Analyses and Optimizations for Shared Address Space Programs

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, called *cycle analysis*, is based on work by Shasha and Snir and checks for cycles among interfering accesses. We improve the accuracy of their analysis by using additional information from *synchronization analysis*, which handles post-wait synchronization, barriers, and locks. We also make the analysis efficient by exploiting the common code image property of SPMD programs.

We demonstrate the use of this analysis by optimizing remote access on distributed memory machines by automatically transforming programs written in a conventional shared memory style into a Split-C program, which has primitives for non-blocking memory operations and one-way communication. 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. Even larger benefits can be expected on machines with higher communication latency relative to processor speed.

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 [16]. To guarantee sequential consistency under reordering transformations, a new type of analysis called *cycle detection* is required [20].

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 the read of `Flag` returns 1 and the read of `Data` returns 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, since a superscalar may issue an instruction as soon as all its operands are available, 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 preceding write operations. In fact, on the SuperSparcs [22] 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

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense under contracts DABT63-92-C-0026 and F0602-95-C-0136, 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.

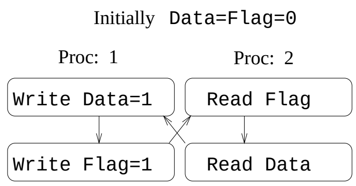

Figure 1: If the read of *Flag* returns 1, the read of *Data* should see the new value.

congestion. Also, two accesses sent to two different processors may be handled out of order, since network latencies may vary. Fortunately, machines 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 explicitly inserted by the compiler or the programmer. 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 can be determined 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. Midkiff and Padua [16] describe eleven instances where a uniprocessor compiler would generate either incorrect or inefficient code.

In this paper, 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 [3][23][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 [18] and *T [2], with their low overheads for communication startup.

	CM-5	T3D	DASH
Remote Access	400	85	110
Local Access (Cache miss)	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* [7]. In a split-phase operation, the initiation of an access is separated from its completion. 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, *conversion to one-way communication*, eliminates the acknowledgement traffic that is required to implement the *sync* operation for *put*s. A final optimization, *caching remote values*, eliminates 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 [20] and later extended by Midkiff, Padua, and Cytron to handle array indices [17]. In this paper, we show that by restricting attention to Single Program Multiple Data (SPMD) programs, one can significantly reduce the complexity of cycle detection. We also improve the accuracy of cycle detection by making use of the synchronization information in the program. Shasha and Snir's analysis, when applied to real applications, discovers a large 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 target programming language is described in section 2. We present basic terminology in section 3. In section 4, we present a brief summary of cycle detection and an efficient algorithm for detecting cycles in SPMD programs. 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 demonstrates the performance benefit 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 [7].

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 global pointers. The shared address space and SPMD parallelism are the only essential features of the source language. For the target language, we make use of Split-C's split-phase (or non-blocking) memory operations. Our compiler can also be viewed as a transformer that converts programs with shared memory accesses into programs that contain split-phase memory accesses.

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 `put`s and `get`s 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 reducing the number of packets but also reduce the processing time spent 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. Since split-phase operations are good for performance but hard to use, it should be the compiler's task to transform blocking operations into split-phase operations. Second, the global address space abstraction is provided in the source language only through a

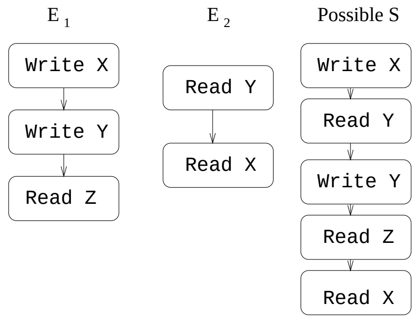

Figure 2: The figure shows a shared memory execution and one possible total order on operations. Executions in which the Write to Y appears to happen before the Write to X would not be legal.

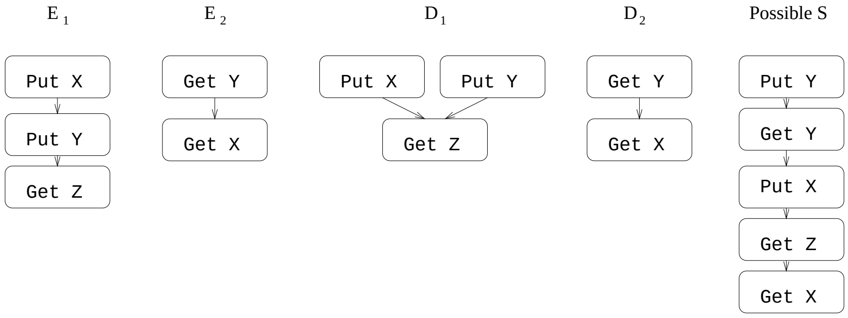

Figure 3: A weak memory execution with D containing some of the edges in E. Since the `puts` in E_1 are not ordered, they may execute in either order. However, an execution in which the `get` of Z appears to happen before either one of the `puts` would be illegal.

