Optimizing Parallel SPMD Programs

Arvind Krishnamurthy and Katherine Yelick*
{arvindk,yelick}@cs.berkeley.edu

University of California at Berkeley

Abstract. We present compiler optimization techniques for explicitly parallel programs that communicate through a shared address space. The source programs are written in a single program multiple data (SPMD) style, and the machine target is a multiprocessor with physically distributed memory and hardware or software support for a single address space. Unlike sequential programs or data-parallel programs, SPMD programs require *cycle detection*, as defined by Shasha and Snir, to perform any kind of code motion on shared variable accesses. Cycle detection finds those accesses that, if reordered by either the hardware or software, could violate sequential consistency. We improve on Shasha and Snir's algorithm for cycle detection by providing a polynomial time algorithm for SPMD programs, whereas their formulation leads to an algorithm that is exponential in the number of processors. Once cycles and local dependencies have been computed, we perform optimizations to overlap communication and computation, change two-way communication into one-way communication, and apply scalar code optimizations. Using these optimizations, we improve the execution times of certain application kernels by about 20-50%.

1 Introduction

Optimizing explicitly parallel shared memory programs requires *cycle detection* analysis to ensure proper parallel program semantics. Consider the parallel program fragment in Figure 1. The program is indeterminate in that the read of `Y` may return either 0 or 1, and if it is 0, then the read to `X` may return either 0 or 1. However, if 1 has been read from `Y`, then 1 must be the result of the read from `X`. Intuitively, the parallel programmer relies on the notion of *sequential consistency*, which says the parallel execution must behave as if it is an interleaving of the sequences of memory operations from each of the processors [9]. As a more useful program example, assume that `X` is a data structure being produced by processor 2 and `Y` is a "presence" bit to denote that it has been produced.

If the two program fragments in Figure 1 were analyzed by a sequential compiler, it might determine that the reads or writes could be reordered, since there

* 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 Lawrence Livermore National Laboratory, by AT&T, and by the National Science Foundation (award numbers CDA-8722788 and CCR-9210260). The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

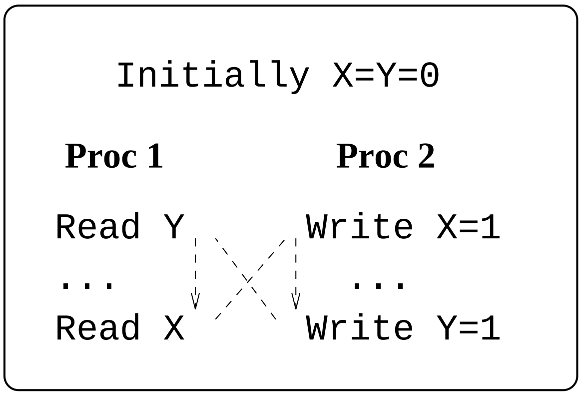

Fig. 1. If the read of Y returns 1, then the read of X must as well.

are no dependencies in either case. If either pair of the accesses is reordered, the execution in which `Y` is 1 and `X` is 0, might result. Alternatively, imagine that the program is executed in a distributed memory environment in which `X` is located on a third processor, and `Y` is located on processor 1. If the writes are non-blocking, i.e., the second one is initiated before the first is complete, then again sequential consistency could be violated. Similar scenarios exist for systems in which the network reorders messages or the compiler keeps copies of shared data in registers.

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 basic optimizations in shared memory programs, whether they 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. In the example, the semantics would determine that either the write or read to `X` (similarly `Y`) must appear in particular order.

In spite of the semantic simplicity of deterministic programming models, in practice, many applications are written in an explicitly parallel model. In this paper, we consider the common special case of Single Program Multiple Data (SPMD) programs, where multiple copies of a uniprocessor program communicate either through a shared address space [5] or through messages. This model is popular for writing libraries like ScaLapack and runtime systems for high level languages like HPF [6] and pC++ [3]. The uniprocessor compilers that analyze and compile the SPMD programs are ill-suited to the task, because they do not have information about the semantics of the communication and synchronization mechanisms. As a result, they 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 [11].

Cycle detection requires finding *conflicts* between concurrent code blocks, which are pairs of accesses to the same location from two different processors where at least one is a write. Cycle detection requires alias information, and is therefore similar to dependence analysis in parallelizing compilers. However, our goal in optimizing SPMD is more modest: the main source of parallelism has been exposed by the application programmer, so our job is to optimize the parallel code by making better use of local processor and network resources.

