Automatic Generation of DAG Parallelism

Ron Cytron*
Michael Hind†
Wilson Hsieh‡

1 Introduction

We present an algorithm for automatically generating
a nested, fork-join parallel program from a sequential
program represented in terms of control and data
dependences. This algorithm embodies two techniques
for dealing with data dependences: the first implicitly
satisfies such dependences by reducing parallelism, and
the second eliminates some dependences by introducing
private variables. This algorithm has been implemented
in the PTRAN system [ABC*87].

Previous work on automatic generation of par-
allelism focused on vectorization, relying on a data-
dependence-based and loop-based representation as
input [Kuc78] [Wol82] [AK87]. This work is well-
developed and includes automatic generation of syn-
chronization for loop iterations [Mid86]. Previous work
has applied privatization in the form of scalar expansion
for vectorization [Wol78]. Other work has fully renamed
a program so as to obviate privatization [CF87b].

Our work extends automatic parallelization in the
following ways:

- We use the notion of control dependence [FOW87]
  [CFR*89] as a basis for potential parallelism. Such
  analysis allows discovery of parallelism among
  statements well in excess of basic blocks, in
  terms of granularity and scope, yet avoids the
  redundancy introduced by trace-scheduling meth-
  ods [FERN84] [Nic86].

- Where necessary, we honor data dependences
  among statements by partially ordering statements

*IBM Research Division, T. J. Watson Research Center,
Yorktown Heights, NY 10598.

†Courant Institute, New York University, 251 Mercer St., New
York, NY 10012. Work supported by a grant from IBM.

‡Laboratory for Computer Science, Massachusetts Institute
of Technology, Cambridge, MA 02139. Work supported by the
National Science Foundation (NSF Fellowship).

whose concurrent execution initiates simultane-
ously and potentially results in the data dependent
statements' execution. The resulting program
has better synchronization behavior; moreover,
ordering one pair of such statements can satisfy
many dependences. Although this technique could
overly constrain concurrency, experiments have
shown that critical path time is not adversely
affected [BCF*88]. These issues are discussed in
Section 4. The resulting DAG of constraints re-
sults in increased concurrency over previous work,
which was restricted to a total order among such
statements [Hsi88] [BCFH89].

- This paper extends the notion of *shared* and *private*
  storage to accommodate nested processes with
  DAG constraints, where data dependences deter-
  mine how a variable should be logically shared or
  kept private with respect to processes that reference
  that variable.

- We formalize the notion of privatization through
  a data flow framework, obtaining a fast method
  of determining where privatization can improve
  concurrency.

Section 2 describes the input and output of our
algorithm in terms of an example that should motivate
this work. Our algorithm, presented in Section 3, relies
on two subalgorithms:

- Where statement concurrency is affected by a data
  dependence, the algorithm given in Section 4 deter-
  mines which statements must be partially ordered
  to satisfy the dependence.

- Reductions in concurrency may render some pri-
  vatization ineffective. After statements are par-
  tially ordered, the algorithm given in Section 5
  determines where privatization no longer obtains
  increased concurrency.

Section 6 contains experiments that measure the
effectiveness of our methods.

2 Background

In this section, we describe the input and output of
our algorithm. For output, we use a nested fork-join

parallel model which allows the concurrent execution of loop iterations and statements of a procedure. Such features are found in languages such as OCCAM [PM87] (a derivative of CSP) and Parallel Fortran [IBM88]. Our model also allows any process to declare for itself a private instance of any variable.

For input, we assume that a sequential procedure has been analyzed and its associated control and data dependence graphs have been constructed [ABC*87]. Such analysis may incorporate the effects of other procedures or any facts provided by an informed user; we assume that any such information is represented in the control and data dependence graphs.

2.1   Target Parallel Model

Our parallel model includes the ordinary sequential programming constructs prevalent in languages such as Fortran and Pascal. Such constructs imply sequential execution: a single instruction stream exists before, during, and after each instruction. For the purpose of introducing concurrency, we include two additional constructs that enable the execution of multiple concurrently-executed instruction streams: `doall` and `cobegin...coend`.

The `doall` construct allows its iterations to execute concurrently. The `cobegin...coend` construct is similar to an ordinary `begin...end` pair, except that statements nested immediately inside this construct execute concurrently. Sequential execution is achieved within a `cobegin...coend` scope by appropriately surrounding statements with a `begin...end` pair. We allow an extension of the typical `cobegin...coend` construct: A DAG of constraints can be specified among processes at the same nesting level within a `cobegin...coend` pair. We do not allow arbitrary constraints, for reasons discussed in Section 4. Expression of such parallelism is allowed in Parallel Fortran [IBM88].

```
cobegin
    begin(0)
        S_0
    end
    begin(1) after(0)
        S_1
        S_2
    end
    begin(2) after(0)
        S_3
        S_4
    end
    begin(3) after(1,2)
        S_5
    end
coend
```

Figure 1: Block-Style Parallelism

In Figure 1, statements S_1 and S_2 execute sequentially, as do statements S_3 and S_4. However, S_1 and S_2 may execute concurrently with S_3 and S_4. None of these statements executes until statement S_0 finishes, and statement S_5 is in a block that waits for the completion of both preceding `begin...end` blocks.

Such constraints are not easily expressible in many parallel programming languages. Such constraints could be implemented by synchronization; we specify such constraints by syntax only for reasons of clarity. Our purpose is to demonstrate how such constraints can be automatically determined at compile-time.

We also introduce a `private` statement, by which process-specific instances of variables can be declared for `doall` iterations or for a block of statements inside a `cobegin...coend`. When associated with a `doall`, a `private` statement effectively causes each iteration to access distinct instances of those variables mentioned in the `private` statement. Now consider a `cobegin...coend` pair that surrounds a collection of `begin...end` blocks. When a `private` statement is associated with a `begin...end` block, that block accesses a distinct instance of each variable declared private. Additionally, a private variable may be declared with the tag "`copy out`", which indicates that the value assigned by the parallel construct should persist on termination of the construct. The semantics of value copying for private variables are as follows:

1. Prior to any program-specified concurrent activity for the given construct, space is allocated for the private variables.

2. The parallel components of the construct can then execute concurrently.

3. Once all components have completed execution, all `copyout` operations are performed. For each `copyout` variable, only one process may provide a value.

These semantics avoid race conditions. For example, consider two blocks of a `cobegin...coend` construct, where the first block accesses the (by default) shared instance of `v`, and the second block declares a private instance of a variable `v`, tagged `copyout`. Without the above precautions, the `copyout` value of `v` could accidentally be accessed by the first block.

2.2   Control Dependence

During the execution of a sequential program, the results of some operations, such as branches, determine whether other statements will subsequently be executed. The control dependence graph [FOW87] summarizes those conditions that may affect a statement's execution. To the precision of flow (non-semantic) analysis, control dependences represent control flow relationships that must be respected by any execution of the program, whether parallel or sequential. By examining the control dependence graph, we can eliminate unnecessary sequencing and expose potential parallelism.