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 without requiring 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. 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.

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 [20]. 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 such that $D \subseteq S$ and 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.*

4 Compile-Time Analysis

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.

In this section, we present a brief summary of the cycle detection algorithm, which was originally proposed by Shasha and Snir. We also show that computing minimal delay sets for MIMD programs is an NP hard problem. However, by exploiting the common code image property of SPMD programs, we propose a polynomial time algorithm for SPMD programs. We conclude this section with a discussion on the minimality properties of the various algorithms.

4.1 Cycle Detection

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.

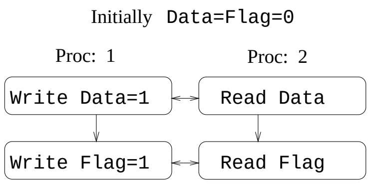

Figure 4: Example of a parallel program that does not require any delay constraints.

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 *[20]* $D_{S\&S}$ *is sufficient.*

The above definition of $D_{S\&S}$ suggests an obvious algorithm for generating correct code. For every pair of access statements, determine whether there exists a back-path in the control flow graph, and if such a back-path is present, introduce a fence to prohibit reordering of the two accesses. However, this formulation of $D_{S\&S}$ suffers from two drawbacks. First, as we will show, determining the existence of a back-path is NP hard in the number of program segments in a MIMD program. However, for SPMD programs, there exists a polynomial time algorithm for discovering back-paths. Second, the minimality result is not sufficient for real programs. 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. As we will discuss later in this section, 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.

4.2 Cycle Detection for MIMD Programs is NP Hard

Although Shasha and Snir do not specify the details of an algorithm for cycle detection, they claim [20] there is a polynomial time algorithm for detecting cycles in a program that "consists of a fixed number of serial program segments." In practice, one does not typically compile a program for a fixed number of processors: either the language contains constructs for dynamically creating parallel threads, or there is a single program that will be compiled for an arbitrary number of processors. We can show that if `PROCS` is taken as the problem size, the computation needed for the Shasha and Snir formulation is NP-complete[1].

Theorem 2 *Given a directed graph* G *with* n *vertices, we can construct a parallel program* P *for* n *processors such that there exists a Hamiltonian path in* G *if and only if there exists a simple cycle in* P*.*

4.3 Cycle Detection for SPMD Programs

We now present an efficient algorithm for computing the minimum delay set in an SPMD program. The algorithm is based on the idea of back-paths, but uses only two copies of the SPMD code, rather than one for each processor. The algorithm eliminates the condition that a back-path must pass through each program segment at most once. We first describe a transformation of the given control flow graph and then present an algorithm for detecting back-paths in the resulting graph. We show that the delay edges computed for the transformed graph are the same as in Shasha and Snir's approach.

[1] The construction and the proof for the following theorem is in the appendix.

Code:

```
while (turn != MYPROC);
numTrans++;
fund += giftAmt;
turn++;
```

Transformed Graph:

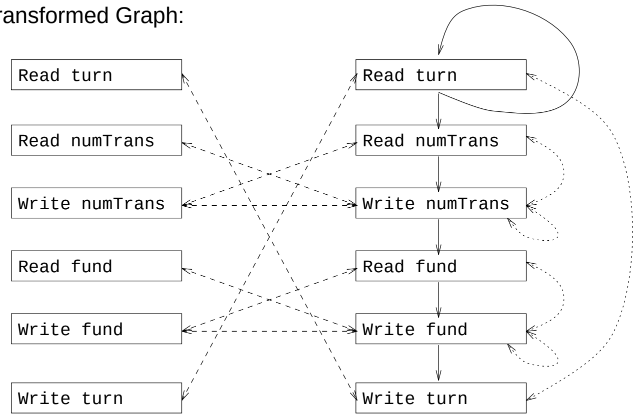

Figure 5: Cycle detection using two copies of the original program

In an SPMD program graph $P = \{P_1, ..., P_n\}$, all P_i are identical. Let V be the set of access statements in some P_i. We define a conflict set C_{SPMD} as pairs of access statements that belong to V such that both statements access the same variable and at least one of them is a **write**. The conflict edges are bi-directional, so we write (u, v) for the pair of edges $[u, v]$ and $[v, u]$.