Cycle detection was first described by Shasha and Snir [15] and later extended by Midkiff, Padua, and Cytron to handle array indices. Their formulation gives an algorithm that is exponential in the number of processors and requires $PROCS$ (the number of processors) copies of the program. We give a polynomial time algorithm for the restricted version of the problem arising in SPMD programs.

Our target machine is a multiprocessor with physically distributed memory and hardware or software support for a global address space. A remote reference on such a machine has a long latency, from roughly 80 cycles on a Cray T3D [13] to 400 cycles on a CM5 using Active Messages [16]. 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.

Two important optimizations for these multiprocessors are communication overlap and the elimination of round-trip message traffic. 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 [5]. 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. In some cases global synchronization information can be used to eliminate the acknowledgement.

Other optimizations are also enabled by our analysis, but are not discussed in detail in this paper. These include improved scalar optimizations, making local cached copies of remote values, and storing a shared value in a register. The last two fall into the general class of optimizations that move values up the memory hierarchy to keep them closer to the processor [1].

The primary contribution of this paper is a new polynomial time algorithm for cycle detection in SPMD programs. This improves on the running time of the algorithm by Shasha and Snir, which is exponential in the number of processors. This analysis is used to perform optimizations such as message pipelining by using the portable Split-C runtime system as an example backend [5].

We describe the compilation and optimization problem for a simple shared memory language. The target language is Split-C, which is described in section 2. We present basic terminology needed for the analysis in section 3 and the analysis itself in section 5.1. Section 6 gives a basic code generation algorithm. Section 7 estimates the potential payoffs of our approach by optimizing a few application kernels. Related work is surveyed in section 8, and conclusions drawn in section 9.

2 The Target Language

Our target language is Split-C, a SPMD language for programming distributed memory machines [5]. Split-C provides a global address space through two mechanisms: a global distributed heap and distributed arrays. Values allocated in the global heap can be accessed by any processor using a *global pointer*; they can be accessed by the processor that owns that portion of the heap using a local pointer. Global pointers can be stored in data structures and passed in and out of functions using the same syntax as normal (local) pointers. Dereferencing a global pointer is more expensive than dereferencing a local one: it involves a check (in software on the CM5) to see whether the value is on the current processor or not, and if not, a message is sent to the owning processor. Split-C also provides a simple extension to the C array declaration to specify *spread arrays*, which are spread across the entire machine. Each index in the spread dimensions is placed on a different processor, mapping them linearly from zero to `PROCS`, wrapping as needed.

Our source language is almost a subset of Split-C, and has the essential components necessary to present our approach. For simplicity, the source language has shared global variables rather than heap-allocated objects. The shared variables have sequentially consistent semantics. A sample program is given in Figure 2. Shared variables are designated with the keyword `shared`: the variable `flag` and the array `result` are shared, while copies of `i` and `sum` exist on each processor. All processor begin executing the `main` procedure together, although they may execute different code in a data-dependent fashion. The pseudo-constant `PROCS` denotes the total number of processors and `MYPROC` denotes the executing processor's identity.

The source language can be trivially compiled into Split-C by allocating shared values in the distributed heap, and turning the shared variable accesses into global pointer dereferences. The shared array construct is directly mapped into a Split-C spread array declaration. The problem of choosing layouts to reduce communication is orthogonal; layout information could come from the programmer, as in HPF or Split-C [6, 5], or from a separate analysis phase [4].

The most important feature of the Split-C language is its support for split-phase memory operations. Given global pointers `src1` and `dest2`, and local values `src2` and `dest1` of the same type, the split-phase operations are expressed simply as:

```
dest1 := *src1;
*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

```
shared event flag;
shared int result[10];
main() {
   int i, sum = 0;
   if (MYPROC == 0) {
      for (i=0; i<10; i++) result[i] = i;
      post(flag);
   }
   wait(flag);
   for (i=0; i<10; i++) sum += result[i];
   barrier();
}
```

Fig. 2. Shared Memory SPMD Program

guaranteed to complete (the values of `dest1` and `dest2` are undefined) until after the `sync` statement.

This mechanism allows for communication overlap, but the `sync` construct provides less control than one might want, because it groups together 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. A family of `get_ctr` and `put_ctr` operations are provided to initiate accesses, along with a `sync_ctr` operation to wait for completion of an access. The signatures, in this case for double word values, are shown below:

