Data Structures for Optimizing Programs with Explicit Parallelism

Michael Wolfe, Harini Srinivasan

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 91-005

March, 1991

Data Structures for Optimizing Programs with Explicit Parallelism

Michael Wolfe, Harini Srinivasan
Oregon Graduate Institute of Science and Technology
Department of Computer Science and Engineering
19600 NW von Neumann Drive
Beaverton, OR 97006
mwolfe@cse.ogi.edu, harinis@cse.ogi.edu

March 8, 1991

Abstract

When analyzing programs with parallel imperative constructs (e.g., cobegin/coend), standard computer intermediate representations are inadequate. This paper introduces a new relation called the *precedence* relation and new data structures called the *Parallel Control Flow Graph* and *Parallel Precedence Graphs* for programs with parallel constructs. We show how to report anomalies in parallel programs using *Parallel Precedence Graphs*. In sequential control flow graphs, the precedence relation is represented by the dominance relation. With explicit parallelism, the dominance relation is not the same as the precedence relation; we discuss the significance of the precedence relation and the new data structures for analyzing programs with parallel constructs. These data structures form a concrete basis for the development of efficient algorithms for optimizing parallel programs.

1 Introduction

Parallel programming has become very popular for implementation of numerical and scientific computations. It is desirable that compilers for programs containing parallel constructs be able to perform classical code optimizations. To do this, it is important to establish an efficient intermediate language. Static Single Assignment form (SSA) is a powerful intermediate representation and an efficient platform for the optimization of sequential programs [Cytr 89]. Extending SSA to parallel programs is a non-trivial task and requires the introduction of many new concepts.

We introduce Parallel Control Flow Graphs and Parallel Precedence Graphs and a new relation called the *precedence relation*. The dominance relation is the same as the precedence relation in sequential control flow graphs; the two relations are not identical in parallel control flow graphs. We illustrate this using examples. We also define *parallel dominance frontiers* using the precedence relation. An application of Parallel Precedence graphs is in reporting anomalies in parallel programs. Associated concepts are the *wait-dominance* relation and the wait-dominator tree, both of which are also used in reporting anomalies in parallel programs.

Section 2 discusses Control Flow Graphs and dominance relations and defines dominance frontiers in the case of sequential programs. Section 3 describes the parallel construct considered in this paper. Section 4 illustrates the need to define the precedence relation and to redefine dominance frontiers.

Section 5 defines the various new data structures introduced in this paper. Section 6 describes an application of Parallel Precedence graphs.

2 Control Flow Graphs for Sequential programs

A control flow graph (CFG) is a directed graph that models program control flow. Nodes of a CFG are the basic blocks of the program with two additional nodes, Entry and Exit. Edges in the CFG correspond to the transfers of control between basic blocks of the program. There is also an edge from Entry to any basic block where control enters the program and an edge to Exit from any basic block where control leaves the program.

Formally, we denote a Control flow graph as G = $\langle$V,E,V_{entry}, $V_{exit}\rangle$ where V denotes the nodes in the CFG, E denotes the edges and V_{entry} is the start node (in our case, Entry) and V_{exit} is the exit node (in our case Exit). Node Z is a successor of node Y if there is an edge Y $\rightarrow$ Z in the CFG. Node X is a predecessor of node Y if there is an edge X $\rightarrow$ Y in the CFG.

2.1 Dominance Relation

Let G = <V, E, Entry, Exit> be a CFG with start node Entry and exit node Exit. A node X *dominates* another node Y (X dom Y) in G if every path from Entry to Y contains X. Note that Entry dominates all other nodes, and every node Y dominates itself. A node X *strictly dominates* node Y if X dom Y and X is not equal to Y.

Node W is the *immediate dominator* of Y if W dominates Y and every other strict dominator of Y also dominates W. Therefore, the immediate dominator of Y is the closest strict dominator of Y (denoted idom(Y)). In a sequential program, the dominance relation can be represented by a tree rooted at Entry with an edge to every node from its immediate dominator.