Consider the example shown in Figure 2. All of the labeled statements except S_1 and S_8 are control dependent on the branch taken by statement S_1, even though other branches intervene between statements S_7 and S_9. If the flow of control passes from S_1 to S_2, the execution of statements S_3, S_4, S_5, S_6, S_7, and S_9 is guaranteed.

Control dependence can be computed from the control flow graph of a program. Nodes of the control flow graph could be any form of basic block: statements,

```
(S₁)if (a)
    then
(S₂)    x = y
(S₃)    z = v
(S₄)    y = z + t
(S₅)    t = y
(S₆)    q = y
(S₇)    if (b)
        then
(S₈)        z = w
        endif
(S₉)    f = z
    endif
```

Figure 2: Original Source Program

maximal basic blocks, or any straight-line sequence. We make the following assumptions:

- a basic block corresponds to a statement of the program.

- the control flow graph contains a unique entry node *Entry* and a unique exit node *Exit*.

- the control flow graph is augmented with an edge from the *Entry* to the *Exit* node to obtain a singly *rooted* control dependence graph.

- if a node has multiple successors, then the edges from that node have distinct labels (such as "T" and "F").

Definition 1 Let $G_{cf} = (N, E_{cf})$ be a control flow graph. We say that node Z is *post-dominated* by node Y in G_{cf} if every directed path from Z to *Exit* contains Y, $Z \neq Y$, and $Y \neq Exit$.

The post-dominator graph is a tree, and we denote the post-dominator tree for the graph G_{cf} by PD_{cf}. Post-dominators can be computed by solving the *dominators* problem [LT79] over the reverse control flow graph.

Definition 2 The control dependence graph $G_{cd} = (N, E_{cd})$ has the same nodes as G_{cf}, with edges determined by the control dependence relation:
        X *cd* Y if

1. there exists a non-empty, directed path P in G_{cf} from X to Y with all nodes in P (except X and Y) post-dominated by Y and

2. X is not post-dominated by Y.

In other words there is some edge from X that definitely causes Y to execute, and there is also some path from X that avoids executing Y. We associate with this control dependence from X to Y the label on the control flow edge from X that causes Y to execute. Where necessary, we denote the edges of a control dependence graph as a triple: (X, Y, l), where l is the associated label. Two nodes X and Y are *identically control dependent* on a node P if both are control dependent on P and the control dependence from P to X has the same label as the control dependence from

P to Y. In other words, (P, X, l) and (P, Y, l) are both edges in the control dependence graph.

The control dependence relation can be computed as the solution to a flow problem over the control flow graph, as suggested by the above definition. In this paper, we rely on a more efficient algorithm that requires a single pass over the *post-dominator* tree of the control flow graph [CFR*89] [CF87a]. More formally,

$$G_{cd} = CD\_alg(E_{cf}, PD_{cf})$$

For sequencing and privatization, we use a subgraph of the control dependence graph that reflects only those control dependences relevant to statement parallelization.

Definition 3 The *forward control dependence subgraph* is $G_{fcd} = (N, E_{fcd})$ where $E_{fcd} \subseteq E_{cd}$ is computed as follows:

1. Compute PD_{cf}: the post-dominators of G_{cf}.

2. Construct $G_{fcf} = (N, E_{fcf})$, where

   $$E_{fcf} = E_{cf} - \{(X, Y) \mid Y \ dominates \ X \ in \ G_{cf}\}$$

   Thus, E_{fcf} contains all edges of E_{cf} except the back edges.

3. Compute $G_{fcd} = CD\_alg(E_{fcf}, PD_{cf})$. The edges E_{fcd} are thus determined by the control dependence algorithm [CF87a], using the control flow edges E_{fcf} and the post-dominator relation PD_{cf}.

Where unambiguous, we drop the subscripts of G_{fcd} in favor of $G = (N, E)$ for the forward control dependence graph.

Theorem 1 G_{fcd} *is a tree if* G_{cf} *is a structured control flow graph.*

Proof. See [Hsi88].

Although structured control flow graphs simplify the problems addressed in this paper, we assume the more general class of *reducible* control flow graphs.

Theorem 2 G_{fcd} *is acyclic if* G_{cf} *is reducible.*

Proof. Assume that G_{fcd} contains a path from node X to itself. By the definition of forward control dependence, there must be a path in G_{cf} from X to itself that contains no back edges. However, this would imply that G_{cf} is irreducible. □

We now prove two lemmas that will participate in proving the correctness of our algorithm. We assume *topo#* is a topological numbering of G_{fcf}; i.e., if there is an edge (X, Y) in E_{fcf} then $topo\#(Y) > topo\#(X)$.

Lemma 1 *If* A *is an ancestor of* B *in* G_{fcd}, $topo\#(B) > topo\#(A)$.

Proof. Since A is an ancestor of B in G_{fcd}, there is a path from A to B in G_{fcf}. Since $>$ is transitive, $topo\#(B) > topo\#(A)$ by induction on the length of the path. □

Lemma 2 *If* A *is an ancestor of* B *in* G_{fcd}, *and* C *post-dominates* A *in the control flow graph* G_{cf}, $topo\#(C) > topo\#(B) > topo\#(A)$.

Proof. By Lemma 1, $topo\#(B) > topo\#(A)$. We prove that C post-dominates B in G_{cf}, thus showing that $topo\#(C) > topo\#(B)$.

Assume that C does not post-dominate B in G_{cf}. Since A is an ancestor of B in G_{fcd}, there must exist a path from A to B in G_{fcf}. Because that path cannot contain any post-dominators of A, it cannot contain C. Since we assumed that C does not post-dominate B in G_{cf}, there must exist a path from B to *Exit* which does not contain C; thus, there exists a path from A to *Exit* which does not contain C. This contradicts the fact that C post-dominates A. $\square$

Ignoring data dependences, control dependence analysis of the program in Figure 2 results in the parallel program shown in Figure 3. Statements S_2–S_7 and S_9 execute concurrently if S_1 takes its "`then`" branch. While these statements are executing, statement S_7 may choose to execute S_8.

```
(S₁)if (a)
    then
        cobegin
(S₂)        x = y
(S₃)        z = v
(S₄)        y = z + t
(S₅)        t = y
(S₆)        q = y
(S₇)        if (b)
            then
(S₈)            z = w
            endif
(S₉)        f = z
        coend
    endif
```

Figure 3: Parallel Program that Ignores Data Dependences

2.3 Data Dependence

Whereas control dependences identify those conditions under which statements execute, data dependences establish an ordering between the statements and iterations of a program. Together, the data and control dependence graphs represent those constraints that must be satisfied by any transformation of the original sequential program. We avoid a full treatment of data dependences [Kuc78] in favor of considering those aspects of data dependence relevant to our algorithms.

We are primarily concerned with satisfying *loop-independent* dependences. The identification of such dependences is a natural consequence of data dependence analysis [BC86]. *Loop-carried* dependences, which exist between statements in different iterations of a loop, do not lead to DAG-constrained parallelism; they can be satisfied by serializing iterations of the loops that carry them.

