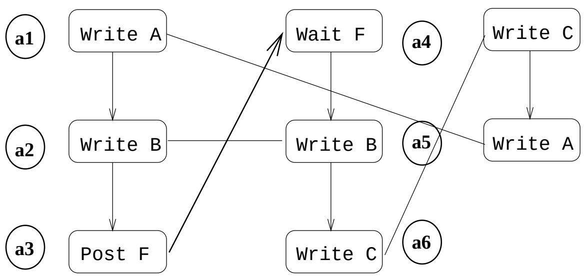

Figure 6: An example where synchronization analysis disqualifies certain accesses from appearing in back-paths.

```
void foo() {

   barrier();
   for (i=0; i<n; i++) {
      ...
      barrier();
   }
   ...
   barrier();
}
```

Figure 7: Inaccuracies in analysis of barrier statements

would have resulted in $[a_1, a_2]$ being added to the delay set.

Using these examples as motivation, we propose a general scheme for finding a delay set. We initially compute the delay set D_1, which relates synchronizing accesses to non-synchronizing accesses, and combine it with direct precedence edges to obtain complete precedence information. For post-wait operations, this combining process requires the dominator tree of the control flow graph. A node u is said to *dominate* a node v if u appears on every path from the entry node of the graph to v. (Domination information is efficiently represented using a *dominator tree*, which stores only the closest dominators.) We now present the modified algorithm for computing the delay set.

1. Compute the dominator tree.
2. Compute initial delay restrictions D_1 by restricting the simple-path algorithm from the previous section to pairs that include one synchronization access.
3. Compute the set of precedence edges, R_1, between matching `post` and `wait` constructs.
4. For every pair of access statements a_1 and a_2, check whether there exists two other statements b_1 and b_2 that satisfy the following constraints.
   (a) a_1 dominates b_1 and b_2 dominates a_2,
   (b) $[a_1, b_1] \in D_1$ and $[b_2, a_2] \in D_1$, and
   (c) $[b_1, b_2] \in R_1$

   Add $[a_1, a_2]$ to R if b_1 and b_2 exist.
5. The original conflict set C contained unordered pairs. Order the pairs that have a precedence as follows: Let $C_1 = C - \{[a_2, a_1] \mid [a_1, a_2] \in R\}$.
6. Let $D = D_1 \cup \{[a_i, a_j] \in P \mid [a_i, a_j]$ has a back-path in $P \cup C_1\}$.

By eliminating accesses and ordering conflict edges before checking for back-paths, we reduce the number of back-paths that are discovered. There is a corresponding decrease in the size of the delay set, which results in improvements in execution times of the programs.

5.2 Analyzing Barrier Synchronization

`Barrier` statements can be used to separate the program into different phases that do not execute concurrently. The analysis for `barriers` is similar to that of post-wait synchronization, since crossing a barrier introduces a precedence relation. As before, we add the delay edges between accesses and `barriers` before compute the delay set for the rest of the program.

To use barriers for computing precedence, we need to line up the barriers, which is undecidable in general. Figure 7 shows a sample program where it is likely that the *barrier* statements do line up during program execution, but to prove that assertion at compile-time, our analysis needs to prove that the function foo gets called by all the processors at the same time and the loop inside the function executes the same number of iterations. Rather than adding sophisticated analysis to line up barriers [10], we use a simple runtime solution that works well for many real programs. We add a run-time check to each barrier to determine whether these are the ones lined up during compilation. The compiler produces two copies of the code, one with pipelining optimizations and the other without any optimizations. If the processors are indeed synchronized and executing the barrier operations as predicted, the optimized version of the code is run. This approach to analyzing barriers also allows us to overcome separate compilation issues for many real programs.

5.3 Lock Based Synchronization

We can extend our synchronization analysis to locks, even though there are no strict precedence relations implied by the use of locks. We again compute D_1 for pairs of accesses that include a synchronization construct. We then determine the set of accesses guarded by a lock. An access a is said to be *guarded* by the lock l, if the following conditions hold:

1. a is dominated by a *lock* l operation (which we will call b_1), and there are no intervening *unlock* l operations.
2. a dominates *unlock* l operation, which we will call b_2.
3. $[b_1, a] \in D_1$ and $[a, b_2] \in D_1$

If access statements a_1 and a_2 are guarded by the lock l, we remove all other access statements that are guarded by the same lock before checking for a simple-path from a_2 to a_1. This is a valid operation by the following reasoning. If $a_2, b_1, b_2, ..., b_k, a_1$ is a simple-path, then the accesses corresponding to $b_1, b_2, ..., b_k$ must occur after a_2 and before a_1. It follows from our definition of being guarded by a lock that none of $b_1, b_2, ..., b_k$ can be guarded by the same lock and still appear in a violation sequence. This improvement to the delay set construction allows accesses within critical regions to be overlapped.