```
void d_get_ctr (double *dest, double *global src, Counter *ctr);
void d_put_ctr (double *global dest, double *src, Counter *ctr);
void sync_ctr (Counter *ctr)
```

The computation from the earlier example can therefore be written:

```
d_get_ctr (&dest1, src1, ctr);
d_put_ctr (dest2, &src2, ctr);
/* Unrelated computation */
sync_ctr (ctr);
```

To separate the completion of the `get` from that of the `put`, two separate counters could be used. In general, this makes it possible to synchronize on a set of accesses, which is useful when two computations (for example two iterations of a loop) are overlapped.

Split-C also provides a `store` operation that 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.

Split-C runs on top of Active Messages on the CM5, and there are prototype implementations for the Paragon, SP-1, and a workstation network [10]. It defines a portability layer with fast, non-blocking remote accesses that, unlike large message passing systems, can be implemented without message buffering on both ends [16]. It blurs the distinction between machines with a hardware global address space and those without, making it a good choice for an abstract machine language.

3 Cycle Detection

A parallel execution on n processors is given by n sequences of instruction executions $P_1, \ldots, P_n$. We can ignore the local computation and local accesses, and therefore take each P_i to be the sequence of `reads` and `writes` to shared variables. Given an execution $P_i = a_1, \ldots, a_n$, we associate with P_i a graph, $(Vert, Edge)$, with vertices $Vert = \{a_1, \ldots, a_n\}$ and directed edges $Edge = \{[a_1, a_2], [a_2, a_3], \ldots, [a_{n-1}, a_n]\}$. The *program order*, P, is defined to be the union of these P_i's. A parallel execution will order accesses to shared variables. Such an ordering of accesses is *consistent* if the read/write behavior is observed, i.e., if reads always return value of the latest preceding write. We assume the following guarantee is made by the architecture:

System Contract 1 *Let V_v be the set of accesses initiated by the processors to the variable v. Then there exists a total order, E_v, of accesses in V_v that is consistent.*

On a distributed memory machine without hardware caching, each of the E_v's are totally ordered by the processor that owns the variable. With caching, it is the hardware designer's responsibility to ensure this semantics through a cache coherence protocol. The union of these E_v's defines the *execution order*, E, which is partial. Figure 3 illustrates these concepts using solid arrows for P edges and dashed arrows for E edges.

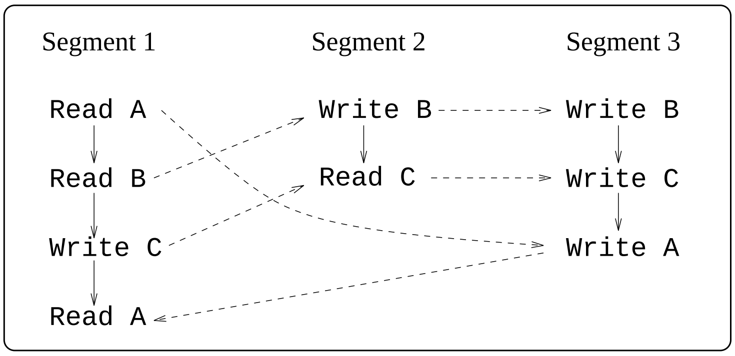

Fig. 3. P and E for a particular execution

The first part of our system contract is very weak—it says nothing about the order in which accesses take place relative to the program order. In a sequentially consistent machine, we would require that there be a single total order that is a superset of the program order, or equivalently, $P \cup E$ is acyclic.

Our target machine does not necessarily keep the accesses from a single program segment in order. The `put` and `get` operations specifically permit hardware reordering. To generate code for such a machine and allow for code motion during optimization, we need to determine which of the "happens after" paths in the program order P can be ignored, and which must be enforced. We augment our program to mark those orderings in P^+, the transitive closure of P, that must be observed. A *delay set*, D, for P is a subgraph of P^+. We now extend our system contract to make sure that the delay set is observed:

System Contract 2 *Given a program order P with delay set D, $D \cup E$ is acyclic.*

Note that if we take D to be P, this forces our machine to produce a sequentially consistent execution. Our goal during analysis, however, is to find a much smaller D that still ensures sequential consistency. We say that a delay set D is *sufficient* for program order P if, on any machine that satisfies the system contracts, $P \cup E$ is acyclic, i.e., the execution is sequentially consistent.

P and E are defined for a particular execution of a program, but we would like our analysis to work with a compile-time representation. We therefore approximate P and E (conservatively) by some superset of each. We approximate P by the control flow graph (CFG) of the program segment. Since there are loops in the control flow graph, CFG is no longer a total order on accesses. Also, the notion of an access is replaced by that of an *access instruction* that could initiate multiple accesses to a particular memory location during the execution of the program. At compile time, the runtime ordering of accesses to a variable is also not known. Hence, we approximate E by undirected versions of the E edges, which are called *conflict edges*, C. In general, the conflict edges may be further approximated if our alias analysis is imperfect. The only restriction necessary for correctness of our compiler is $E \subseteq C$ for any execution of the program.

4 Shasha and Snir's Algorithm

Shasha and Snir proved that there exists a minimum delay set, D, that can be defined by considering cycles in $P^+ \cup C$. We present their result in a slightly different form using the following kinds of paths.