The *dominance frontier* of a node X, DF(X), is the set of all CFG nodes Z such that X dominates a predecessor of Z but does not strictly dominate Z. The rest of the paper refers to dominance frontiers as *sequential dominance frontiers.*

Lengauer and Tarjan [Len 79] describe an O(mα(m,n)) algorithm for finding dominators in a flow-graph. Dominator trees are very useful data structures in code optimizations and also in developing efficient intermediate forms. Ferrante et al [FOW 87] use dominators in the reverse control flow graph to find control dependences and Cytron et al [Cytr 89] use dominator trees and dominance frontiers to develop the Static Single Assignment intermediate form.

3 The Parallel Sections Construct

The Parallel Sections Construct [PCF] is similar to a cobegin-coend [Brin 73] or the parallel cases statement introduced by Allen et al[ABC 88].

The Parallel Sections Construct is block structured and is used to specify parallel execution of identified sections of code. The parallel sections may also be nested. The sections of code must be data independent, except where an appropriate synchronization mechanism is used. If a variable is assigned in two different sections, the compiler may need to detect an anomaly. In the case of array variables, data dependence analysis is essential to report anomalies. Synchronization between parallel sections is realized using Wait clauses. An example of a Parallel Sections Construct using Wait clauses is shown in figure 1. Each {stmt list} contains one or more statements and execution within a section is sequential.

Consider the parallel program in figure 1. Assuming that there are no other assignments to variables v and u in the program we have the following situation:

```
v = 90
Parallel sections
Section A
        v = 100
        u = 1
        {stmt list}
Section B
        u = 2
        {stmt list}
Section C, Wait(A)
        {stmt list}
Section D, Wait(B,C)
        {stmt list}
Section E, Wait(D)
        {stmt list}
Section F, Wait(A)
        v = 120
        t = u
        {stmt list}
end parallel sections
```

Figure 1: Parallel Sections Construct

The value of u used at section F is indeterminate. Since sections A and B execute in parallel, the variable u may be modified before section F starts execution. On the other hand, section F could execute before section B and the value of u used in F will then be the value of u assigned in section A i.e., 1.

To circumvent such non-determinism in the program, we follow *copy-in/copy-out* semantics in the following manner. When control flow enters a parallel block, every parallel section forked will get a local copy of the values of variables in memory. If there are no wait clauses, then an update is done to the global copy at the coend node. Of course, if there are two assignments to the same variable within the parallel block, an anomaly has to be reported. In the presence of wait clauses, we need to propagate the modified local values to the *waiting* sections. Again, if a variable is assigned in more than one section for which another section is waiting, then it is an anomaly and should be reported. The value of the variable is, however, still propagated to the waiting section. At the coend node, the global values are updated, as before.

4 Example

Consider the programs in figures 2.a and 2.b. The corresponding control flow graphs are shown in figures 3.a and 3.b respectively. In figure 3.a, nodes c and d do not dominate node e. This is because nodes c and d do not always appear on the path from Entry to node e.

In the case of figure 3.b, nodes r and s do not dominate node t. However, we know that both r and s must precede node t in execution. Therefore, even though the dominance relation does not exist between nodes r and t and nodes s and t, there does exist a precedence relation.

Consider the parallel program in figure 4. The corresponding control flow graph (using the definition in section 2) is given in figure 5. By definition of the dominance relation, nodes c, m and q do not dominate node t. This is because nodes c, m and q do not appear on all paths from Entry to node t.

Figure 2.a

```
X = 1
Y = 1
if P then
   X = 2
else
   Y = 2
endif
Z = X+Y
```

Figure 2.b

```
X = 1
Y = 1
Parallel sections
Section
   X = 2
Section
   Y = 2
end parallel sections
```

Figure 2: Example programs

However, the semantics of the parallel construct considered here requires that nodes c, m and q *must all execute* before node t. In other words, the execution of c, m and q *must precede* node t.