Consider again the program shown in Figure 2. Analysis [BC86] results in the following dependences relevant to the parallelization of statements:

1. Statement S_3 computes a value for z that can be used by S_4 and S_9. S_4 computes a value for y that

is used by S_5 and S_6. S_8 computes a value for z that can be used by S_9.

2. Statement S_2 uses a value for y that is subsequently overwritten by S_4. S_4 uses values for t and z that can be overwritten by S_5 and S_8, respectively.

3. Statement S_3 computes a value for z that can be overwritten by S_8.

The first group of dependences represents value flow; such dependences cannot be eliminated. The remaining dependences are storage-related; privatization can sometimes eliminate such dependences.

We now see that the parallel program of Figure 3 is incorrect, because data dependent statements execute concurrently. The algorithm presented in Section 4 constrains concurrency to satisfy dependences.

Figure 4 shows the effects of satisfying all data dependences by reducing concurrency. Notice that the DAG constraints allow the concurrent execution of statements S_5 and S_6. In previous work based on the control dependence graph, all statements in this example would execute sequentially [BCFH89].

```
(S₁)if (a)
    then
        cobegin
        begin(2)
(S₂)        x = y
        end
        begin(3)
(S₃)        z = v
        end
        begin(4) after(2,3)
(S₄)        y = z + t
        end
        begin(5) after(4)
(S₅)        t = y
        end
        begin(6) after(4)
(S₆)        q = y
        end
        begin(7) after(3,4)
(S₇)        if (b)
            then
(S₈)            z = w
            endif
        end
        begin(9) after(3,7)
(S₉)        f = z
        end
        coend
    endif
```

Figure 4: Correct Parallel Program, No Privatization

Figure 5 shows the results of also performing privatization to the example in Figure 2. The difference between Figures 4 and 5 is that in Figure 5, y is privatized to eliminate the dependences from S_2 to S_4.

```
(S₁)if (a)
    then
        cobegin
        begin(2)
(S₂)        x = y
        end
        begin(3)
(S₃)        z = v
        end
        begin(4) after(3)
        private y
            cobegin
            begin(40)
(S₄)            y = z + t
            end
            begin(50) after (40)
(S₅)            t = y
            end
            begin(60) after (40)
(S₆)            q = y
            end
            coend
        end
        begin(7) after(3,4)
(S₇)        if (b)
            then
(S₈)            z = w
            endif
        end
        begin(9) after(3,7)
(S₉)        f = z
        end
        coend
    endif
```

Figure 5: Correct Parallel Program With Privatization

3 Algorithm

We now consider how to generate DAG parallelism from the control and data dependence graphs of a procedure. The algorithm is in the style of *data flow* algorithms, in that it initially assumes an incorrect, liberal solution and works toward a correct (perhaps conservative) solution.

Because privatization can increase parallelism, our algorithm initially assumes that we can profitably eliminate all data dependences by introducing private variables for every store. We then assess the correctness or benefits of the assumed privatization. The more immediate concern is when privatization cannot eliminate the dependence. For example, a flow dependence between two statements cannot be eliminated by privatization. When we cannot eliminate a dependence by performing the assumed privatization, we instead reduce parallelism by *sequencing* the appropriate statements. The secondary issue concerns the benefits of privatization. We consider privatization to eliminate a dependence between two statements S_i and S_j inappropriate if:

- Statements S_i and S_j execute sequentially anyway. Privatization does not affect concurrency between the statements, and so is useless. In the example

of Figure 5, the dependence between statements S_4 and S_5 (due to t) is not eliminated because the two statements are sequenced due to the flow dependence on y.

- The cost of managing privatized variables is excessive. In Figure 5, consider the variable z, which is defined by S_3 and S_8 such that either definition can reach the use of z at S_9. Because the definition at S_8 is conditional, a run-time test would be required to decide whether S_3 or S_8 provides the value for S_9.

Both of these issues are discussed further in Section 5. We emphasize that the sequencing due to a dependence may affect the usefulness of privatization throughout a procedure. The algorithm is outlined in Figure 6.

Initialization

1. Initialize all loops to `doall`.
2. For each node $i \in N$, initialize $SEQSET(i) = \{\}$. The set $SEQSET(i)$ identifies those identically control dependent nodes that are sequenced after node i. Initially, *all* identically control dependent nodes are assumed to execute concurrently as blocks of a `cobegin...coend`, so each $SEQSET$ is empty.

Computation For each data dependence $d = S_i \delta\ S_j$

1. if d is *loop-carried*, then change the appropriate loops from `doall` to sequential `do` loops [BCFH89].
2. if d is *loop-independent* and cannot be eliminated by privatization then
   (a) SEQUENCE(S_i, S_j)
   (b) Repeat until no change:
       i. change = false
       ii. For each data dependence $p = S_k \delta S_l$ if CHECK_PRIV(p) reneges privatization, then
           A. change = true
           B. SEQUENCE(S_k, S_l)

Figure 6: Main Algorithm

The simple form of our algorithm, as stated above, checks each dependence after any sequencing to determine if privatization benefits are lost. A more clever scheme, described in Section 5, limits the dependences that must be checked after sequencing.

4 Satisfying Data Dependences

We use the algorithm in this section to satisfy a loop-independent data dependence $A\ \delta\ B$. Such dependences relate nodes whose statements cannot execute concurrently. We reduce concurrency among statements by introducing DAG constraints to ensure that B does not execute before A. Although the data dependent

statements could themselves be constrained, we choose to impose constraints only between *identically control dependent* nodes. This greatly simplifies synchronization: Suppose that the data dependent statements were themselves synchronized, perhaps through P and V semaphores. The statements may be guarded by branches, such that the source of the dependence is skipped prior to executing the sink of the dependence. Generally, "fix-up" code would be required to manage the semaphores [BCF*89]. However, if synchronization is placed on identically control dependent statements, then either all associated synchronization instructions execute or none do. Although this approach can overly constrain concurrency, a preliminary experiment observed no significant adverse affect [BCF*88].

Consider a loop-independent dependence between two arbitrary nodes A and B of the control dependence graph. The algorithm in Section 3 assumes that all identically control dependent nodes can execute concurrently. Constraints must therefore be placed between any pair of identically control dependent nodes α and β that have A and B as children, respectively, to ensure that A executes before B:

$$\mathrm{SEQ}(A, B) =$$

$$\{(\alpha, \beta) \mid$$
α and β, $\alpha \neq \beta$, are identically control dependent on some Z in G_{fcd},
$\exists$ a path in G_{fcd} from α to A,
$\exists$ a path in G_{fcd} from β to B $\}$

In terms of the algorithm in Section 3, $\beta \in SEQ\_SET(\alpha)$ if $(\alpha, \beta) \in SEQ(A, B)$ for any nodes A and B.

Since G_{fcd} is a tree if G_{cf} is a structured control flow graph (Lemma 1), $SEQ(A, B)$ contains at most one pair of nodes in such graphs for any A and B. In general, where G_{fcd} is constructed from reducible control flow graphs, $SEQ(A, B)$ can contain more than one pair of nodes; nevertheless, our algorithm computes such information efficiently.