Definition 1. A path $[a_1, \ldots, a_n] \in P^+ \cup C$ is *simple*, 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 access a_{i+1} is also in P_k, then for all other accesses a_j ($a_j \neq a_i$ and $a_j \neq a_{i+1}$), a_j is not in 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 $a_j \neq a_i$, a_j is not in P_k.

Thus, a simple path is one that visits each processor at most once, with at most two adjacent accesses on a processor. The following special case of simple paths defines the existence of a potential violation of sequential consistency.

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

Shasha and Snir use the notion of a *simple cycle*, which is given by an edge in P^+ along with its back-path. The two are clearly equivalent, but ours lends itself more naturally to an algorithm presentation. We define a particular delay set, $D_{S\&S}$, to be the those edges in P^+ possessing back-paths:

$$D_{S\&S} = \{[a_i, a_j] \in P^+ || [a_i, a_j] \text{ has a back-path in } P^+ \cup C\}$$

Shasha and Snir proved that if $D_{S\&S}$ is observed, the execution will be sequentially consistent:

Theorem 3. *[15] $D_{S\&S}$ is sufficient.*

They also proved that $D_{S\&S}$ in some sense characterizes the *minimal* delays: in any execution in which an edge in $D_{S\&S}$ executes out of order, there could be a violation of sequential consistency. This notion of minimality is not as strong as we would like, because it ignores the existence of control structures and synchronization that can prevent reorderings from happening even though the cycles exist statically.

5 Shasha and Snir's Algorithm is Exponential

Although Shasha and Snir do not specify the details of an algorithm for computing back-paths, they claim [15] there is a polynomial time algorithm for detection of backpaths 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[2].

Theorem 4. *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 iff there exists a simple cycle in P.*

5.1 Cycle Detection for SPMD Programs

In this section we 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. It eliminates the condition that a back-path must pass through each program segment at most once. For SPMD programs, the delay edges computed by our algorithm is still minimal.

[2] The construction and the proof for the following theorem is in [8]

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.

The Transformed Graph

In an SPMD program graph $P = \{P_1, ..., P_n\}$, all P_i are identical. Let V be the set of vertices in some P_i and E be the set of directed edges in P_i. The conflict edges are bi-directional, so we write (u, v) for the pair of edges $[u, v]$ and $[v, u]$. We define a conflict set, C_{SPMD} as the set of edges in P such that at least one of the accesses in the edge is a `write`.

We generate a new graph P_{SPMD} with nodes V_{SPMD} and edges E_{SPMD}, defines as follows. V_{SPMD} is two copies of the accesses in G, which we label L and R for left and right.

$$V_{SPMD} = \{< v, L >,\ < v, R > \ \mid v \in V\}$$
$$T_1 = \{(< u, L >, < v, R >),\ (< v, L >, < u, R >) \mid (u, v) \in C_{SPMD}\}$$
$$T_2 = \{(< u, R >, < v, R >) \mid (u, v) \in C_{SPMD}\}$$
$$T_3 = \{(< u, R >, < v, R >) \mid [u, v] \in P\}$$
$$E_{SPMD} = T_1 \cup T_2 \cup T_3$$

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 4 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 G'. We construct 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 n is the number of accesses in the program, the delay set can be computed in $O(n^3)$ time. Our algorithm has nearly the same sufficiency property as Shasha and Snir's, but is slightly more conservative if there are very long backpaths. The *length* of a backpath is defined as the number of conflict edges in the path.

Theorem 5. *Given an SPMD program for which the longest backpath P_{SPMD} is less than or equal to $PROCS$, $D_{SPMD} = D_{S\&S}$.*

Proof: *Omitted for brevity.*

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

Code:

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

Transformed Graph:

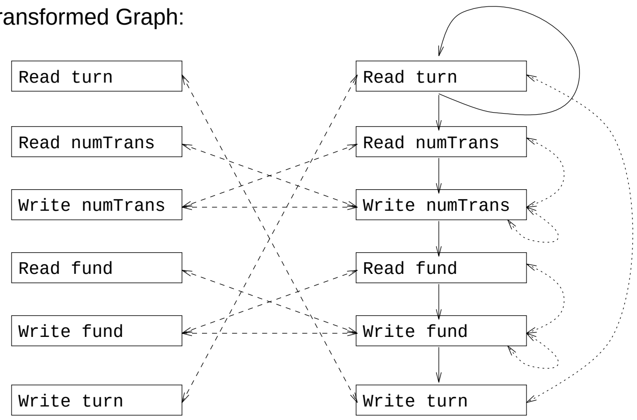

Fig. 4. Cycle detection using two copies of the original program

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, our backpath detection algorithm can search for backpaths shorter than $PROCS$.

6 Code Generation

In this section, we describe how the delay set information is used to generate Split-C style code and some of the trade-offs that arise during this process. The formulation of a delay set from the previous section is quite general, and can be used on a variety of memory models. Our presentation of Split-C code generation is simply one concrete example of such code generation.

The input to code generation is the control flow graph, the delay graph computed by the back-path recognition algorithm, and the *use-def* graph for local variables (as obtained through standard sequential compiler analysis). In code generation process, we need to satisfy the following constraints:

1. Delay constraints are observed.
2. Before every use of a local variable, the corresponding definition is complete.

Consider the program shown in figure 5. The solid line is the delay edge, and the dashed line is a def-use edge for the local variable x.

6.1 A Simple Code Generation Module

We describe a code generation strategy that is simple, but not optimal, and then describe some improvements for it.