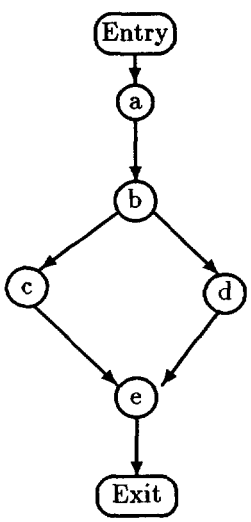


Figure 3.a

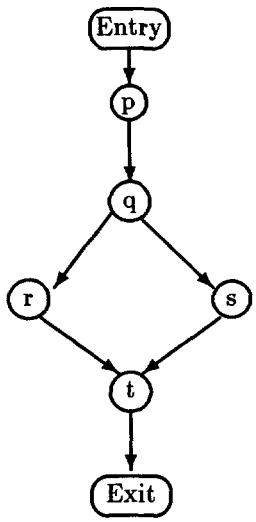


Figure 3.b

Figure 3: Control Flow Graphs for the programs in figures 2.a and 2.b

Similarly, node l, p and s do not dominate node t, but *must precede* node t in execution. It is clear from the definitions in section 2.1 that a dominator tree cannot convey the precedence relation explained above.

Since we are interested in doing static single assignment for parallel programs by extending the algorithms in [Cytr 89], we find a need to compute dominance frontiers in programs having parallel constructs.

```
begin
(a)   F ← 1
(a)   E ← 7
(a)   G ← 0
(a)   L ← 1
(a)   K ← 5
(a)   B ← 7
(b)   Parallel sections
      Section
(c)      if P then
(d)         E ← F
(d)         G ← G+1
(d)         D(i) ← F−1
         else
(e)         E ← F+G
(f)         Parallel sections
(g)         Section
(g)            H ← E
            Section
(h)            if Q then
(i)               G ← G+3
(j)            endif
(k)         end parallel section
(l)      endif
      Section
(m)      if R then
(n)         L ← L× K
(p)      endif
      Section
(q)      if M then
(r)         E ← D(j)
(r)         D(j) ← B × 9
(s)      endif
(t)   end parallel section
(u)   print B
end
```