1. Input: A forward control dependence graph $G_{fcd} = (N, E_{fcd})$

2. for all $n \in N$ initialize $reach(n) = \{n\}$

3. for each node $n \in N$ in reverse topological order of G_{fcd} do

   for each node $x \in succ(n)$ do
   $\qquad reach(n) = reach(n) \cup reach(x)$

Figure 7: Algorithm to compute *reach* set

Before considering how to compute a SEQ set, we describe a preprocessing step that improves the efficiency of our algorithm. We compute the set $reach(n)$ for every $n \in N$, where a node x is in $reach(n)$ if and only if x is reachable from n:

$$reach(n) = \{x \in N \mid \exists \text{ a path in } G \text{ from } n \text{ to } x\}$$

The *reach* set can be formulated as a simple data flow problem (see Figure 7). For all $n \in N$, initialize $reach(n) = \{n\}$. Visit the nodes N in reverse topological order of G, adding $reach(x)$ to $reach(n)$ for every successor x of n. Since G is acyclic (by Theorem 2), we can solve this data flow problem in one iteration.

4.1 Sequencing algorithm

Given a data dependence, we must now determine which nodes to sequence in the forward control dependence graph $G_{fcd} = (N, E_{fcd})$. [1] Informally, we search the graph for identically control dependent nodes that could result in the concurrent execution of data dependent nodes. We use the *reach* sets to hasten this process.

The search is initiated by the algorithm shown in Figure 6, which determines that sequencing is either necessary or desirable. The search begins at a node (LCA) through which all paths must pass to reach either data dependent node. The algorithm then proceeds through the forward control dependence graph in search of identically control dependent siblings that can reach the data dependent nodes.

The *SEQUENCE* algorithm shown in Figure 8 computes (LCA) and initializes the search. The *Find_SEQ* algorithm shown in Figure 9 performs the actual search.

Invocation: SEQUENCE (A,B), where

$\qquad$ A and B are data dependent nodes whose concurrent execution should be prevented.

Initialize: All nodes are unvisited by Find_SEQ

Procedure:

1. Compute LCA: the closest dominator of nodes A and B in G_{fcd} For structured programs, LCA can be computed as the least common ancestor of A and B. Otherwise, the node corresponding to the innermost common loop that contains both A and B suffices.
2. Invoke Find_SEQ (LCA, A, B)

Figure 8: SEQUENCE algorithm

4.2 Correctness of Sequencing Algorithm

Theorem 3 *Find_SEQ will always terminate when called on a node in a forward control dependence graph G which is computed from a reducible control flow graph.*

Proof. Since G_{cf} is a reducible control flow graph, G_{fcd} is acyclic (by Lemma 2). Therefore, we cannot loop forever due to an infinite chain of recursive calls to Find_SEQ. Eventually, within a call to some node Find_SEQ(n), statement 2c will be executed, and node n will be marked. Since there are a finite number of nodes in G, Find_SEQ will eventually mark all of them, and must terminate. $\square$

---

[1]Recall that edges are triples, where the third component is the label associated with the control dependence.

Invocation: Find_SEQ(n, A, B), where

- The search will begin at node n.
- A and B are the data dependent nodes.

Procedure: mark each label l_i as not processed.

1. if n is already visited, then return
2. for each edge $(n, d_1, l) \in E_{fcd}$, where l is not processed

   (a) if $A \in reach(d_1)$ and $B \in reach(d_1)$ then compute Find_SEQ(d_1, A, B)

   (b) otherwise if there exists d_2 such that $(n, d_2, l) \in E_{fcd}$ and $B \in reach(d_2)$ then

   $$SEQSET(d_1) = SEQSET(d_1) \cup \{d_2\}$$

   (c) Mark d_1 as visited

Figure 9: Find_SEQ algorithm

The following theorem is central to the correctness of Find_SEQ.

Theorem 4 *Given a forward control dependence graph $G = (N, E)$ that is computed from a reducible control flow graph G_{cf}, a connected subgraph $G' = (N', E') \subseteq G$ is a tree if it contains the root (Entry), and all successors of any node $n \in N'$ are identically control dependent upon n.*

Proof. Assign a topological numbering $topo\#$ to the nodes in G_{fcf}.

Assume that G' is not a tree. There must exist a node $n \in N'$ which has two predecessors, $p_1, p_2 \in N'$. By the definition of G', $|SEQ(p_1, p_2)| \geq 1$. Choose some member of $SEQ(p_1, p_2)$ and call it (p_1', p_2').

By the definition of control dependence, either p_2' post-dominates p_1', or vice-versa. Without loss of generality, assume p_2' post-dominates p_1'. Since p_1' is an ancestor of n in $G' \subseteq G$, $topo\#(p_2') > topo\#(n) > topo\#(p_1')$ (by Lemma 2). However, p_2' is an ancestor of n in G', and $topo\#(n) > topo\#(p_2')$ (by Lemma 1), which is a contradiction. $\square$

This theorem shows that Find_SEQ is correct. When statement 2b is executed, there can be at most one node d_2 identically control dependent with d_1 that reaches y, so we do not have to search for any other node identically control dependent with d_1 that reaches y. Otherwise, Find_SEQ visits every node dominated by LCA. Since A and B are dominated by LCA, all members of $SEQ(A, B)$ are also pairs of nodes dominated by LCA, and we are guaranteed that SEQUENCE will find every pair in $SEQ(A, B)$.

In fact, the theorem also implies that we can speed up our algorithm by pruning the search tree. If d_1 reaches both x and y, there can be no other node identically control dependent with d_1 that can reach either x or y. Therefore, we do not always have to search through every edge from n.

Finally, we show that sequencing d_1 and d_2 according to their topological order in the control flow graph is correct.

Theorem 5 *If there is a loop-independent data dependence from a to b, then for every $(\alpha, \beta) \in SEQ(a, b)$, sequencing α before β is equivalent to sequencing the two according to some topological numbering, $topo\#$, of G_{fcf}.*

Proof. Since the dependence from a to b is loop-independent, there must be a path in G_{fcf} from a to b. α is an ancestor of a in G, so $topo\#(a) > topo\#(\alpha)$, and therefore $topo\#(b) > topo\#(\alpha)$. Since α and β must be identically control dependent upon some node z, one of the nodes must post-dominate the other. If α post-dominates β, then $topo\#(\alpha) > topo\#(b) > topo\#(\beta)$ (by Lemma 2), which is a contradiction. Therefore, β post-dominates α, and $topo\#(\beta) > topo\#(\alpha)$. $\square$

4.3 Complexity of Sequencing Algorithm

To compute *reach* for all $|N|$ nodes takes $|E|$ bit vector operations. This preprocessing step is executed only once, independent of the total number of data dependences in the program.

During the execution of SEQUENCE, each edge in E which is in the subgraph dominated by LCA will be traversed at least once. Since we store information at the nodes to prevent the re-traversal of edges, each control dependence edge is traversed exactly once, so running SEQUENCE takes $O(|E|)$ bit vector operations.