```
x = X

y = 2

if (foo()) {

    y = x + 1

}

Z = 1
```

```
get(&x, &X, ctr1)

y = 2

if (foo()) {

   sync_ctr(ctr1);

   y = x + 1;
}

sync_ctr(ctr1);

z = 1
```

Fig. 5. Code Generation

A simple strategy is to generate a temporary counter for every remote access statement. One should bear in mind that an access statement might initiate multiple accesses during a program's execution. The counter can be used to ensure completion of all the accesses that have been initiated by the access statement. A Split-C code generated for a sample program is shown in figure 5. The counter variable is generated by the compiler. A split-phase get operation is initiated to fetch the value of X into the local variable x. A later `sync_ctr` operation on $ctr1$ ensures completion of all accesses initiated by the access statement.

The `sync_ctr` operation waits until the accesses are complete by waiting for the counter value to be reset. A property that makes code generation easy is that a `sync_ctr` operation behaves like a *null* operation if the program has already executed a `sync_ctr` on the same counter. In other words, a particular control path through the program can encounter multiple `sync_ctr` operations on the same counter. This suggests the following simple scheme for code generation.

Let a be an access statement in the program. Let $[a, b_1]$, $[a, b_2]$,..., $[a, b_k]$ be the set of delay constraints on this statement, and if a is a remote read operation, let $[a, c_1]$, $[a, c_2]$,..., $[a, c_l]$ be the set of def-use edges for the local variable being defined by the statement. The compiler converts a into a split-phase operation, and inserts a `sync_ctr` operation just in front of the access statements b_1, b_2,..., b_k, c_1, c_2,..., c_l. If, however, a `write` access does not have any delay constraints, we transform the `write` access into a `store` access, which is more efficient since it avoids acknowledging the completion of the access.

6.2 Pragmatics of Code Generation

The primary drawback of the simple code generation algorithm is the excessive use of the `sync_ctr` operation. Certain obvious improvements can be made to the simple scheme. However, it is not clear whether an optimal compile-time technique exists for code generation. As we will discover in this section, the code generation problem is similar in spirit to other compile-time techniques that need profiling information to generate near-optimal code.

Even though correctness of the program's execution is not violated by introducing extra sync_ctr operations, we would like to minimize their use since there is a cost attached to executing a sync_ctr operation. The first step is to reduce the number of program points at which sync_ctr operations are introduced.

Here is the modified algorithm for introducing sync_ctr operations:

1. Every remote access operation a is split into two operations: the corresponding split-phase initiate and a sync_ctr operation.
2. Let s be the sync_ctr operation associated with the split-phase initiate statement i. Rules are used for propagating s through the control flow graph in order to increase the number of instructions between i and s.
   (a) If s is ahead of b in a basic block in the control flow graph and if there are no delay or def-use constraints of the form $[i, b]$, then move s past b. If there are delay or def-use constraints of the form $[i, b]$, s comes to a halt in front of b.
   (b) If s is at the end of a basic block, propagate s to all the successors of the basic block, and continue the motion of the different copies of s.
   (c) If s is ahead of another copy of s, merge the two s operations into a single s operation.

This algorithm propagates the sync_ctr operations as far away from the initiation as possible. Also, if the access a is constrained to complete before the set of access statements b_1, b_2,..., b_k and if for some statement b_l there is no possible flow of control that hits b_l without encountering one of the other b_i statements, then the algorithm does not introduce a sync_ctr operation ahead of b_l. The simple algorithm would have incurred the penalty of an extra sync_ctr operation.

