University of Colorado, Boulder
CU Scholar

Computer Science Technical Reports | Computer Science

Winter 12-1-1991

An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs ; CU-CS-564-91

Harini Srinivasan
University of Colorado Boulder

Dirk C. Grunwald
University of Colorado Boulder

Follow this and additional works at: http://scholar.colorado.edu/csci_techreports

Recommended Citation

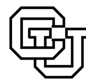
Srinivasan, Harini and Grunwald, Dirk C., "An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs ; CU-CS-564-91" (1991). *Computer Science Technical Reports*. 543.
http://scholar.colorado.edu/csci_techreports/543

This Technical Report is brought to you for free and open access by Computer Science at CU Scholar. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of CU Scholar. For more information, please contact cuscholaradmin@colorado.edu.

An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs

Harini Srinivasan
Dirk Grunwald

CU-CS-564-91 December 1991



University of Colorado at Boulder

Technical Report CU-CS-564-91
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309

An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs

Harini Srinivasan
Dirk Grunwald*
Department of Computer Science
University of Colorado
Boulder, Colorado

December 1991

Abstract

This paper describes an efficient method of computing Static Single Assignment form explicitly parallel programs using `parallel sections` and `wait` clauses. Static Single Assignment form [Cytron et al. 89] is an efficient intermediate representation and has been used as a platform for various classical code optimization algorithms [Rosen et al. 88], [Alpern et al. 88], [Wegman & Kenneth Zadeck 85]. We believe the SSA form will also be useful in performing code optimizations on parallel programs and our techniques will allow us to extend existing algorithms to optimize this class of explicitly parallel programs.

1 Static Single Assignment Form

Static Single Assignment (SSA) form [Cytron et al. 89] is an efficient intermediate representation and has been used as a platform for various classical code optimization algorithms [Rosen et al. 88], [Alpern et al. 88], [Wegman & Kenneth Zadeck 85]. Algorithms that convert a sequential program into SSA form traverse the program *Control Flow Graph* (CFG). The Control Flow Graph of a sequential program is a 4-tuple $\langle N, E, Entry, Exit\rangle$. The set of nodes, N, represents the basic blocks in the program. The set of edges, E, represents sequential

*This research supported in part by the National Science Foundation under NSF Grant CCR-9010624.

$$
\begin{array}{rcl}
v & = & 4 \\
 & = & v + 5 \\
v & = & 6 \\
 & = & v + 1
\end{array}
\qquad\qquad
\begin{array}{rcl}
v_1 & = & 4 \\
 & = & v_1 + 5 \\
v_2 & = & 6 \\
 & = & v_2 + 1
\end{array}
$$

(a) Original Code Fragment (b) SSA Form

Figure 1: Conversion of Straight-Line Code to SSA Form

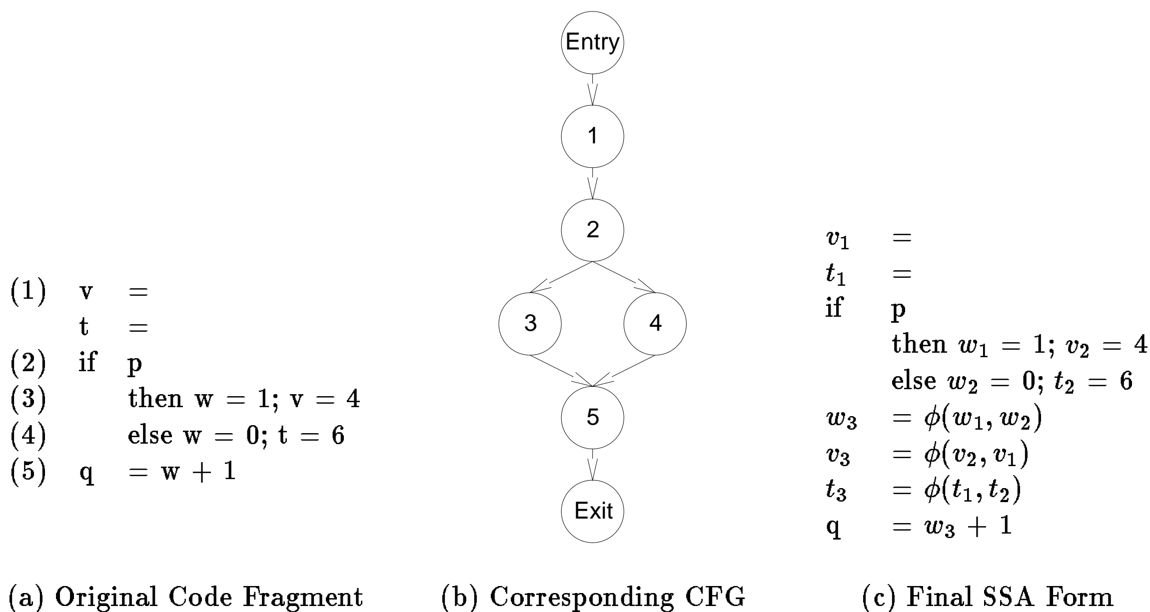
flow of control between basic blocks. Two distinguished nodes, *Entry* and *Exit*, represent the unique program entry and exit points of the program. If there is more than one entry point to the program, edges are added from the *Entry* node to the different entry points resulting in a single entry point. Similarly, if there is more than one exit node, edges are added from these nodes to the single *Exit* node.