6 Code Generation

The notion of a delay set can be used to generate code for a variety of memory models, one of which is the `put`, `get` model provided by Split-C. In this section we describe our source level transformer for Split-C, which uses cycle detection and synchronization analysis.

The input to the code generation phase is the control flow graph, the delay set computed by the back-path recognition algorithm, and the *use-def* graph for each processor's variable access (obtained through standard sequential compiler analysis). As we mentioned in section 4, to compute the delay set, we can use a conservative approximation to the conflict set C without affecting the correctness of our analysis. This is crucial to our analysis since there are inherent inaccuracies in the array analysis that is required to generate conflict edges for array accesses.

During the code generation process, both the delay constraints and local dependencies must be observed. The generated code contains `put`, `get`, and `store` constructs, as well as various types of `sync` statements. Normally, a `sync` statement will force completion of all previous `puts` and `gets` from the issuing processor. However, Split-C also provides a mechanism called *synchronizing counters* to wait for the completion of a subset of outstanding accesses. The programmer specifies a counter when issuing `puts` and `gets`, and again when issuing the `sync`, which will wait for only those accesses with a matching counter.

The first step in code generation is to split remote accesses into an initiation and synchronization. A remote `read` of `X` into `y` is transformed into `get_ctr(y, X, counter)` followed by `sync_ctr(counter)`, where `counter` is either a new or reused synchronizing counter. This transformation is always legal, but analysis is needed to move the two operations away from each other, thereby allowing communication overlap.

Separating Initiation from Completion. Figure 8 shows an example of code that might be generated after calculating delay and def-use edges. The synchronization of the `get` on X has been moved away from the initiation, but because of the conditional, two synchronizing points are needed. The duplication may result in unnecessary overhead, but is legal because `sync_cntr` operations are idempotent.

The algorithm for moving a `sync_ctr` operation a_{sync} away from its corresponding initiation a_{init} involves repeated applications of the following rules:

1. If a_{sync} is at the end of a basic block, propagate a_{sync} to all the successors of the basic block and continue the motion on each copy of a_{sync}.
2. If a_{sync} is in the middle of a basic block, let a' be the operation the immediately follows it.
   (a) If there is a delay or def-use constraint of the form $[a_{init}, a']$, terminate the movement of a_{sync}.
   (b) If a' is another copy of the synchronization a_{sync}, merge the two a_{sync} operations.
   (c) Otherwise, move a_{sync} past a'.

The above algorithm moves `sync_ctr` operations as far away from initiation as possible. As shown in Figure 8, this may not be the best strategy, since it can lead to multiple synchronization points in the same control path. Similarly, if a `sync_ctr` is propagated into a loop body, it will be executed in every iteration, even though the first execution is sufficient. Our compiler uses heuristics to avoid some of these cases, since data on average executions of control paths and machine parameters would be needed to choose the best placement.

One-Way Communication Optimization. Another transformation that provides large performance benefits on some machines is to transform two-way communication into one-way communication. The `put` operation has an acknowledgement that signals the completion of the operation. If there are no delay constraints that require the completion of a `put` access, we can transform the access into a one-way operation, called a `store`. One way to ensure completion of `stores` is with a global synchronization point that stalls until all stores across the machine have completed. This transformation is usually valid if the `sync_ctr` operation propagates all the way to a `barrier` synchronization.

```
x = X
y = 2
if (foo()) {
  y = x + 1
}
Z = 1
```

Source code

```
get_ctr(&x, &X, ctr1)
y = 2
if (foo()) {
  sync_ctr(ctr1);
  y = x + 1;
}
sync_ctr(ctr1);
z = 1
```

Target code

Figure 8: Code Generation: Upper case letters are shared variables, lower case letters are local variables, the solid line is a delay edge, and the dashed line is a def-use edge.