However, the algorithm still suffers from two drawbacks. First, there could be still certain control paths that execute more than one sync_ctr operation (as in figure 5). Second, if there is a delay constraint in which the initiation and the sync_ctr are nested within different conditionals and loops, our algorithm could execute unnecessary sync_ctr operations.

For example, given a delay constraint $[a, b]$ where b appears inside a loop, but a does not, we would not want to introduce a sync_ctr operation inside the loop since that would require the operation to be executed as many times as the loop would be executed. All but the first sync_ctr operation would be redundant. To avoid the cost of unnecessary sync operations, we could employ a loop-unrolling technique. We could separate the first iteration of the loop from the other iterations and introduce the sync_ctr operation only in the code for the first iteration.

The opposite problem occurs for a delay $[a, b]$ where a appears inside a conditional, but b does not. It is not clear where the sync_ctr operation should be introduced to ensure optimal performance. If we have the sync_ctr operation just ahead of b, we could suffer the penalty of executing the operation even when a a access had not been executed. Note that this does not affect the correctness of the code due to the nature of Split-C counters. On the other hand, if we introduce the sync_ctr operation at the end of the conditional containing a, we might be hiding only part of the latency by prematurely waiting for its completion. Static analysis cannot help in choosing between the two alternatives. Relative costs of remote accesses and local memory operations (for updating counters) could be used as an heuristic for code generation.

7 Potential Benefits

We quantify the benefits of our approach by studying the effect of the optimizations on a set of computational kernels. The four applications in our benchmark suite:

1. **FFT:** Computing the fast-fourier transform.
2. **Stencil:** 4-point stencil computation on a regular grid.
3. **Cg:** Computing the conjugant gradient of a sparse matrix.
4. **Em3d:** Solving Maxwell's equations on an irregular grid.

The prototype compiler automatically introduces the message pipelining and one-way communications optimizations for FFT and $Stencil$. For Cg and $Em3d$, the loops appearing in the benchmarks were manually unrolled before invoking the compiler[3]. The execution times of these applications were improved by 20-50% through message-pipelining and one-way communication optimizations. These were measured on the CM5 multiprocessor. The relative speedups should be even higher on machines with lower communication startup costs or longer relative latencies (when the fraction of the latency that can be overlapped is higher).

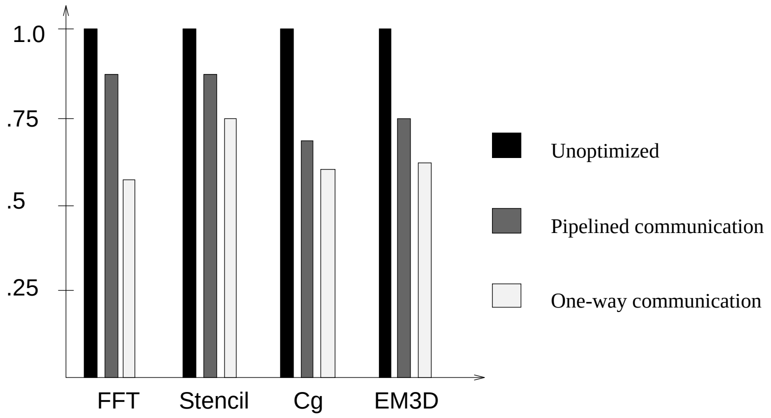

Fig. 6. Normalized Execution Times

[3] The loop-unrolling transformation is done as a pre-processing transformation that enlarges the size of basic blocks, thereby increasing the scope of prefetching and pipelining.

Figure 6 gives the performance results of these experiments. The figure gives the execution times, normalized so that the unoptimized execution time is 1. Thus, a relative speed of 0.5 corresponds to a factor of 2 speedup. Other optimizations, such as caching remote values, are also enabled by our analysis, and result in additional performance improvements on some of these applications [8].

8 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[11] 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[15], which was later extended by Midkiff et al[12] to handle array based accesses. However, their analysis technique is computationally expensive even for programs with a small degree of parallelism since both the minimal cycle detection problem and the array subscript analysis problem have exponential running times. The algorithm presented in this paper for SPMD programs does not deal with array analysis, but we believe their techniques for handling array subscripts could be incorporated into our SPMD framework.