Figure 4: Example Parallel Program

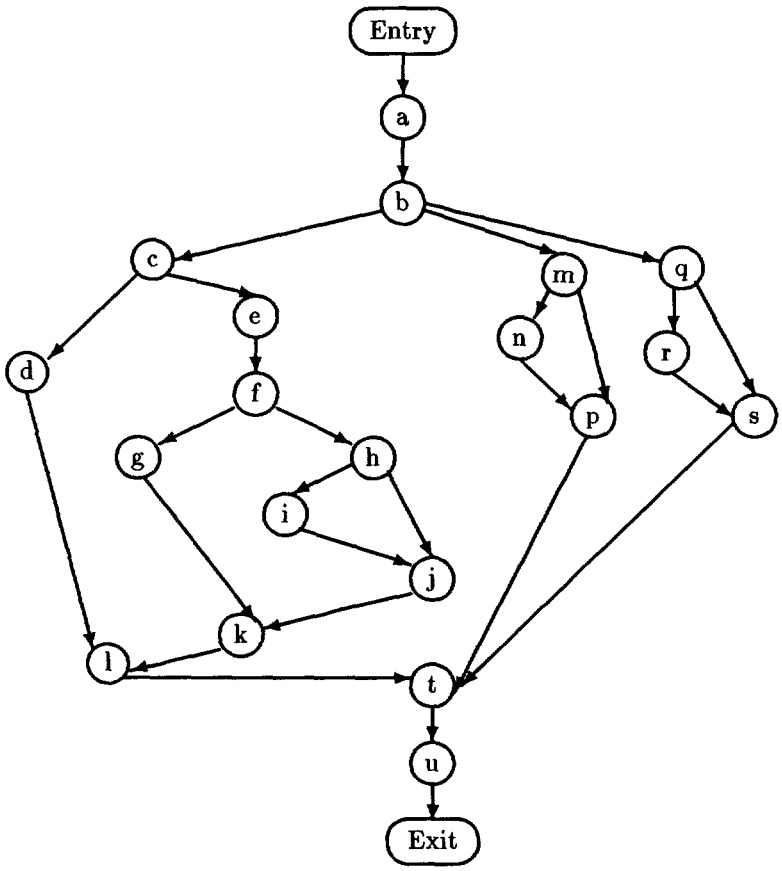


Figure 5: CFG for the parallel program in figure 4

5 Parallel Control Flow Graphs and Parallel Precedence Graphs

A *Parallel Control Flow Graph* is a CFG which may have a new type of node called *supernode* or *parallel block*. A parallel block essentially represents the parallel construct described in the Section 3.

A Wait clause in a parallel block imposes a dependence (called *wait-dependence*) between the waiting section and the sections specified in the wait clause. We introduce a new data structure called the *Parallel Precedence Graph* that represents a parallel block. Nodes in the PPG are the various sections in the parallel block with two additional nodes, cobegin and coend. The edges in the PPG (also called *wait-dependence arcs*) are those representing the wait dependences. For example, the PPG for the program in figure 1 in shown in figure 6. For technical reasons, there is an edge from the cobegin node to the corresponding coend node in the PPG. In the absence of any wait clause in the parallel block, we have a degenerate case where there are only two sets of wait-dependence arcs, one from the cobegin node to the sections in the parallel block and other from the different sections to the coend node.

Formally a PCFG is defined as the graph G = (V, E, S_{entry}, S_{exit}) where V, E, S_{entry} and S_{exit} are defined as follows :

V = { a | a is either a basic block or a parallel block}

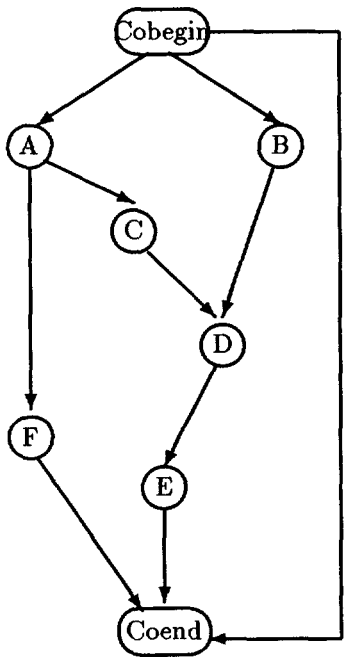


Figure 6: PPG for the program in figure 1

E = { a → b | a,b ∈ V }

Edges represent flow of control in the parallel program.

S_{entry} is the start node of the PCFG.

S_{exit} is the exit node of the PCFG.

A parallel block is represented by a PPG, which is formally defined as a graph $G_p = \langle V_p, E_p, P_{entry}, P_{exit} \rangle$ where

V_p = {N | N is either the cobegin node, coend node or a section in the parallel block }

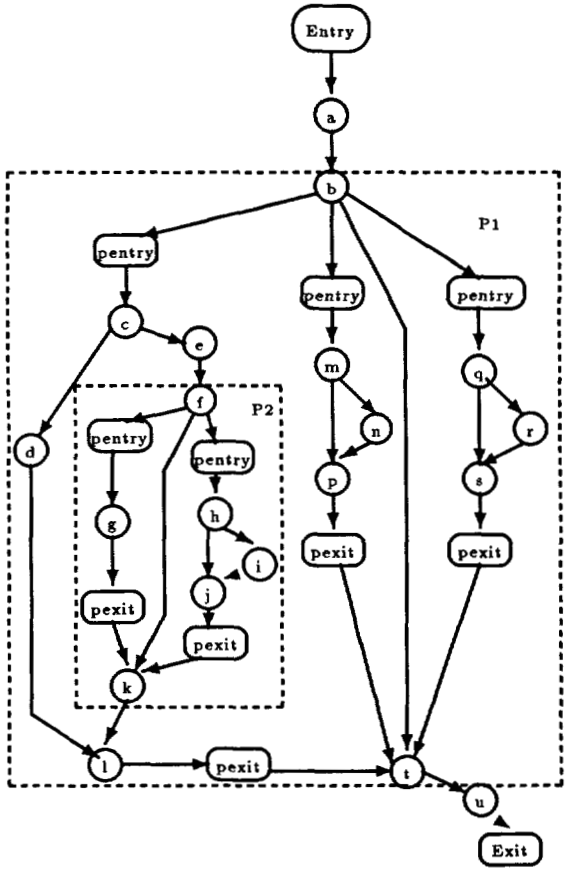
Each section in a PPG is again a PCFG with two additional nodes, p_entry (marking the entry into that section) and p_exit (marking the exit from that section).