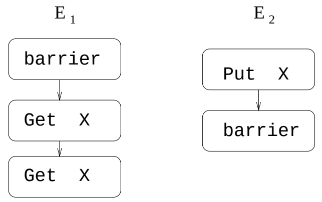

Figure 9: Barrier synchronization guarantees that X is read-only in the second phase.

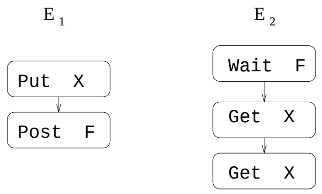

Figure 10: X can be cached by E_2 since the updates to X are guaranteed to be complete.

7 Eliminating Remote Accesses

Delay sets are needed for any transformation that involves code motion on explicitly parallel code. In this section we consider a second class of transformations for distributed memory machines, which lead to the elimination of remote accesses through a kind of common-subexpression elimination.

Consider two accesses a_1 and a_2 within a single basic block that are gets to the same variable X. If X is not being written concurrently, then the second get can be eliminated. Two examples are shown in Figure 9. In the first case, there is a barrier that marks the transition to X being read-only, and in the second, the post-wait synchronization ensures that the gets are issued only after the put is complete. The synchronization analysis described in Section 5 identifies these synchronization regions and orders the conflict edges between the gets and puts to X. Since there will not be a delay edge between the two gets to X, the second one can be eliminated.

Mutually exclusive access is sufficient but not necessary for elimination of repeated gets. It may be possible to reuse a previously read value even when there are intervening global accesses, as long as it is legal to move the second get up to the point of the first one. The algorithm used for remote access elimination is essentially the reverse of that used for sync_ctr propagation: the second get is moved backwards in the code until it reaches a operation that shares a delay edge or local dependence. If this propagation is successful, we will end up with a sequence like:

```
get(local1, X, counter1)
...
sync_cntr(counter1)
get(local2, X, counter2)
...
sync_cntr(counter2)
```

At this point, the second get is replaced by a local assignment of local1 to local2, and the second sync_cntr is eliminated, along with any of its copies.

The examples presented so far eliminate redundant reads, which is similar to saving a value in register. The technique can be applied to a variety of other communication-eliminating optimizations as illustrated in Figure 11. For simplicity, they are shown as transformations on the higher level code, with temporaries introduced to minimize conflicts during code motion. Reading a remote variable that has recently been written can be avoided if the written value is still available. When a thread issues two successive writes to the same variable, the earlier writes can be buffered in a local variable and the final value written to the remote copy at a later point. This is equivalent to using write-back cache, rather than a write-through cache.

8 Experimental Results

We quantify the benefits of our approach by studying the effect of the optimizations on a set of application kernels that use a variety of synchronization mechanisms. A brief description of the applications is given below:

Ocean: This benchmark is from the Splash benchmark suite[19], and studies the role of eddy and boundary currents in large-scale ocean movements. The primary data structure is a grid, and the core of this application is a stencil-like computation.

EM3D: Em3d models the propagation of electromagnetic waves through objects in three dimensions [14]. The computation consists of a series of "leapfrog" integration steps: on alternate half time steps, changes in the electric field are calculated as a linear function of the neighboring magnetic field values and *vice versa*.

Epithelial Cell Simulation: Biologists believe that the geometric structure of the embryo emerges from a few simple, local rules of cell movement. This application is a cell aggregation simulation that allows scientists to posit such

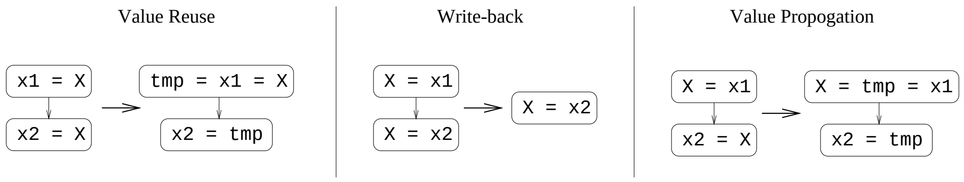

Figure 11: A set of transformations to eliminate remote accesses that are similar in spirit to standard uniprocessor optimizations like common subexpression elimination. Upper case letters denote global variables, and lower case ones local variables.