For a given program graph (P, V), we generate a graph P_{SPMD} with nodes V_{SPMD} and edges E_{SPMD}, defined as follows. V_{SPMD} is two copies of the accesses in V, which we label L and R for left and right.

$$\begin{aligned}
V_{SPMD} &= \{v_L,\ v_R |\ v \in V\} \\
T_1 &= \{(u_L, v_R),\ (v_L, u_R) \mid (u, v) \in C_{SPMD}\} \\
T_2 &= \{(u_R, v_R) \mid (u, v) \in C_{SPMD}\} \\
T_3 &= \{(u_R, v_R) \mid [u, v] \in P\} \\
E_{SPMD} &= T_1 \cup T_2 \cup T_3
\end{aligned}$$

This transformed graph has two copies of the original program. A backpath will have endpoints in the left part of P_{SPMD} and internal path nodes in the right part. The T_1 edges connect the left and right nodes. The T_2 edges are conflict edges between right nodes. The T_3 edges are program edges that link the right nodes. The left nodes have no internal edges. Therefore, a path from v_L to u_L is composed of a T_1 edge, followed by a series of T_2 and T_3 edges and terminated with a T_1 edge. Figure 5 illustrates the construction for a simple program.

For every edge $[u_L, v_L] \in P$, we check whether there exists a path from v_L to u_L in the graph P_{SPMD}. We compute the set D_{SPMD} that consists of all edges $[u, v]$ having a path from v_L to u_L. Our algorithm runs in polynomial time: if m is the number of statements in the program that initiate remote accesses, the delay set can be computed in $O(m^3)$ time.

4.4 Minimality of D_{SPMD}

Shasha and Snir prove that their algorithm computes a delay set that is, in some sense, minimal. In this section we compare the delays sets computed by both algorithms for their minimality properties.

There are three potential sources of inaccuracy in the analyses that can lead to a non-minimal delay set: 1) the programs are assumed to be free of aliases, and if the analysis instead used a conservative approximation of aliases, spurious conflicts could be detected; 2) control flow information is ignored in the analysis; 3) the backpath may be longer than the total number of processors. Both our algorithm and theirs assume that there are no aliases to variables and, up to this point, do not include control flow information in the analysis. In the next section we will address the issue of control flow for the special cases of synchronization constructs, but without this, the only difference between the delay sets arises when there are long cycles.

We define the *length* of a backpath as the number of conflict edges in the path and give the following results.

Theorem 3 *Given an SPMD program for which the longest backpath P_{SPMD} is less than or equal to $PROCS$, $D_{SPMD} = D_{S\&S}$.*
Proof: *Straightforward from the construction of P_{SPMD}.*

Our algorithm is correct regardless of the assumption on the longest backpath. To see why ours is more conservative in the (probably rare) case in which the program contains a long backpath, consider such a program. Our algorithm, as described, will compute a delay set for an arbitrary number of processors. If a program with a backpath of length n is run on $PROCS < n$ processors, the execution order identified by that backpath has insufficient number of processors to actually take place. Thus, the delay edge added for that backpath is unnecessary.

Even this difference between the algorithms is not fundamental; if the value of $PROCS$ is known at compile time, as Shasha and Snir assume, our backpath detection algorithm can search for backpaths shorter than $PROCS$, which will be exactly those produced using cycle detection on $PROCS$ copies of the program. The cycle detection analysis, as proposed by Shasha and Snir, produces a minimal delay set if the program segments comprise of straight-line code. Modulo the requirement of knowing the number of processors at compile-time, our algorithm observes the same minimality property.

There is one more substantial difference between the algorithms: we analyze SPMD code, while Shasha and Snir consider MIMD code. Given an MIMD program with $PROCS$ different programs, we can convert it into an SPMD program with a branch for processor number around each program segment. Our analysis will run in polynomial time on this transformed MIMD program while theirs will be exponential on the original program. Assuming one starts with a branch-free MIMD program, their analysis would be minimal but ours would not, because of the branches introduced in the conversion process. In practice, programmers rarely write programs with $PROCS$ different pieces of code, so the loss of minimality relative to practicality of the analysis is a good trade-off to make.

In summary, the same theoretical results on minimality can be obtained for both algorithms, assuming the programs are branch-free and the number of processors is known at compile-time. As we will show in section 8, this theoretical result is not very useful in practice, since without control flow information both algorithms find essentially universal delay sets–every pair of accesses have a delay edge and there are no opportunities for optimizations in the example programs. We address this pragmatic problem in the next section by considering synchronization analyses, which are special cases of control flow analysis for explicit synchronization primitives.