4.4 Sequencing Optimizations

Sequencing two nodes of the forward control dependence graph may effectively sequence other nodes as well. Two observations about the side-effects of sequencing are a direct result of the following theorem.

For every $X \in N$, let $DOM(X)$ correspond to the set of nodes in the forward control dependence graph G that are dominated by the node X.

Theorem 6 *Let A and B be any two nodes in G. If there is no path between A and B, then for all $A' \in DOM(A)$ and $B' \in DOM(B)$, $SEQ(A', B') = SEQ(A, B)$*

Proof. Assumme there is no path between nodes A and B in G. We will show that $SEQ(A', B') \supseteq SEQ(A, B)$ and $SEQ(A', B') \subseteq SEQ(A, B)$.

$\supseteq$: Let (α, β) be any member of $SEQ(A, B)$. By the definition of $SEQ(A, B)$, there is a path from α to A in G. As A dominates A', there is a path in G from A to A' and thus from α to A'. Similarly, there is a path in G from β to B'. Since $(\alpha, \beta) \in SEQ(A, B)$, α and β are distinct nodes which are identically control dependent on some Z in G. Therefore, $(\alpha, \beta) \in SEQ(A', B')$. $\square$

$\subseteq$: Let (α, β) be any member of $SEQ(A', B')$ and Z be the node that identically controls nodes α and β. It suffices to show that there is a path from α to A and from β to B. We first show that there is no path in G from A to Z, for if there were such a path, there would also be a path from *Entry* to β (and thus to B') which goes through A. As there is no path between A and B, there is a path

from $Entry$ to B' which does not go through B, contradicting the assumption that $B' \in Dom(B)$. Since there is no path from A to Z, there is a path from $Entry$ to α which does not go through A. As $\alpha \in SEQ(A', B')$, there is a path from α to A'. Since A dominates A', this path must go through A and thus there is a path from α to A. Similarly, there is a path from β to B. Therefore, (α, β) is in $SEQ(A, B)$. $\square$

This theorem allows us to make two observations. One deals with the side-effects of directly sequencing two nodes, while the other considers the consequences of satisfying a data dependence via sequencing. These observations are used when processing a new data dependence. If we can determine that a dependence between two nodes has already been satisfied by some prior sequencing, then no further processing is needed for this dependence.

Let $A \ \delta \ B$ be a data dependence and let (α, β) be a member of $SEQ(A, B)$.

- Since (α, β) is a SEQ pair, $SEQ(\alpha, \beta) = \{(\alpha, \beta)\}$. By Theorem 4, there is no path in G between α and β. By Theorem 6, for all $\alpha' \in DOM(\alpha)$ and $\beta' \in DOM(\beta)$, $SEQ(\alpha', \beta') = \{(\alpha, \beta)\}$. Thus, once α and β have been sequenced, all nodes that α dominates have effectively been sequenced with all nodes that β dominates.

- By Theorem 6, if there is no path between A and B in the forward control dependence graph G, then once the dependence $A \ \delta \ B$ has been processed, all nodes in G that are dominated by A have been effectively sequenced with those nodes that are dominated by B.

Although these two statements appear to be similar, they describe different side-effects and can in general be disjoint. Whenever the graphs rooted at α and β are trees, the first observation will include those pairs of sequenced nodes described in the second. However, if either graph is not a tree, the second observation will capture side-effects that the first could not. These observations allow the elimination of redundant processing and may reduce the actual running time of the algorithm.

5 Privatization

In this section we present an algorithm that determines where privatization increases parallelism by eliminating storage-related dependences. Although privatization comes in two flavors (`doall` privatization and `cobegin...coend` privatization), this section is concerned only with statement concurrency. Privatization with respect to loops has been previously examined [BCFH89].

Consider the benefits of privatization with respect to statement concurrency, where some statements may be sequenced by DAG constraints. The following two conditions may hold:

C1: If the affected nodes are already ordered by DAG constraints from flow dependences, then removing the storage-related dependence is of no benefit. Consider flow dependences from nodes X to Y and from Y to Z. Eliminating a storage-related dependence from X to Z would not increase program parallelism, and therefore is not performed.

C2: Privatization introduces process-specific instances of variables; such variables exist only for the duration of their allocating process. Consider a set of concurrently executing processes, where each process privatizes and computes a value for some variable x. If a value for x is required after the processes terminate, then some decision must be reached as to which process provides the surviving value. We consider privatization inefficient unless the process responsible for providing such a value can be determined at compile-time. Otherwise, a run-time recurrence solution is required, the expense of which may cancel the benefits of privatization.

If neither of the above conditions exists, then privatization will increase parallelism and is therefore performed.

The above criteria serve as a guideline for determining when the use of `cobegin...coend` privatization is beneficial. We will now describe how these guidelines can be incorporated into the framework described previously.

5.1 CHECK_PRIV

As in the algorithm of Section 4, `cobegin...coend` privatization is formulated as a path problem over the control dependence graph. We define the two sets,

$$PATH(Y) = \{X \in N \mid \exists \text{ a directed path of sequenced} \\ \text{nodes in } G \text{ from } Y \text{ to } X\}.$$

and its inverse,

$$RPATH(Y) = \{X \mid Y \in PATH(X)\}$$

We use these sets to determine how privatization would affect a storage-related dependence. As nodes are sequenced, each of these sets must be updated. Before we describe how to perform these updates, we show how these sets are used.

We initially set $PATH(X) = RPATH(X) = \{X\}$. Whenever the Find_SEQ algorithm (Figure 9) sequences node X with node Y, where X topologically precedes Y in the original program, we will perform the following:

1. For all $Z \in RPATH(X)$ do

   - $PATH(Z) = PATH(Z) \cup PATH(Y)$

2. For all $Z \in PATH(Y)$ do

   - $RPATH(Z) = RPATH(Z) \cup RPATH(X)$

We now describe how we use the $PATH$ set to categorize storage-related dependences. Let X and Y participate in a storage-related loop-independent data dependence. Without loss of generality, assume that X precedes Y in the topological order of the original program.

Whenever $Y \in PATH(X)$, condition $C1$ holds, and thus, nodes X and Y have already been effectively sequenced. Consider the flow dependence between statements S_4 and S_5 in Figure 2; this dependence forces

the statements to be sequenced. Since $S_5 \in PATH(S_4)$, there is no need to perform privatization between these statements.