E_p is the set of edges or wait-dependence arcs in the PPG.

P_{entry} is the cobegin node

P_{exit} is the coend node.

If the PCFG corresponds to a section in the parallel block, S_{entry} and S_{exit} are the p_entry and p_exit nodes respectively; otherwise, they are the Entry and Exit nodes. The PCFG of the parallel program in figure 4 is shown in figure 7. For technical reasons, there is an edge from the cobegin node to the corresponding coend node in the PPG.



Node	SDF	PDF
Entry	–	–
a	–	–
b	–	–
c	t	–
d	l	l
e	l	l
f	l	l
g	k	l
h	j	j
i	j	j
j	k	l
k	l	l
l	t	–
m	t	–
n	p	p
p	t	–
q	t	–
r	s	s
s	t	–
t	–	–
u	–	–
Exit	–	–

Sequential and Parallel Dominance Frontiers

Figure 7: PCFG for the program in figure 4

5.1 Precedence Relations and Parallel Dominance Frontiers

A node X **precedes** node Y in the PCFG if the execution of X precedes the execution of Y. The following observations can be made about the precedence relation:

- If a parallel block X precedes a node Y in the PCFG, not all the basic blocks within X must precede Y. For example, in figure 7, P2 precedes node *l* but node *i* within P2 does not precede *l*.
- Consider the sections of code within a parallel block to be represented by individual nodes. Let X and Y be nodes that represent two sections in the parallel construct. X precedes Y if the p_exit node in X must execute before the p_entry node in Y. This is true when there are wait clauses in the parallel program. In the parallel program in figure 1, section A precedes section C because the p_exit for A in the PCFG of this program will execute before the p_entry node in C.
- In the case of sequential CFG's the dominance and precedence relations are the same. This is evident from the example in figure 2.a.

Definition

Let the set $\mathcal{S}(X)$ denote the sequential dominance frontier of a node X in the PCFG. We proceed to determine the parallel dominance frontier of X as follows.

If $\exists$ Y such that Y is either a coend node or a p_entry node and $Y \in \mathcal{S}$ then the parallel dominance frontier of X is defined as

$\mathcal{P}(X) = \mathcal{S}(X) - Y \cup \mathcal{P}(P_y)$ where P_y is the closest parallel block enclosing Y if there exists such a parallel block.

else $\mathcal{P}(X) = \mathcal{S}(X)$.

For example, the sequential dominance frontier of node *j* is node *k* which is a coend node. But the parallel dominance frontier of node *j* is the parallel dominance frontier of P2 which is node *l*. Similarly, the parallel dominance frontier of nodes *l*, *p* and *s* in figure 7 is not *t* but the parallel dominance frontier of the enclosing parallel block, P1 ie., the empty set. This is because P1 is the outermost parallel block.

It should be noted that any node that is the merge point of two or more sections in a parallel block in the PCFG cannot be in the parallel dominance frontier of any other node. Such nodes are referred to as *parallel merge nodes*. This is the reason why coend and p_entry (which can be a merge point in the presence of wait clauses) are not considered as parallel dominance frontiers.

The parallel dominance frontiers are used to do static single assignment in parallel programs. We do not, however, discuss SSA here as it is the topic of a subsequent paper.

5.2 Wait-dominance

We define a new relation between the different nodes in a Parallel Precedence graph called *wait-dominance*.