5 Using Synchronization Information

The cycle detection algorithm 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 6 shows two program segments that access the variables X and Y. The delay set $D_{S\&S}$ contains edges between both pairs of accesses to X and Y, thereby prohibiting any overlap. However, if synchronization is taken into account, the delays between the reads and writes are seen to be unnecessary, as long as 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 also describe runtime techniques 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. Note that if we have accurate control flow analysis, then the synchronization analysis would fall out (for example, detect that a loop

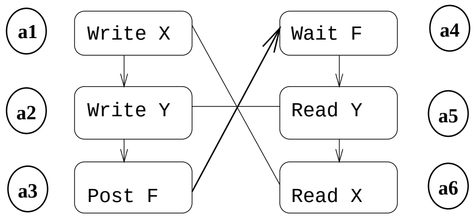

Figure 6: The post-wait synchronization enables the ordering of conflict edges, which results in fewer back-paths.

is actually a spin-wait loop). Since such an analysis is too hard, even undecidable in the general case, we provide synchronization primitives as language constructs and analyze their usage to obtain precedence and mutual exclusion information.

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 synchronization creates a strict precedence between the operations that are forced to execute before the `post` and the operations that are delayed to execute 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. Matching post-wait pairs are one example of pairs in R.

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 6 where post and wait are treated as conflicting accesses. $D_{S\&S}$ 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 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. It is sufficient to use the smaller delay set $\{[a_2, a_3], [a_1, a_3], [a_4, a_5], [a_4, a_6]\}$, because taken with the precedence ordering $[a_3, a_4]$, the other conflict edges $[a_1, a_6]$ and $[a_2, a_5]$ are ordered by transitivity, thus destroying the remaining back-paths.

As this example illustrates, we will compute the delay set and precedence relation 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 discovering other precedence constraints in the program. Note that a synchronization construct will not result in a fence for every outstanding memory operation, but only for those accesses that share a back-path with the synchronization operation.

The process of pruning back-paths 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 7, there are simple paths from a_3 to a_1 and from a_6 to a_4. Furthermore, since a_3 and a_4 are synchronization accesses, $[a_1, a_3]$ 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 searching for simple-paths to a_1.

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

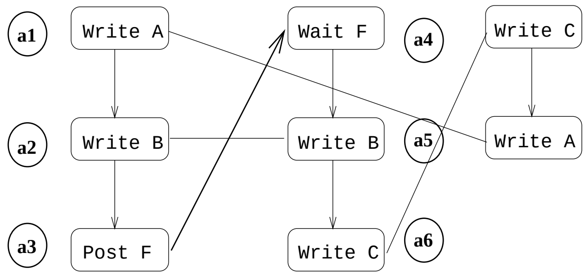

Figure 7: An example where synchronization analysis disqualifies certain accesses from appearing in back-paths.

Removal of a_6 destroys the simple-path from a_2 to a_1, which otherwise 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 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 original 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, T.
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 in T,
   (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. $D = D_1$.
7. For every pair $[a_i, a_j] \in P$, let $X = \{\ b \mid [a_i, b] \in R\}$. If $[a_i, a_j]$ has a back-path in $P \cup C_1 - X$, then $D = D \cup \{[a_i, a_j]\}$.

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 `barrier`s before we compute the delay set for the rest of the program.

```
void foo() {

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

Figure 8: Inaccuracies in analysis of barrier statements

To use barriers for computing precedence, we need to line up the barrier statements in the program text that will execute together at runtime. Since barriers can occur in branches and loops, the problem of lining up barriers is undecidable in general. Figure 8 shows a sample program where it is likely that all the processors will execute the final *barrier* statement at the same time, 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 on different processors. Rather than adding sophisticated analysis to line up barriers [11], we use a simple runtime solution that works well for many real programs. We add a runtime 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. If, however, adequate information is available at compile-time through analysis similar to the algorithm described in [11], the cycle detection analysis is applied on all the code fragments that could be executing concurrently on different processors.

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 l_1), and there are no intervening *unlock* l operations.
2. a dominates *unlock* l operation, which we will call u_1.
3. $[l_1, a] \in D_1$ and $[a, u_1] \in D_1$

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 (see Figure 9). The removal of these access statements 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

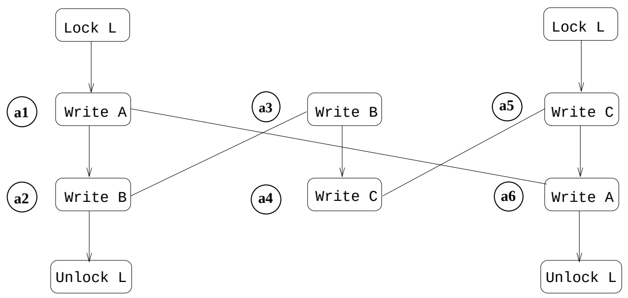

Figure 9: Eliminating accesses guarded by the same lock. While checking for back-paths between a_1 and a_2, accesses a_5 and a_6 are not considered. However, there does exist a valid back-path between accesses a_3 and a_4.