The second reason for avoiding privatization concerns compile-time indeterminacy with respect to flow of values. The semantics of privatization require that all nodes that use a private variable be in the `begin...end` block that declares that variable. If it cannot be determined at compile-time whether a value corresponding to a private variable will flow to a use, privatization is not performed.

```
(S₁)if (a)
    then
        cobegin
        begin(2)
(S₂)        x = y
        end
        begin(3)
        private z
            cobegin
            begin(30)
(S₃)            z = v
            end
            begin(40) after(30)
            private y
                cobegin
                begin(400)
(S₄)                y = z + t
                end
                begin(500) after(400)
(S₅)                t = y
                end
                begin(600) after(400)
(S₆)                q = y
                end
                coend
            end
            coend
        end
        begin(7)
        private z
            cobegin
            begin(70)
(S₇)        if (b)
            then
(S₈)            z = w
            endif
        end
        begin(80) after(70)
(S₉)        f = z
        end
        coend
    coend
    endif
```

Figure 10: Privatization Producing an Incorrect Program

Consider the output dependence involving statements S_3 and S_8 of Figure 10. Privatizing z in these statements, requires a decision as to which statements should be included in the respective `begin...end` blocks. Since statement S_4 uses the private value of z of

Block 3, it is included in this block. However, since we are unable to determine at compile-time which value of z will be used by statement S_9, it is incorrect to include it in either Block 3 or 7. Privatizing cannot be used to eliminate the storage-related dependence between statements S_3 and S_8; such privatization would produce an incorrect program.

The correct version of Figure 10 is given in Figure 5. Since z is not privatized in statements S_3 and S_8, statement S_9 is ensured to get the correct value of z. However, the block containing statement S_8 must now wait until statements S_3 and S_4 have executed before it writes into variable z.

Previous work has described conditions that prevent a compile-time decision for `copy out` [BCFH89]. In our framework, those conditions can be characterized using the set $SEQSET$ set. For any pair of nodes, X and Y, the intersection of $SEQSET(X)$ and $SEQSET(Y)$ includes all the nodes which require values computed by both X and Y. In the example of Figure 10, the $SEQSET$s S_3 and S_8 are not disjoint: $SEQSET(S_3) = \{S_3, S_4, S_9\}$ and $SEQSET(S_8) = \{S_8, S_9\}$. Thus, privatization is not used; instead, the storage-related dependence is satisfied by sequencing the appropriate nodes.

If $Y \notin SEQSET(X)$ and $SEQSET(X) \cap SEQSET(Y) = \{\}$, the variable involved in the storage-related dependence is privatized. This can be seen in Figure 3, where y is privatized in block 4 to eliminate the anti- dependence between S_2 and S_4.

We can now specify the routine CHECK_PRIV mentioned in section 3. CHECK_PRIV, shown in Figure 11, is passed a data dependence and will determine if this dependence should no longer be privatized. The tests using $PATH$ and $SEQSET$ discussed in the previous section are performed in this routine. CHECK_PRIV will return true (privatization has been reneged) whenever condition 1 or 2 holds.

Invocation: CHECK_PRIV(p), where

- $p = S_i \delta S_j$ and S_i topologically precedes S_j

This routine will return true if the dependence that is passed should no longer be privatized. It will return false, otherwise.

Procedure: 1. If $S_j \in PATH(S_i)$ then
         return(true)
   2. If $SEQSET(S_i) \cap SEQSET(S_j) \neq \{\}$ then
         return(true)
   3. else
         return(false)

Figure 11: CHECK_PRIV Algorithm

5.2   A More Efficient Privatization Algorithm

Whenever two statements are sequenced by the algorithm in Figure 6, each data dependence is checked to ensure that privatization is still applicable; this process

is very expensive. This section describes an improved version of the Repeat-Until loop in Figure 6. In the improved algorithm, whenever two nodes X and Y are sequenced, only the data dependences that can be affected by the sequencing of X and Y are examined.

With each node X, we associate the set of nodes $PRIV(X)$ that participate in a privatization relation with X; these nodes are involved in an output dependence with X. Since nodes correspond to statements, and each statement can write to at most one variable, $|PRIV(X)| \leq 1$. We also assume the presence of a set $UD(X)$ which corresponds to all nodes involved in an anti-dependence with X, such that the store of the variable is done at the statement corresponding to node X.

We also define the following sets:

$$APV(Y) = \{X \in N \mid \exists \text{ a private variable at node } X \text{ and } Y \in PATH(X)\}$$

$$DPV(Y) = \{X \in N \mid \exists \text{ a private variable at node } X \text{ and } X \in PATH(Y)\}$$

The APV (Ancestor Private Variables) set of a node X contains all private variable nodes that are ancestors of X when considering sequenced edges. Similarly, the DPV (Descendant Private Variables) set corresponds to the private variable nodes that are descendants of X.

For all X in N, we initialize $APV(X) = DPV(X) = \{\}$. Whenever we privatize a variable at a node X we set $APV(X) = DPV(X) = \{X\}$. When a node X is sequenced with a node Y, $APV(Y)$ and $DPV(X)$ are updated in a similar manner to $PATH$ and $RPATH$ as illustrated below.

1. For all $Z \in RPATH(X)$ do
   - $DPV(Z) = DPV(Z) \cup DPV(Y)$

2. For all $Z \in PATH(Y)$ do
   - $APV(Z) = APV(Z) \cup APV(X)$

The sequencing of two nodes, X and Y, might require the elimination of previously defined privatizations. If $APV(X)$ and $DPV(Y)$ are not disjoint, there is a variable, v, that is currently being privatized by two nodes V_1 and V_2, where $V_1 \in RPATH(X)$ and $V_2 \in PATH(Y)$. The sequencing of X and Y will effectively sequence V_1 and V_2; therefore, the privatization is not required. On the other hand, if there is a $Z \in PRIV(X)$ such that $Y \in SEQSET(Z)$, privatizing nodes X and Z might result in the incorrect value flowing to node Y.

When a privatization is eliminated, the corresponding nodes are removed from the appropriate $PRIV$, APV and DPV sets and SEQUENCE is called to satisfy this storage-related dependence.

Figure 12 gives the optimized version of the computation phase of the general algorithm of Figure 6. A stack of nodes to be sequenced, representing a worklist, is maintained. Once a privatization has been reneged, the nodes involved must be sequenced, and are therefore pushed onto the stack. The while loop continues as long as there are nodes remaining on the stack.

When a pair of nodes is popped from the stack the test of statement iii (Figure 12) checks whether privatization has been reneged. If this test is true then RENEGE_PRIV (see Figure 13) is called to remove the

Computation For each data dependence $d = S_i \delta\ S_j$

1. if d is *loop-carried*, then change the appropriate loops from `doall` to sequential `do` loops [BCFH89].
2. if d is *loop-independent* and cannot be eliminated by privatization then
   - (a) PUSH(S_i, S_j)
   - (b) while not empty do
     - i. POP(S_i, S_j)
       Let v correspond to the variable that is assigned in statement S_j.
     - ii. SEQUENCE(S_i, S_j)
     - iii. If $APV(S_i) \cap DPV(S_j) \neq \{\}$ then
       Let $V_1 \in APV(S_i)$, $V_2 \in DPV(S_j)$ correspond to the nodes that privatize v.
       A. RENEGE_PRIV(V_1, V_2)
       B. For all $S_k \in UD(S_j)$ do
          PUSH(S_k, S_j)
     - iv. else if there exists a $S_k \in PRIV(S_i)$ such that $S_j \in SEQSET(S_k)$ then
       A. RENEGE_PRIV(S_i, S_k)
          Let S_l correspond to the node between S_i and S_k which is topologically larger.
       B. For all $S_z \in UD(S_l)$ do
          PUSH(S_z, S_l)

Figure 12: Improved Version of Computation Phase of General Algorithm

Invocation: RENEGE_PRIV(S_i, S_j), where

- S_i and S_j participate in an output dependence

This routine will renege the privatization of the variable between statements S_i and S_j.