After converting a program into its SSA representation, every use of a variable has exactly one *reaching definition.* That is, each variable has been assigned a single value. Transforming a standard sequential program into SSA form introduces additional variables representing distinct values of the original variables.

In the example shown in Figure 1(a) (taken from [Cytron et al. 89]), the variable v is assigned two values. When using SSA form, the code fragment is transformed to the version shown in Figure 1(b). The original variable v is renamed to v_1, and the second definition, and subsequent uses, are named v_2. The original v is a variable name that may have some meaning for the programmer. The names v_1 and v_2 are value names; all uses of a particular value defined by an assignment use the same value name.

Because SSA only represents static dataflow, it can not determine the true value of variables when two edges merge at a node in the Control Flow Graph (CFG). At such merge points, ϕ-functions are introduced to merge distinct values, producing a new value name. The ϕ-functions are artifact of the SSA method; they are used to indicate that the precise value of a variable is no longer known.

In the example in Figure 2(a), each branch of the conditional assigns a subset of the variables v, t and w. The assignments in each branch of the conditional introduce new value names, and



(a) Original Code Fragment (b) Corresponding CFG (c) Final SSA Form

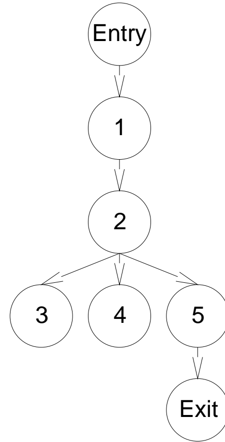
Figure 2: Introduction of ϕ-functions

ϕ-functions are introduced in the merge node, node (5). For example, consider the value of v at the merge node; unless the value of p at node (2) is known at compilation time, it is impossible to know which value reaches the end of the conditional. Thus, the new value name v_3 is introduced by a ϕ-function, indicating that v_3 may have the value of either v_1 or v_2.

Each definition of a value name is stored with a list of each use. Several code optimizations can be efficiently rephrased using this *def/use* information. Many of these algorithms are simplified because the SSA form explicitly indicates the live range of the values. For example, transforming the program fragment in Figure 2(a) to that in 2(c) allows a simple register scheduling method[Cytron & Ferrante 87] to allocate the three variables using two registers without resorting to complex coloring algorithms. SSA form has also been used for constant propagation, dead-code elimination, detecting equality of variables and so on.

1.1 Dominance Relations

A central problem when constructing SSA form is determining the minimal set of nodes where ϕ-functions must be introduced. The efficient algorithm given in [Cytron et al. 89] uses the concepts of *dominance relation* and *dominance frontier*. For two nodes X and Y, X *dominates* Y (expressed $X \ggeq Y$) if X appears on all paths from *Entry* to Y. In Figure 2, node (2) dominates (2), (3), (4) and (5). Nodes (3) and (4) do not dominate any other node shown. Node (1) dominates node (2) and all nodes dominated by (2). A node X *strictly dominates* Y, or $X \gg Y$, if $X \ggeq Y$ and $X \neq Y$. In Figure 2, node (2) dominates itself, but does not strictly dominate itself. The *immediate dominator* of a node X, denoted *idom*(X), is the closest strict dominator of X; in figure 2, the *idom*(2) is node (1) and node (2) is the immediate dominator of nodes (3), (4) and (5). The immediate dominator of a node is unique and can be used to construct a *dominator tree* with an edge from every node to its immediate dominator; the following figure shows the dominator tree for Figure 2.