```
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 10: 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.

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 and from the semantics of lock/unlock operations that none of $b_1, b_2, ..., b_k$ can be guarded by the same lock and still appear in a violation sequence. To arrive at this property, we assume that locks cannot be indiscriminately tampered by the program. In particular, a lock cannot be unlocked by a processor that has not acquired the lock through a previous lock operation. Also, a lock variable cannot be accessed by standard read/write operations in the program. The first constraint can be checked at runtime whenever an unlock operation is executed. The second constraint is guaranteed by the type system of the language that prevents direct access to a lock.

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. The Cray T3D, the Stanford DASH, and the Wisconsin Wind Tunnel are examples of machines that have support for prefetching read and non-blocking write operations. Instead of generating code for a particular machine, the compiler produces Split-C code, and the Split-C compiler[14] is responsible for mapping the Split-C operations to the primitive operations supported by the target machine. In this section we describe the transformations introduced by our source level transformer.

For code generation, we use the control flow graph of the SPMD program, the delay set computed after synchronization analysis, and the *use-def* graph for each processor's variable access (obtained through standard sequential compiler analysis). 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 `put`s and `get`s issued by the 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 `put`s and `get`s, 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 initiation and completion events. 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.

6.1 Separating Initiation from Completion.

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 that 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}.

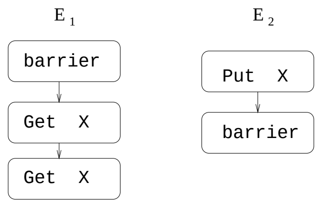

Figure 11: Barrier synchronization guarantees that X is read-only in the second phase.

(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 10, this may not be the best strategy, since it can lead to multiple synchronization points in the same control path. The duplication may result in unnecessary overhead, but is legal because `sync_cntr` operations are idempotent. 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 these cases. To choose the best placement, the compiler would require data on average executions of control paths and machine parameters.

6.2 One-Way Communication Optimization.

An optimization that is beneficial on many distributed memory machines is to transform two-way communication into one-way communication. The `put` operation generates an acknowledgement that signals the completion of the write on the remote processor. If there are no delay constraints that require the completion of a `put` access, the acknowledgement is not required, and therefore the `put` access can be replaced by the `store` operation. If the completion of a `put` access is required at a barrier synchronization point, we could still transform the `put` into a `store` as long as the barrier synchronization waits for all the stores across the machine to complete. However, this requires a stronger form of barrier synchronization. Fortunately, on most modern machines like the CM5 and the T3D, this form of barrier synchronization is relatively inexpensive, and the benefit obtained by eliminating the acknowledgement traffic outweighs the additional cost.

7 Eliminating Remote Accesses

Delay sets are necessary for any transformation that involves code motion including the motion of memory access initiation away from its completion. 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. The idea is to cache remote values that are being written or read multiple times in compiler generated local variables. We begin by examining when these optimizations are valid within a basic block. We then describe the correctness criteria for optimizing across basic blocks.

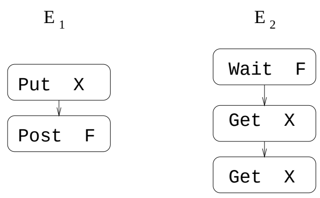

Figure 12: X can be cached by E_2 since the updates to X are guaranteed to be complete.

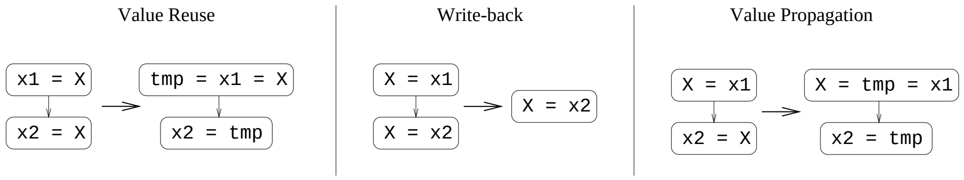

Figure 13: A set of transformations to eliminate remote accesses that are similar in spirit to standard uniprocessor optimizations like common subexpression elimination. Upper case letters denote global variables, and lower case ones local variables.

7.1 Optimizations within a Basic Block

When there are repeated accesses within a single basic block to the same variable X, and if X is not being written concurrently, the value of X can be cached. By virtue of synchronization analysis, it is easy to determine whether X is being concurrently written. Two examples are shown in Figures 11 and 12. 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 `get`s 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 `get`s and `put`s to X. Consequently, there is no delay edge between the two `get`s, and the second access can be eliminated.