Procedure:
1. $PRIV(S_i) = \{\}$
2. $PRIV(S_j) = \{\}$
3. For all $S_p \in RPATH(S_i)$ do
   $DPV(S_p) = DPV(S_p)$ - $\{V_1\}$
4. For all $S_p \in PATH(S_j)$ do
   $APV(S_p) = APV(S_p)$ - $\{V_2\}$

Figure 13: RENEGE_PRIV Algorithm

privatization of variables associated with nodes S_i and S_j. Since privatization removes storage-related dependences, we must now sequence to satisfy all dependences that were previously eliminated by privatization. This is done implicitly (via the stack) in statement iiiB.

If the first test is passed without any removal of privatization, the test at statement iv is performed. If it is true, RENEGE_PRIV is called to remove the privatization at statements S_i and S_k. As is the case with the test at statement iii, statements that require sequencing are pushed on to the stack.

The body of RENEGE_PRIV is given in Figure 13. The first two statements remove the private variables

from their $PRIV$ sets. Statements 3 and 4 update the DPV and APV sets, respectively, to reflect the removal of the private variable.

5.3 Complexity Analysis of Improved Privatization Algorithm

The stack of dependences to be sequenced drives the improved algorithm of Figure 12. For each pair of nodes popped from the stack, we perform the tests at statements iii and iv. The test at statement iii involves taking the intersection of two sets, $APV(S_i)$ and $DPV(S_j)$. This test will require $O(\max(|APV(S_i)|, |DPV(S_j)|))$ operations (assuming the sets are stored in a search tree). Similarly, the test at statement iv will require $O(|PRIV(S_i)|)$ operations. Since $|PRIV(S_i)| \leq 1$, the test at statement iv is a constant time operation.

The procedure $RENEGE\_PATH$ of Figure 13, requires $O(|RPATH(S_i)| + |PATH(S_j)|)$ operations to iterate through these two sets. The for loops at the statements iiiB and ivB require an iteration of a UD set. However, for each element of the UD set, we push a pair of nodes on to the stack to be sequenced. Since the push is a constant time operation, the cost of iterating though the set can be factored into the processing of the pair of nodes when they are popped from the stack. Therefore, the body of these tests requires $|RPATH(S_i)| + |PATH(S_j)|$ operations.

For each node that is sequenced, we perform $O(\max(|APV(S_i)|, |DPV(S_j)|) * (|RPATH(S_i)| + |PATH(S_j)|))$ operations. If we let Δ correspond to the number of data dependences in the original program, in the worst case, the number of nodes to be sequenced will be $O(|\Delta|)$. This will give an overall complexity of $O(|\Delta| * (\max(|APV(S_i)|, |DPV(S_j)|) * (|RPATH(S_i)| + |PATH(S_j)|))$ for the while loop of the new algorithm.

Note that the APV, DPV, $PATH$, and $RPATH$ sets are a function of the amount of sequencing. While in general the size of these sets can be as large as $|N|$, the experiments discussed in Section 6 indicate that, in practice, we can expect these sets to be small compared to $|N|$. Thus, we expect a running time close to $O(|\Delta|)$.

5.4 Comparison of Privatization Algorithms

Consider the original algorithm of Figure 6. Statement ii iterates through the set of data dependences, Δ. In the worst case, during each iteration one instance of privatization can be reneged, forcing another iteration through Δ. The two tests in CHECK_PRIV (Figure 11) require constant time and $O(\max(|SEQSET(S_i)|, |SEQSET(S_j)|))$, respectively. As is the case with the $PATH$ set these sets can contain up to $|N|$ nodes, but experiments suggest that the actual sizes are relatively small, in practice. Thus, the complexity of the repeat-until loop of Figure 6 is $O(|\Delta|^2 \cdot (\max(|SEQSET(S_i)|, |SEQSET(S_j)|)))$.

Since $\max(|APV(S_i)|, |DPV(S_j)|) \cdot (|RPATH(S_i)| + |PATH(S_j)| \leq |\Delta| \leq |N|^2$, the improved algorithm is at least as efficient in theory. In practice, the significant factor of both the improved and original algorithm is $|\Delta|$. Since the expected time in practice of the improved algorithm is $O(|\Delta|)$, while the expected time in practice for the original algorithm is $O(|\Delta|^2)$, we expect the improved algorithm to be more efficient in practice.

6 Experiment

Using the PTRAN system [ABC*87], we performed an experiment to determine the effectiveness of our techniques with respect to statement parallelism. Normally, PTRAN includes a partitioning [Sar89] pass that eliminates inefficient parallelism, based on expected execution times and overheads. Because such parameters are specific to a given architecture, we performed our experiments without partitioning. For input, we chose three *complete* programs from numerical analysis:

EISPACK a benchmark that computes all eigenvalues and eigenvectors of a nonsymmetric matrix [SBD*76]

LINPACK a Gaussian elimination benchmark that factors and solves a general system of simultaneous equations [DBMS79]

SIMPLE a benchmark program for computational fluid dynamics and heat flow [Gil80]

Our experiment accounted for only the static properties of these programs. More experimentation is required to determine how run-time performance is improved. Our static measurements do not account for frequency of execution or for granularity of tasks. However, control dependence will determine concurrency among large-grained tasks, such as entire do loops and procedure calls.

First, we present some general statistics about the programs we examined. The table in Figure 14 contains:

NODES the total number of control dependence graph nodes, summed over all procedures of the program.

CONCUR the number of nodes that could execute concurrently with some other node, ignoring data dependences.

FOUND the number of nodes for which some concurrency was found by our best technique.

DIFF the number of nodes for which a difference was observed between the concurrency discovered by our best and worst techniques. These differences are elaborated by the table in Figure 16 which compares weakened versions of our best algorithm.

Program	NODES	CONCUR	FOUND	DIFF
EISPACK	784	524	453	127
LINPACK	405	280	262	55
SIMPLE	901	618	600	180

Figure 14: Program Statistics

Our experiment essentially examined each node of a program and measured the concurrency of that node as various techniques were applied to eliminate and satisfy dependences. In this experiment, the following statistics

are computed for each node n of the control dependence graph for which any concurrency could be achieved: [2]

MAX the number of siblings of n that could execute concurrently with n. Ignoring data dependences, this statistic measures maximal concurrency. A given combination of concurrent nodes is counted only once, so if X could execute concurrently with Y, then we do not count that Y executes concurrently with X. This statistic determines an upper bound on "strict" statement concurrency, where statements do not execute in the parallel program unless they would execute in the sequential program.

BEST the number of siblings of n that could execute concurrently with n after our best treatment of data dependences: elimination of dependences by privatization and satisfying dependences by DAG constraints. We express this statistic as a percent of $MAX(n)$.

The average and median for these statistics are shown in Figure 15.