Compilers and runtime systems for data parallel languages like HPF and Fortran-D[7] implement message pipelining optimizations. The Parti runtime system and associated HPF compiler uses a combination of compiler and runtime analysis to generate code for overlapping communication, aggregating groups of messages, and other optimizations [2]. These optimizations have also been studied in the context of parallelizing compilers[14]. However, as discussed earlier, compiling data parallel programs is fundamentally different than compiling SPMD programs. First, 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 directed conflict edges. Standard data-dependence techniques can be used in data parallel language to determine whether code-motion or pipelining optimizations are valid.

9 Conclusions

We have presented analysis techniques and optimizations for SPMD programs on distributed memory multiprocessors. The potential payoff of a few of these optimizations is estimated using hand optimizations on a small set of applications. The performance improvements are as high as a factor of two on the CM5, with even better performance expected on future architectures with lower communication startup.

The new form of analysis that is needed for explicitly parallel programs in a general (not data-parallel) execution model, is cycle detection, as introduced by Shasha and Snir. We showed that their formulation of the analysis led to an NP-complete problem and, therefore, an algorithm that was exponential in the number of processors. Applied to an SPMD program, their algorithm relied on analyzing `PROCS` copies of the code. We improved on their basic algorithm by giving an alternate formulation that uses only two copies of the code and computes nearly the same set of cycles in polynomial time. Finally, we showed how to use this analysis to generate code for an abstract machine language, Split-C.

References

1. B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies, Oct. 1990.
2. H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory multiprocessors. *Concurrenty: Practice and Experience*, pages 159–178, June 1991.
3. F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Maloney, and B. Mohr. Implementing a parallel C++ runtime system for scalable parallel system. In *Supercomputing '93*, pages 588–597, Portland, Oregon, November 1993.
4. S. Chatterjee, J. Gilbert, R. Schreiber, and S.-H. Teng. Optimal evaluation of array expressions on massively parallel machines. In *Workshop on Languages, Compilers and Run-Time Environments for Distributed Memory Multiprocessors*, pages 68–71, 1993.
5. 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*, pages 262–273, Portland, Oregon, November 1993.
6. High Performance Fortran Forum. High Performance Fortran language specification version 1.0. Draft, Jan. 1993.
7. 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.
8. A. Krishnamurthy. Optimizing explicitly parallel programs. Technical Report CSD-94-835, University of California, Berkeley, September 1994.
9. L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.
10. S. Luna. Implementing an efficient global memory portability layer on distributed memory multiprocessors. Master's thesis, University of California, Berkeley, May 1994.
11. S. Midkiff and D. Padua. Issues in the optimization of parallel programs. In *International Conference on Parallel Processing - Vol II*, pages 105–113, 1990.
12. S. P. Midkiff, D. Padua, and R. G. Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402–422, 1990.
13. W. Oed. The Cray research massively processor system: T3D. Ftp from ftp.cray.com, Nov. 1993.
14. A. Rogers and K. Pingali. Compiling for distributed memory architectures. *IEEE Transactions on Parallel and Distributed Systems*, march 1994.
15. D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.
16. 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*, 1992.