Mutually exclusive access is sufficient but not necessary for elimination of repeated `get`s. It may be possible to reuse a previously read value even when there are intervening global accesses to the variable, 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, if there are no non-local accesses initiated between the two `get`s, 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 13. For simplicity, these optimizations 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. Note that since there could be intervening reads to the same variable that have been transformed into reads from local memory through the value propagation transformation, the first write access is not dead-code.

7.2 Optimizations across Basic Blocks

We cannot always apply these transformations to a pair of accesses that belong to different basic blocks. Here is a simple example that illustrates the dangers associated with applying the value reuse transformation across basic blocks.

```
x = X + 1;
...
while (X != 3);
```

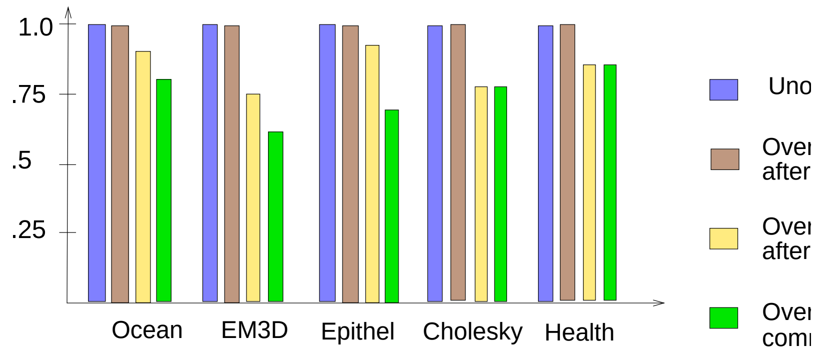

Figure 14: The figure gives the execution times, normalized so that the execution time of the code generated without cycle analysis is 1. Thus, a relative speed of 0.5 corresponds to a factor of 2 speedup.

Clearly, if we replace the remote reference to X in the spin-loop by a local reference to x, the program may stall. To address this problem, we classify variables into two types: data variables and control variables. Informally, the control variables are variables that affect the control flow of the program while the data variables do not. The following definition describes the criteria used for the classification.

1. Let $CV = \{v \mid v \text{ appears in a conditional}\}$.
2. Repeat until there is no change to CV:
   $CV = CV \cup \{\ u \mid \exists\ v \in CV \text{ such that } u \text{ is used to compute } v\}$.

The classification occurs after synchronization analysis, which partitions the program into different phases. Therefore, we might discover that certain variables that are used as control variables in certain phases of the program might be data variables in other phases. After classifying the variables, we apply the transformations that eliminate remote accesses only on the data variables.

8 Experimental Results

We quantify the benefits of our analysis 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[21]. **Ocean** studies the role of eddy and boundary currents in large-scale ocean movements. The primary data structures are grids that are updated using stencil-like computations. Distinct phases of the program synchronize using barriers.
EM3D: Em3d models the propagation of electromagnetic waves through objects in three dimensions [15][3] 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*. The alternate half time steps are separated using barrier synchronization.
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 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. Barrier synchronization is used frequently in this application.
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. Synchronization is enforced at the granularity of blocks using post-wait operations on flags.

[3] We use a version of Em3d that uses arrays for storing electric and magnetic field values and is written in a style that is more akin to Fortran and HPF programming models. Hence, the program has performance characteristics that are different from those of the pointer-based versions described in [7].

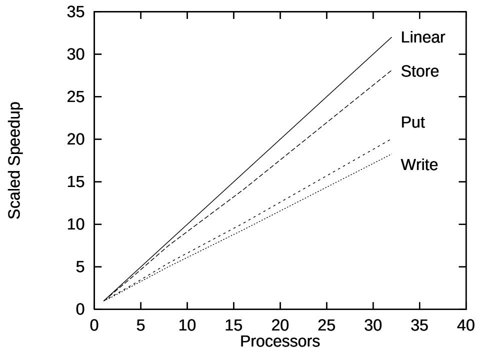

Figure 15: Speedup curves for the Epithelial application kernel with varying degrees of optimization. As expected, the optimized versions scale better with processors.

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. Communication occurs along the implicit tree structure imposed by the hierarchical system.

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. The measurements were done 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 14 gives the performance results of these experiments. The base program is the original unoptimized program. For this set of applications, cycle detection without synchronization analysis does not discover any opportunities for code motion. Our synchronization analysis results in much smaller delay sets, which in turn enables greater applicability of the message pipelining and one-way communication optimizations.

As a result of introducing the message pipelining optimizations, the speedup characteristics of the program changes. Figure 15 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.