Program	MAX		BEST	
	Avg.	Median.	Avg.	Median.
EISPACK	3.1	2.0	66%	71%
LINPACK	3.7	2.0	74%	100%
SIMPLE	5.5	3.0	82%	100%

Figure 15: Potential Concurrency and Percentage Detected

We also measured the effectiveness of less-powerful techniques, relative to the *BEST* measurement described above. We were especially interested in how much concurrency was due to allowing DAG constraints. Recall that the alternative is the total ordering (i.e., sequential execution) of identically control dependent nodes, whose concurrent execution potentially results in the execution of data dependent nodes [BCFH89]. Each of the following statistics is expressed as a percentage-difference from $BEST(n)$:

WORST No dependences are eliminated by privatization, and all dependences are satisfied without DAG constraints.

PRIV Dependences are eliminated by privatization, but dependences are satisfied without DAG constraints.

DAG DAG constraints are allowed, but no dependences are eliminated by privatization.

The median for each of these statistics is shown in Figure 16.

Although *BEST* can never be outdone by *WORST*, there is no provable ordering between *DAG* and *PRIV*. However, the experiments indicate that concurrency is not significantly improved by privatization, with or without DAG parallelism. Thus, allowing DAG

---

[2]Such nodes are counted in *CONCUR* of Figure 14.

Program	DAG	PRIV	WORST
EISPACK	0%	−71%	−77%
LINPACK	0%	−50%	−50%
SIMPLE	0%	−67%	−67%

Figure 16: Comparison of Weaker Techniques (Median)

parallelism and performing no privatization results in almost all parallelism discovered in our experiments.

Our experiment has yielded some interesting results. First, if data dependences are ignored, then almost 70% of the control dependence graph nodes could execute concurrently with (an expected) three other nodes. Examining *MAX* in Figure 15, the disparity between the average and the mean for *SIMPLE* implies greater concurrency among some nodes. In fact, one node had 36 concurrent siblings! If data dependences are considered, then our best approach reduces statement concurrency by roughly 25% to satisfy dependences that could not be eliminated by privatization. Removing privatization does not significantly impair parallelism.

There are three explanations for the apparent uselessness of privatization:

- Although arrays are not specifically excluded from our algorithm, conditions that allow privatization are difficult to apply to arrays. For example, the determination of liveness is difficult because each definition of an array *preserves* rather than *kills* the array. Although the expense of array allocation and copying may render array privatization prohibitively expensive, we must investigate the benefits of privatizing arrays.

- We examined the benefits of privatization only for concurrency among statements. Other work has verified the usefulness of privatization (scalar expansion) in the context of loop vectorization [CKV85].

- We do not privatize variables where the associated statements are already sequenced, or where compile-time determination of `copy out` fails.

We do not know which, if any, of these explanations can exonerate privatization; more experimentation is required.

7 Conclusion

We have presented algorithms that use control and data dependence to restructure programs for a variant of `cobegin...coend`-style parallelism. The algorithms are efficient and have been implemented in the PTRAN system. Our experiments have thus far indicated the success of DAG parallelism over more restrictive forms of constraints, but privatization did not play a major role in determining statement concurrency. Perhaps a more ambitious privatization algorithm would find greater concurrency. Of greater urgency is the trial of our techniques as we execute the automatically parallelized programs on a real multiprocessor.

Acknowledgements

References

[ABC*87] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Proceedings of the 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, Vol. 5, No. 5, pp. 617-640.

[AK87] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–592, October 1987.

[BC86] Michael Burke and Ron Cytron. Interprocedural dependence analysis and parallelization. *Proceedings of the Sigplan '86 Symposium on Compiler Construction*, 21(7):162–175, July 1986.

[BCF*88] Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, Vivek Sarkar, and David Shields. Automatic discovery of parallelism: a tool and an experiment. *ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, 77–84., July 1988.

[BCF*89] Michael Burke, Ron Cytron, Jeanne Ferrante, Michael Hind, Wilson Hsieh, and Vivek Sarkar. *Automatic Parallelization for DAG Parallelism*. Technical Report, MIT, NYU, IBM, 1989. Text in preparation.

[BCFH89] Michael Burke, Ron Cytron, Jeanne Ferrante, and Wilson Hsieh. *Automatic Generation of Nested, Fork-Join Parallelism*. Technical Report, MIT,IBM, February 1989. To appear in The Journal of Supercomputing.

[CF87a] Ron Cytron and Jeanne Ferrante. *An Improved Control Dependence Algorithm*. Technical Report, IBM, 1987. Tech. Report RC 13291.

[CF87b] Ron Cytron and Jeanne Ferrante. What's in a name? or the value of renaming for parallelism detection and storage allocation. *Proceedings of the 1987 International Conference on Parallel Processing*, 19–27, August 1987.

[CFR*89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method for computing static single assignment form. *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 25–35, January 1989.

[CKV85] Ron Cytron, David J. Kuck, and Alex V. Veidenbaum. The effect of restructuring compilers on program performance for high-speed computers. *Computer Physics Communications*, 37:39–48, 1985.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *Linpack Users' Guide*. SIAM Press, 1979.

[FERN84] J. A. Fisher, J. R. Ellis, J. C. Ruttenberg, and A. Nicolau. Parallel processing: a smart compiler and a dumb machine. *Proceedings of the ACM Symposium on Compiler Construction*, 37 − 47, June 1984.

[FOW87] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 319–349, July 1987.

[Gil80] E. J. Gilbert. *An Investigation of the Partitioning of Algorithms Across an MIMD Computing System*. Technical Report, Computer Systems Laboratory, Stanford University, May 1980. TR No. 176.

[Hsi88] Wilson C. Hsieh. *Extracting Parallelism from Sequential Programs*. Technical Report, Massachusetts Institute of Technology, May 1988. Master's thesis.

[IBM88] IBM. *Parallel Fortran Language and Library Reference*. Technical Report, International Business Machines, March 1988. Pub. No. SC23-0431-0.

[Kuc78] David J. Kuck. *The Structure of Computers and Computations*. John Wiley and Sons, 1978.

[LT79] T. Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, July 1979.

[Mid86] Samuel Pratt Midkiff. *Automatic Generation of Synchronization Instructions For Parallel Processors*. Technical Report, Center for Supercomputing Research and Development, U. of Illinois, May 1986. CSRD Rept. No.588, UILU-ENG-86-8002.

[Nic86] Alexandru Nicolau. *A Fine-Grain Parallelizing Compiler*. Technical Report TR 86-792, Department of Computer Science-Cornell University, December 1986.

[PM87] Dick Pountain and David May. *A tutorial introduction to OCCAM programming*. March 1987.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.

[SBD*76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and

C. B. Moler. *Matrix Eigensystem Routines – Eispack Guide.* Springer-Verlag, 1976.

[Wol78]   Michael J. Wolfe. *Techniques for Improving the Inherent Parallelism in Programs.* Technical Report, University of Illinois at Urbana-Champaign, 1978. M.S. Thesis.

[Wol82]   Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, 1982. Report No. UIUCDCS-R-82-1105.