The *Dominance Frontier* of a node X (denoted DF(X)) in the CFG is the set of all nodes Y, such that X dominates a predecessor of Y but does not strictly dominate Y. For example, in figure 2, the predecessors of node (5) are (3) and (4). Node (3) dominates itself, but does not dominate node (5). This is an example of the definition with X = (3) and Y = (5); thus, node (5) is in the dominance frontier of node (3) (and, by symmetry, node (4)).

To place the minimal set of ϕ-functions when transforming to SSA form, [Cytron et al. 89] show they should be placed at the iterated dominance frontier of the definition nodes. In

[Cytron et al. 89], an algorithm is given to compute minimal SSA form in time $O(E+T+|DF|)$, where E is the number of edges in the CFG, T is the total number of ordinary assignments and ϕ-functions in the program and |DF| is the total size of all dominance frontiers.

2 Parallel Constructs

The SSA form has been used for a number of optimizations for sequential programs. We believe the SSA form can be similarly applied to explicitly parallel programs. The remainder of this paper extends Control Flow Graphs to model some parallel control flow constructs and describes how to derive the SSA form of such explicitly parallel constructs.

This paper considers a subset of the parallel constructs from the PCF Parallel Fortran extensions [Parallel Computing Forum 91]. We consider structured parallelism using **`Parallel Sections`** with **`wait`** clauses.

The **`Parallel Sections`** construct is a block structured construct that specifies parallel execution of identified sections of code, akin to the **`cobegin`** ... **`coend`** notation. Each section of code identified in a **`Parallel Sections`** construct is interpreted as a *parallel thread*. The parallel sections must be *data independent* except where an appropriate synchronization mechanism is used. The **`wait`** clause specifies a partial ordering among the sections of code. All sections whose names are listed in the **`wait`** clause *must* complete before the section with the **`wait`** clause can begin execution.

When sections synchronize, any shared variables are made consistent. For example, if a section modifies a variable used by another section, the updated value must be copied or updated after the sections synchronize. Note that this does not imply a strict copyin-copyout semantics. Rather, it means that values cached in registers or stored in cache must be updated at the synchronization points; however, the values *may* be updated prior to this on, e.g., a system with strongly consistent memory.

Parallel sections with **`wait`** clauses are analogous to sections using **`post`** and **`wait`** statments at the end and beginning of the respective sections of code. Semantically, the concepts

are identical, but `wait` clauses are syntactically simpler and do not result in unstructured parallelism.

Although we only consider `wait` clauses in this paper, we must model the wait clauses as `post` and `wait` events. In our restricted set of programs using only `wait` clauses, sections may only wait for other sections in the same `parallel sections` construct.

If two sections modify the same variable in an unsynchronized manner, the resultant value is undefined. Such programs are considered non-standard conforming PCF Fortran programs and we do not consider them in this paper; they will be addressed in a forthcoming paper.

3 Parallel Flow Graphs

Parallel Flow Graphs are a variant of CFG's extended to handle the `parallel sections` with `wait` clauses, as described in the previous sections. The Parallel Flow Graph is similar to the the Program Execution Graph [Balasundaram & Kennedy 89] and the Synchronized Control Flow Graph [Callahan et al. 90].

Formally, a Parallel Flow Graph (PFG) of an explicitly parallel program is a tuple

$\langle V_p, E_p, \mathit{Entry}, \mathit{Exit} \rangle$, where *Entry* and *Exit* correspond to unique entry and exit points of the program; V_p represents the set of nodes in the Parallel Flow Graph. A node can represent a basic block, a fork node (corresponding to a `parallel sections` statement), a join node (corresponding to an `end parallel sections` statement), *Entry* or *Exit*. The set E_p is the set of edges in the PFG and is the union of the sequential flow edges E_{sf} (edges representing sequential flow of control) and consistency edges E_c.

The consistency edges represent the potential update of a subset of the program values; for example, the head of the edge points to a `wait` statement and the tail originates from a `post`. In this model, the `parallel sections` command is essentially a `post` statement that enables the different sections. The semantics of PCF Fortran actually allow the values to change at any point in a `parallel sections` construct; that is, the semantics are not strict copy-in copy-out, but the compiler may choose to ignore intermediate values in order to keep values in registers or

caches or perform other optimizations. Values from distinct sections of the **parallel sections** *must* be made consistent when the wait clause synchronizes.