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 [16] 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 [20], which was later extended by Midkiff et al [17] to handle array based accesses. Neither of these included implementations and the algorithms as presented were not practical because synchronization behavior is ignored. Related to our work is the AC compiler [6], which uses the non-blocking memory operations on the Cray T3D. However, since the AC compiler does not employ cycle detection, the compiled code could potentially generate executions that are not sequentially consistent.

In our analyses, we analyze the synchronization accesses in the program to obtain precedence and mutual exclusion information. 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 [5][9]. 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, 8]. 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 [10] 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 [4], and these optimizations have also been studied in the context of parallelizing compilers [19]. 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 a data parallel language to determine whether code-motion or pipelining optimizations are valid.

10 Conclusions

We presented analyses for explicitly parallel programs that communication through a global address space. The first analysis, cycle detection, improves on work by Shasha and Snir by making the analysis algorithm practical for SPMD programs. Whereas their framework leads to an exponential algorithm, our algorithm runs in polynomial time, and is more conservative only in rare instances that are unlikely to arise in practice. We implemented this analysis and showed that, using either formulation of the algorithm, cycle detection on its own is not sufficient to allow for any code motion or other optimizations in a set of simple application kernels.

We therefore extended the programming language with a set of synchronization primitives that are known to the compiler and incorporated synchronization information into the analysis. By treating synchronization operations as special accesses, we significantly improve cycle detection and can therefore perform transformations on the example applications.

The analyses are useful for a variety of optimizations. On shared memory parallel machines with some form of weakly consistent memory, our analysis can be used to automatically introduce memory fences or synchronization instructions to ensure that the programmer observes a sequentially consistent execution. On machines with explicit prefetch or non-blocking writes, it can be used to safely convert traditional loads and stores into these non-blocking counterparts. We use Split-C with its split-phase versions of read and write as an abstract target language. Our transformations convert Split-C programs with only traditional read and write operations into optimized programs that make use of the split-phase accesses.

The main optimization is masking latency of remote accesses by message pipelining, split-phase writes, and prefetching, split-phase reads. We also make use of an unacknowledged write operation to reduce network traffic, bulk messages to combine several reads or writes to the same processor, and common subexpression elimination which leads to caching of remote values. We have a prototype compiler that implements the first two optimizations and quantified the payoff on a set of application kernels. The performance improvements are as high as 35% on the CM-5. On machines with lower communication startup and longer relative latencies, the benefits for overlapping communication would be even higher. On machines with higher message startup, say from trapping to the kernel, the use of bulk operations or caching would be more important.

By providing a natural programming model based on shared memory, we allow programmers to focus on the higher-level problems of designing algorithms that are correct, have the right amount of parallelism, and are well load-balanced. The compiler is responsible for ensuring sequential consistency and performing communication optimizations, along with their traditional role of generating efficient code for each type of machine.

References

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

[2] B. S. Ang, Arvind, and D. Chiou. StarT the Next Generation: Integrating Global Caches and Dataflow Architecture. In *ISCA 1992 Dataflow Workshop*, 1992.

[3] 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.

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

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[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] S. Luna. Implementing an Efficient Global Memory Portability Layer on Distributed Memory Multiprocessors. Master's thesis, University of California, Berkeley, May 1994.

[15] 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.

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

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

[18] M. D. Noakes, D. A. Wallach, and W. J. Dally. The J-Machine Multicomputer: An Architectural Evaluation. In *20th International Symposium on Computer Architecture*, 1993.

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

[20] 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.

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

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

[23] 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.

A Back-path Recognition is NP Hard

In this section, we show that detecting back-paths is NP hard. We prove that any instance of the Hamiltonian Path recognition problem (HPR) can be reduced to a corresponding instance of the back-path recognition problem (BPR). Given a general graph (V, E), we construct a parallel program with program order P and conflict relation C, such that a Hamiltonian path exists in the original graph if and only if there exists a particular back-path.

Let the vertices in V be $v_1, \ldots, v_n$. The HPR problem is to determine whether there exists a simple path of length $n-1$ from v_1 to v_n. We construct a parallel program with $n-1$ threads that access a set of shared variables of the form v_j^i where $1 \leq i \leq n$ and $1 \leq j \leq n$. For every vertex in V, we construct a program thread