1. **Wait-dominance** – A node X in the PPG wait-dominates a node Y (X wdom Y) if X appears on all paths from the fork node to Y.

   The wait-dominance relation is reflexive and transitive. The *immediate wait-dominator* of Y in the PPG is the closest wait- dominator X of Y such that $Y \neq X$.

   It is clear that wait-dominance is the dominance relation in the PPG; in the former we are considering sections of the parallel block. When there are no wait clauses, the wait-dominance tree is of depth one. Figure 8 gives the wait-dominator tree for the PPG in figure 6.

2. **Wait-dominance Frontier** – The wait-dominance frontier of a node X in the PPG is defined as follows

   WDF(X) = { Y | $\exists$ P $\ni$ (P $\in$ Pred(Y) $\wedge$ X wdom P $\wedge$ X does not strictly wdom Y) }

   In our example in figure 8, the wait-dominance frontiers of A,C and B is {D}; the WDF's of D,E and F is {coend}.

6 Reporting Anomalies in Parallel Programs

An application of parallel precedence graphs is in reporting anomalies in parallel programs. When a variable is assigned in more than one branch of a parallel block, an anomaly should be reported because it is not clear which value assigned to the variable will reach the coend node. How do we detect such anomalies? This is discussed in this section with relevant examples.

ψ–functions : A ψ–function signifies the merge of potentially anomalous birth points [WZ 84]. Whenever a variable is assigned in more than one branch of a parallel block, a ψ–function is placed at the merge point. In the presence of wait clauses, a ψ–function is placed for a variable at a node if

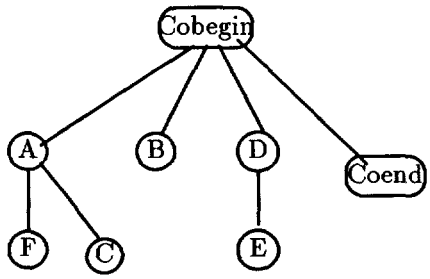


Figure 8: Wait Dominator tree for the PPG in figure 6

the variable is assigned in more than one predecessor of this node in the PPG. A ψ–function is of the form $v_n = \psi(v_1, v_2, \ldots, v_{n-1})$ where each v_i is a variable and the number of operands (ie. n-1) is the number of predecessors Y of the point where the ψ–function appears in the PPG such that Y contains an assignment to the variable v. The number of operands is always greater than one.

For example, in our example program in figure 4, a ψ–function is required for variable `E` because it is assigned in more than one section of the outer parallel sections construct. A ψ–function may be required for the array variable `D`. Since `D` is a subscripted variable, determining if a ψ–function is needed calls for data dependence analysis [Burk 86, Wolf 87]. Also, in figure 1, a ψ–function is required for variable v at the merge point, D.

6.1 Placement of ψ functions

Algorithm 1 (figure 9) places ψ functions using the concept of wait-dominance and wait-dominator trees defined in the previous section.

We use the following data structures in the algorithm to place ψ–functions:

$\mathcal{A}$(V) : nodes where variable V is assigned.

$\mathcal{W}$(N) : wait-dominance frontiers of node N.

$\mathcal{P}$(V) : Accumulates the pair-wise wait-dominance frontiers of all nodes where variable V is assigned.

ψ_comp(N) : A boolean array which tells if a ψ–function exists at node N.

Wait-dominance frontiers are computed using the same algorithm for computing dominance frontiers presented in [Cytr 89]. Note that the algorithm for finding wait-dominance frontiers will work on the PPG and wait-dominator tree instead of CFG and dominator tree as in [Cytr 89].