For the compiler, the consistency edges serve a purpose similar to one aspect of control flow edges; the consistency edges indicate that values may change, and indicates the block containing the new values. The presence of consistency edges requires the compiler to relax assumptions about the current value of variables; in effect, new value names are introduced using ϕ-functions. The semantics of **wait** clauses, and the more general **post** and **wait** events, actually define precedence relations, but the consistency edges specify a weaker constraint. In particular, if additional precedence edges are added to the graph, the scheduling opportunities for the program are changed. Introducing additional consistency edges may not affect the scheduling of the program; it simply reduces opportunities for optimization. Removing edges may affect the program semantics; in practice, edges from **post** and **wait** statements should not be removed. If we only consider **wait** clauses or **post**/**wait** synchronization, the semantic distinction between consistency edges and precedence edges is slight. In an actual compiler, both precedence and consistency edges would be needed, but for different purposes; in this paper, we need only consider consistency edges.

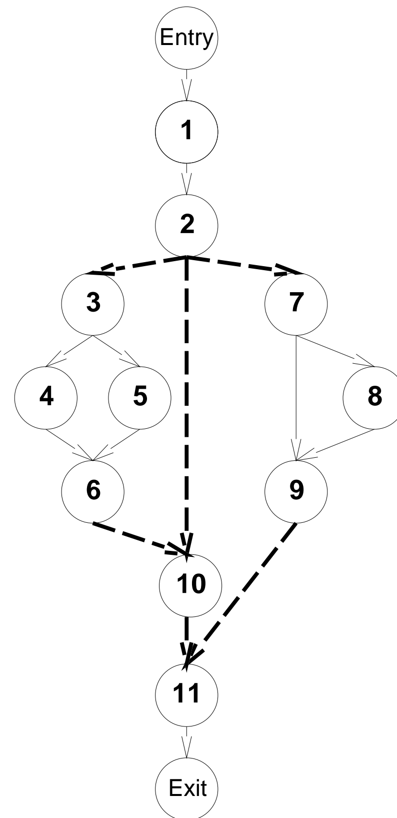
Our goal is to introduce the minimal set of new value names for variables updated across control flow and consistency edges, providing more opportunities for optimization.

As in a Control Flow Graph, certain nodes are distinguished by their function; for example, in a CFG, conditional and loop nodes are distinguished. We distinguish nodes at the beginning and end of parallel section and nodes containing **post** and **wait** statements introduced by converting **wait** clauses. Each section has a sequence of implicit or explicit **wait** statements followed by a number of sequential statements and ending in a sequence of implicit **post** statements. In practice, each **wait** and **post** statement would be a distinct node, allowing the graph to be modified between **post** and **wait** statements; our examples consolidate many of these nodes for expository simplicity.

A *fork node* is either a **parallel sections** statement or an implicit **post** statement in a section that introduces consistency edges to other waiting sections. Likewise, a *parallel merge*

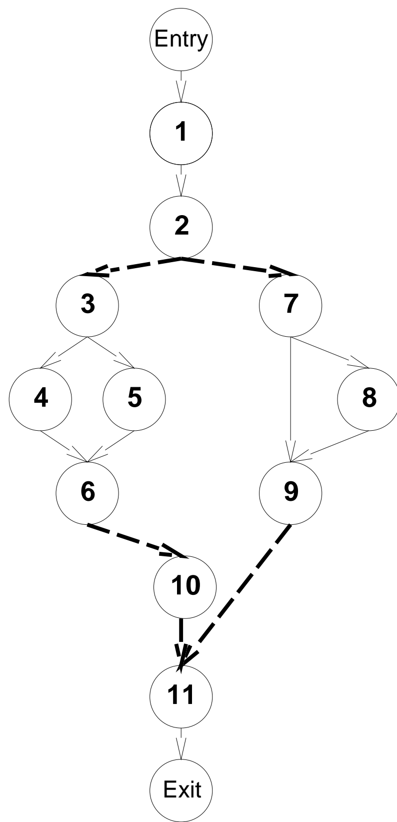
```
(1)   f = 0.9
(2)   Parallel Sections
      Section (A)
(3)      a = 0.5
         if (P) then
(4)         b = a**2
            f = 1;
         else
(5)         b = a/(a+5)
(6)      endif
(7)   Section (B)
         c = 8
         d = 3.9
         if (Q) then
(8)         d = d + c**4
(9)      endif
(10)  Section (C), Wait (A)
         a = a - 0.2
         f = a*b
(11)  End Parallel Sections
```