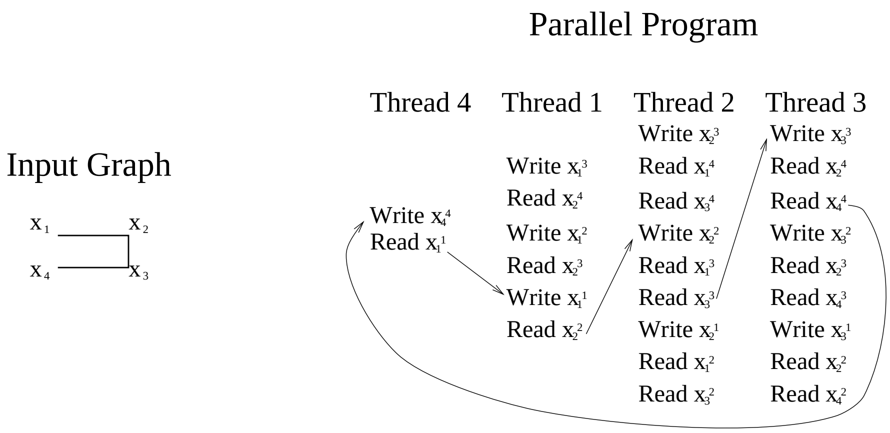

Figure 16: Constructing a parallel program for a given graph.

that initiates accesses to the shared variables in a particular order. The code for thread j is constructed based on the set of neighbors of vertex v_j in G. Let $w_1, \ldots, w_d$ be the neighbors of vertex v_j. We now define a macro called $CodeSeq$, which we will later use for fully specifying the code for thread j.

CodeSeq(j,i):
- *Write* v_j^i*;*
- *Read* w_1^{i+1}*;*
- *Read* w_2^{i+1}*;*
- ...
- *Read* w_d^{i+1}*;*

The code for thread j, P_j*, for* $1 \leq j < n$ *is defined as:*
- *CodeSeq(j, n-1);*
- *CodeSeq(j, n-2);*
- ...
- *CodeSeq(j, 1);*

P_n *is defined as: Write* v_n^n*; Read* v_1^1

We can show that a Hamiltonian path exists from v_1 to v_n if and only if there is a back-path between the two accesses initiated by thread P_n. Figure 16 illustrates this construction for a simple graph consisting of four vertices. The figure shows a Hamiltonian path from vertex x_1 to vertex x_4 in the graph and a corresponding back-path between the two accesses initiated by P_4.

Theorem: There exists a Hamiltonian path from v_1 to v_n in the graph G if and only if there exists a back-path from *Read* v_1^1 to *Write* v_n^n, which are accesses initiated by thread P_n.

Proof: We will first show that if there is a Hamiltonian path from v_1 to v_n in G, then a back-path exists. Let $u_1, \ldots, u_n$ be the Hamiltonian path from v_1 to v_n where u_1 is v_1 and u_n is v_n. Consider the access sequence *Read* u_1^1, *Write* u_1^1, *Read* u_2^2, *Write* u_2^2, *Read* u_3^3, *Write* u_3^3, ..., *Read* u_n^n, *Write* u_n^n. By construction, *(Read* u_i^i*, Write* u_i^i*)* $\in C$ and *(Write* u_i^i*, Read* u_{i+1}^{i+1}*)* $\in P_{u_i}$. Also, this access sequence visits each thread exactly once. Therefore, this path is a valid back-path from access *Read* v_1^1 to *Write* v_n^n.

We can also show that if a back-path exists between the two accesses, there is a Hamiltonian path in G that begins at vertex v_1 and ends at vertex v_n. To prove this assertion, we make use of the following properties of such a back-path.

1. All conflict edges that appear in the back-path are of the form *(Read x, Write x)*.
2. All program edges that appear in the back-path are of the form *(Write* v_j^k*, Read* v_l^{k+1}*)* where vertex v_l is adjacent to the vertex v_j in G.

3. The back-path visits exactly $n-1$ threads (excluding the thread P_n).

These properties follow from the structure of the parallel program and the definition of a back-path. The accesses initiated by a thread after a *Write* v_j^k access are either **reads** to variables of the form v_l^m with $m \leq k+1$ or **writes** to variables of the form v_l^m with $m \leq k$. The accesses initiated after a *Read* v_j^k access are either **reads** to variables of the form v_l^m with $m \leq k$ or **writes** to variables of the form v_l^m with $m < k$. Therefore, if there is a back-path from an access *Op1* v_q^r to an access *Op2* v_s^t, then the back-path must contain exactly $t-r$ P edges of the form *(Write* v_j^k*, Read* v_l^{k+1}*)*. Since $t=n$ and $r=1$, the back-path between accesses initiated by thread n has the above mentioned properties. These properties also imply that there is a Hamiltonian path from v_1 to v_n due to the constraint that a back-path does not visit a thread more than once.

Therefore, to solve the HPR problem on a graph, we can use the reduction specified in this section and formulate an equivalent BPR problem. Hence, BPR is NP hard. □