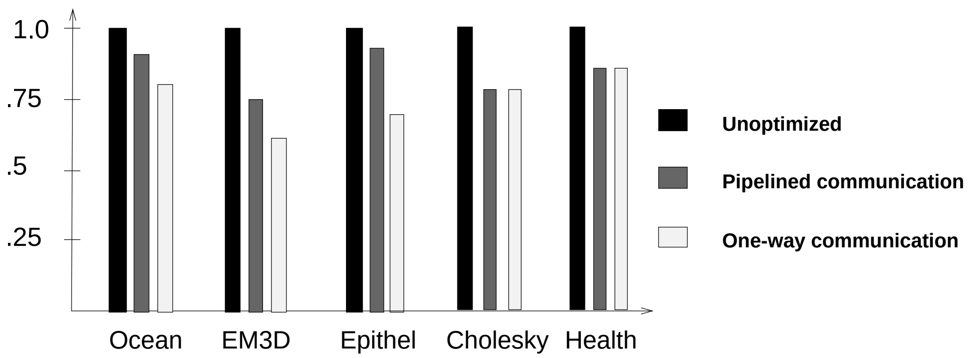

Figure 12: The figure gives the execution times, normalized so that the execution time of the code generated without analyzing synchronization constructs is 1. Thus, a relative speed of 0.5 corresponds to a factor of 2 speedup.

rules. At each time-step of the simulation, a Navier-Stokes solver calculates the fluid flow over a large grid by performing 2-D FFTs.

Cholesky: Cholesky computes the factors of a symmetric matrix. The primary data structure is a lower triangular matrix, which is distributed in a blocked-cyclic fashion. The computation is structured in a producer-consumer style, and synchronization is ensured using post-wait operations on flags.

Health: This benchmark is from the Presto application suite. Health simulates the Colombian health service system, which has an hierarchical service-dispensing system. Exclusive access to shared data structures is guaranteed by the use of locks.

The prototype compiler automatically introduces the message pipelining and one-way communications optimizations for all the applications. The execution times of these applications were improved by 20-35% through message-pipelining and one-way communication optimizations. These were measured on a 64 processor CM-5 multiprocessor. The relative speedups should be even higher on machines with lower communication startup costs or longer relative latencies. Figure 12 gives the performance results of these experiments. The base program is the executable generated after applying Shasha and Snir's cycle analysis. Our synchronization analysis results in much smaller delay sets, which in turn enables greater applicability of the message pipelining optimizations.

As a result of introducing the message pipelining optimizations, the speedup characteristics of the program changes. Figure 13 shows that the efficiency of a parallel program increases when we transform blocking operations by asynchronous operations. The increase in efficiency is a direct result of the reduction in either the time spent waiting for remote accesses to complete or the overhead of sending messages.

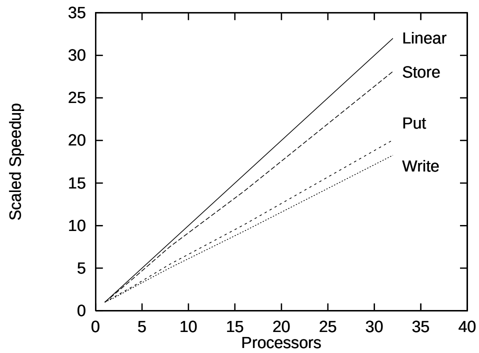

Figure 13: Speedup curves for the Epithelial application kernel with varying degrees of optimization. As expected, the optimized versions scale better with processors

9 Related Work

Most of the research in optimizing parallel programs has been for data parallel programs. In the more general control parallel setting, Midkiff and Padua [15] describe eleven different instances where standard optimizations (like code motion and dead code elimination) cannot be directly applied. Analysis for these programs is based on the pioneering work by Shasha and Snir [18], which was later extended by Midkiff et al [16] to handle array based accesses.

We analyze the synchronization[5] accesses in the program and obtain precedence and mutual exclusion information regarding remote accesses. Others have proposed algorithms for analyzing synchronization constructs in the context of framing data-flow equations for parallel programs, where strict precedence information is necessary [4][8]. Our algorithm for analyzing *post-wait* synchronization is similar in spirit; however, we can also exploit mutual-exclusion information on accesses. Also related to our work is the research that proposes weaker memory models [1, 7]. Those approaches change the programmer's model by giving programming conventions under which sequential consistency is ensured. Our work shifts this burden from the programmer to the compiler. Our analysis could also be used for compiling weak memory programs since it can determine when code motion is legal, which is critical for generating prefetch instructions.