(a) Parallel Program

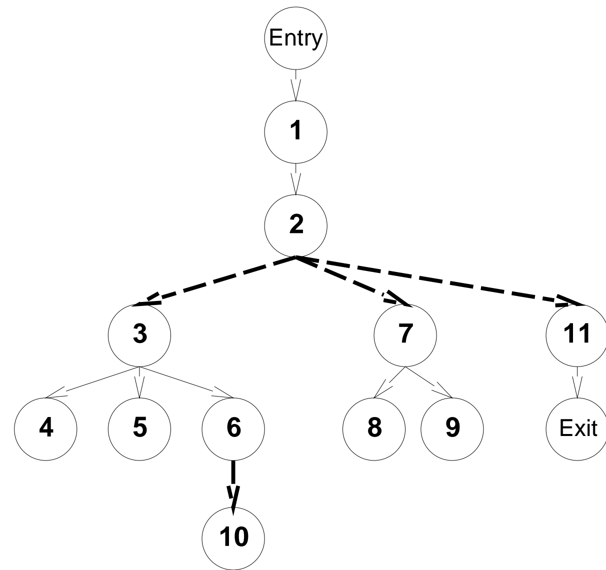


(b) Parallel Flow Graph

Figure 3: Example 1: Parallel Program with Wait clauses



(a) Simplified Parallel Flow Graph



(b) Parallel Dominator Tree

Node	3	4	5	6	7	9	8	10
Dominance Frontier	11	6	6	11	11	11	9	11

(c) Non-null Dominance Frontier entries

Figure 4: Parallel Flow Graph and Dominator Tree for Example 1

node begins with `wait` statements and has at least two incoming consistency edges; examples include an individual section or an `end parallel sections` statement.

Figure 3(a) illustrates a parallel program with the `parallel sections` construct and `wait` clauses. The corresponding Parallel Flow Graph is shown in Figure 3(b). Node (2) is a fork node and (11) is a parallel merge node. The dashed edges represent consistency edges, the other edges are sequential flow edges. Node (10) receives values from the fork node as well as from node (6). In certain cases we can remove redundant consistency edges. In particular if a node X has at least one incoming consistency edge and there is another path from *idom*(X) to X that does not contain a definition of any variable appearing in the other path, the edge can be trivially removed, provided another path from *idom*(X) to X remains. This is a special application of a more general result, to be shown in a later paper. In Figure 3(b), this means the edge between nodes (2) and (10) can be removed, because an edge from (6) to (10) exists, and no variable is modified in the path from (2) to (10). The edge between (6) and (10) can not be removed. If these trivial edge removals are not performed here, they will introduce additional ϕ-functions that will be removed by the algorithm in §4. The simplified Parallel Flow Graph is shown in Figure 4(a).

The Parallel Flow Graph is similar to the Parallel Control Flow Graph [Wolfe & Srinivasan 91] used in a previous formulation of parallel SSA form. However, in [Wolfe & Srinivasan 91], parallel blocks, or supernodes, are distinguished from ordinary basic blocks. In the Parallel Flow Graph, we distinguish between consistency edges and sequential flow edges and do not need data structures to represent parallel blocks.

We can extend the existing notions of precedence, successor, dominator, immediate dominator and dominance frontier to PFG's by ignoring the distinction between consistency and sequential control-flow edges. In the Control Flow Graph, each arc represents a path by which information may possibly be transferred between basic blocks. The same is still true in the Parallel Flow Graph; both sequential and consistency edges indicate the propagation of values. In our final formulation of parallel SSA, a variable update at a merge node will only occur if that variable is modified. Thus, the compiler is free to make assumptions about variables not

updated in those sections; in particular, the values may be cached in registers or other private memory.

Figure 4(b) shows the dominator tree for the PFG in Figure 4(a). Nodes (3) and (7) are dominated by node (2), even though they are connected by consistency edges. Likewise, applying the definitions of §1, nodes (6) and (9) have the same dominance frontier, node (11).

In sequential programs, ϕ-functions are introduced to indicate when the compiler is unable to determine the precise value of a specific variable due to merges in the Control Flow Graph. The ϕ-functions are placed at the nodes defined by the iterated dominance frontier set; in short, they are placed at each node where control flow merges and a value for the variable in question is defined along one of the incoming paths.