Proof of Correctness of Algorithm 1

Algorithm 1 first computes the iterated wait-dominance frontier, IWDF of all the nodes where a variable is assigned. The union of the iterated wait-dominance frontiers and $\mathcal{A}$(V) for the variable is stored in $\mathcal{I}$(V). The next phase is the placement of ψ–functions which is done at the *pair-wise intersection* of the WDF's of all the nodes in $\mathcal{I}$(V). A ψ–function for a variable V is considered as an assignment to V. This is achieved by adding the WDF of the nodes where V is assigned to $\mathcal{I}$(V). A ψ–function is placed at a parallel merge point where two or more assignments to the same variable reach. A variable may be assigned in two nodes X and Y whose wait-dominance frontiers do not intersect. However the paths from X to coend and Y to coend *must* intersect at some point in the PPG, at least at the coend node. By computing the iterated wait-dominance frontiers, it is possible to find the points

```
for each variable V do
    hasalready(*) = 0
    ψ_complete = False
    I(V) = A(V)
    while (ψ_complete = False) do
          for each N_i ∈ I(V) ∋ hasalready(N_i) = 0 do
              hasalready(N_i) = 1
              if W(N_i) ⊈ I(V) then
                 add W(N_i) to A(V)
                 ψ_complete = False
              else
                 ψ_complete = True
              endif
          end
    end

    for each N_i, N_j ∈ I(V) and N_i ≠ N_j do
        P(V) = ⋃(W(N_i) ∩ W(N_j))
    end
    Place ψ−function at all nodes in P(V)
end
```

Figure 9: Algorithm 1

of intersection of the paths from X to coend and Y to coend. Since a variable may be assigned in more than two nodes of the PPG it is required that we consider the pair-wise intersection of the iterated WDF's of all the nodes where the variable is assigned.

For the algorithm to halt it is necessary that the while loop terminate. The while loop *will* eventually terminate because the wait-dominance frontier is a finite set of nodes.

The time required to process a single variable using algorithm 1 is proportional to the time to compute the IWDF's plus the time to compute the pair-wise intersection of the WDF's. The former takes $O(N)$ time where N is the number of nodes in the PPG and the latter takes $O(N^2)$ time. Therefore, the overall algorithm takes $O(N + N^2)$ time.

Referring to figures 1 and 6, $\mathcal{A}$(u) is {A,B}, $\mathcal{I}$(u) is {A,B,D,coend} and $\mathcal{P}$(u) is {D,coend}.

7 Conclusion

The dominance relation is a very important concept used in code optimization algorithms and in developing efficient intermediate forms. We observe that the dominance relation does not convey the control flow properties of parallel programs as it does for sequential programs. The precedence relation is an important concept for analyzing parallel programs. We have not discussed SSA here. SSA is discussed in detail in [Cytr 89] and we have extended dominance frontiers used in the algorithms to compute SSA to take parallel constructs into account. Anomalies in parallel programs are very common and it is desirable to report anomalies at compile time. We discuss an efficient method of reporting anomalies in parallel programs.

References

[Cytr 89] Cytron, R., J. Ferrante, B.K. Rosen, M.N. Wegman, F.K. Zadeck . An Efficient Method of Computing Static Single Assignment Form. *Conf. Record of the 16th Annual ACM Symp. on Principles of Programming Languages*, 25-35, 1989.

[Brin 73] Per Brinch Hansen. Operating Systems Principles. Prentice-Hall Series in Automatic Computation, 57-59, 1973.

[ABC 88] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Journal of Parallel and Distributed Computing*, Vol. 5, No. 5, 617-640, October 1988.

[Len 79] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, 121-141, July 1979.

[Burk 86] Michael Burke, Ron Cytron. Interprocedural Dependence Analysis and Parallelization. *Proc. of the SIGPLAN 86 Symp. on Compiler Construction*, 162-175, Palo Alto, CA, June 25-27, 1986.

[Wolf 87] Michael Wolfe, Utpal Banerjee. Data Dependence and Its Application to Parallel Processing. *Intl Journal of Parallel Programming*, 137-178, April 1987.

[FOW 87] Ferrante, J., K.J Ottenstein, and J.D Warren. The program dependence graph and its use in optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319-349, July 1987.

[PCF 90] PCF–FORTRAN. *Parallel Computing Forum*, Version 3.0, April 1990.

[WZ 85] Mark N. Wegman, Frank Kenneth Zadeck. Constant Propagation with Conditional Branches. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, 291-299, January 1985.