Compilers and runtime systems for data parallel languages like HPF and Fortran-D [9] implement message pipelining optimizations and data re-use. The Parti runtime system and associated HPF compiler uses a combination of compiler and runtime analysis to optimize communication [3], and these optimizations have also been studied in the context of parallelizing compilers [17]. However, as discussed earlier, compiling data parallel programs is fundamentally different from compiling explicitly parallel programs. First, in a data parallel program, it is the compiler's responsibility to map parallelism of degree n (the size of a data structure) to a machine with `PROCS` processors, which can sometimes lead to significant runtime overhead. Second, the analysis problem for data parallel languages is simpler, because they have a sequential semantics resulting in only directed conflict edges. Standard data-dependence techniques can be used in data parallel language to determine whether code-motion or pipelining optimizations are valid.

10 Conclusions

We presented analyses and optimizations for explicitly parallel programs on distributed memory multiprocessors. The main optimization is masking latency of remote accesses by message pipelining and prefetching. Other optimizations similar to subexpression elimination and constant propagation are also enabled by the analysis. We have a prototype compiler that implements these optimizations, and we quantified the potential payoff of a few of these optimizations on a set of application kernels. The performance improvements are as high as 35% on the CM-5, with even better improvement expected on future architectures with lower communication startup.

A new form of analysis called cycle detection is needed for explicitly parallel programs in a general (not data-parallel) execution model. The analysis computes the constraints on access order to ensure a sequentially consistent programming model. We improved on the accuracy of the analysis by using synchronization information. This analysis is important, since most parallel programs reduce the number of conflicting accesses through synchronizing operations. We also show how to use this analysis to generate code for an abstract machine language, Split-C.

References

[1] S. V. Adve and M. D. Hill. Weak Ordering–A New Definition. In *17th International Symposium on Computer Architecture*, April 1990.

[2] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *International Symposium on Computer Architecture*, June 1995.

[3] H. Berryman, J. Saltz, and J. Scroggs. Execution Time Support for Adaptive Scientific Algorithms on Distributed Memory Multiprocessors. *Concurrenty: Practice and Experience*, June 1991.

[4] D. Callahan and J. Subhlok. Static Analysis of Low-level Synchronization. In *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988.

[5] W. W. Carlson and J. M. Draper. Distributed Data Access in AC. In *ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[6] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Supercomputing '93*, Portland, Oregon, November 1993.

[7] K. Gharachorloo, D. Lenoski, J. Laudon, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *17th International Symposium on Computer Architecture*, 1990.

[8] D. Grunwald and H. Srinivasan. Data flow equations for Explicitly Parallel Programs. In *ACM Symposium on Principles and Practices of Parallel Programming*, June 1993.

[9] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiler Optimziations for Fortran D on MIMD Distributed-Memory Machines. In *Proceedings of the 1991 International Conference on Supercomputing*, 1991.

[10] T. E. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *ACM Symposium on Principles and Practice of Parallel Programming*, July 1995.

[11] A. Krishnamurthy and K. Yelick. Optimizing Parallel SPMD Programs. In *Languages and Compilers for Parallel Computing*, 1994.

[12] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), September 1979.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *17th International Symposium on Computer Architecture*, May 1990.

[14] N. K. Madsen. Divergence Preserving Discrete Surface Integral Methods for Maxwell's Curl Equations Using Non-Orthogonal Unstructured Grids. Technical Report 92.04, RIACS, February 1992.

[15] S. Midkiff and D. Padua. Issues in the Optimization of Parallel Programs. In *International Conference on Parallel Processing - Vol II*, 1990.

[16] S. P. Midkiff, D. Padua, and R. G. Cytron. Compiling Programs with User Parallelism. In *Languages and Compilers for Parallel Computing*, 1990.

[17] A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, March 1994.

[18] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2), April 1988.

[19] J. P. Singh, W. D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News*, March 1992.

[20] The SPARC Architecture Manual: Version 8. Sparc International, Inc., 1992.

[21] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, May 1992.