For example, in Figure 4, the variable `f` is defined in nodes (1), (4) and (10). Figure 4(c) lists the non-null dominance frontier sets for the simplified PFG. The dominance frontier of node (4) contains node (6). A ϕ-function would be introduced at node (6), because a live value of `f` could reach node (6) from node (4) or node (1). The ϕ-function indicates that values may reach along either branch of the conditional statement.

By comparison, values from *all* branches of consistency edges may reach a merge node. The semantics of PCF Fortran indicate that a program is undefined if more than one value reaches such a node. Since we only consider standard conforming PCF Fortran programs in this paper, it would appear that the ϕ-functions introduced by consistency edges would be identical to those introduced by sequential edges. However, consider the previous case where a value is defined along a single edge; if we place ϕ-functions for consistency edges analogous to those of sequential edges, a ϕ-function would be introduced indicating that a previous value and a new definition reach the same node. For example, the variable `f` is defined in node (10) and node (1); a ϕ-function is introduced in node (11) since it is in the dominance frontier of node (10).

The semantics of the parallel constructs dictate that the definition in node (10) eclipse that of node (1). Although it is not incorrect to introduce a ϕ-function in this case, it inhibits potential optimizations. In particular, if a value is defined along one path in a parallel section, that value should be propagated to successive sections. The compiler can use such information

to better optimize the program. By needlessly placing a ϕ-function at a merge point for consistency edges, inefficient code may be generated.

If we simply apply the algorithms for sequential SSA to programs with `wait` clauses, ignoring the distinction between sequential and consistency edges, a number of these spurious ϕ-functions will be introduced.

4 Translating to SSA form

In his section, we show how to compute parallel SSA form for the parallel constructs discussed. The method involves computing standard SSA form, followed by a 'clean up' phase where spurious ϕ-functions are removed. In a latter paper, we extend the algorithms in this paper to handle `parallel sections` with arbitrary `post` and `wait` events, and non-standard confirming programs where values are modified in unsynchronized sections.

As mentioned in §3, parallel merge points may introduce spurious ϕ-functions. The distinction between a parallel merge node and a sequential merge node can be seen in Example 1. Node (9) is a sequential merge point (corresponding to an `endif` statement) and (11) is a parallel merge point (corresponding to a `end parallel sections` statement) We need a ϕ-function for `d` at (9) since `d` is defined in the if statement depending on the condition Q. However, at (11), we do not need a merge function for the variable `f` since `f` will be assigned at node (10) and only this value should reach the end of the parallel block. Using the SSA algorithms as in [Cytron et al. 89], a ϕ-function for `f` will be introduced at node (11). Therefore, we must eliminate such spurious merge functions.

A *non-determinant* or *anomalous* update occurs if more than one value can potentially reach a parallel merge node. Non-determinate updates may be introduced by non-standard conforming PCF Fortran programs. Detecting and reporting such anomalies at compile time is the topic of another paper; in this paper we treat only standard conforming programs.

Translation of standard-conforming parallel programs using `wait` clauses to SSA form involves the following steps:

Variable	$\mathcal{S}$	DF($\mathcal{S}$)	DF$^+$($\mathcal{S}$)
a	(1), (3), (10)	(11)	(11)
b	(1), (4), (5)	(6)	(6), (11)
c	(1), (7)	(11)	(11)
d	(1), (7), (8)	(9)	(9), (11)
f	(1), (4), (10)	(6), (11)	(6), (11)

Figure 5: Table showing the iterated dominance frontier corresponding to each variable in Example 1

1. Convert the `wait` clauses into into the equivilent `post`/`wait` events.

2. For each variable, we compute the iterated dominance frontier set of the set of nodes, $\mathcal{S}$, that contain definitions for the variable. The iterated dominance frontier set, DF$^+$(S), is as defined in [Cytron et al. 89],

$$\begin{aligned} \mathrm{DF}_1^+ &= \mathrm{DF}(\mathcal{S}) \\ \mathrm{DF}_{i+1}^+ &= \mathrm{DF}(\mathcal{S} \cup \mathrm{DF}_i^+) \end{aligned}$$

   The dominance frontier is from the Parallel Flow Graph rather than the control flow graph. The iterated dominance frontier set for the parallel program in Example 1 is shown in figure 5.

3. In [Cytron et al. 89], the set DF$^+$($\mathcal{S}$) defines the set of nodes where ϕ-functions are placed for the variable being examined. The same is true at this stage for the parallel flow graph. While placing ϕ-functions, we check if the node is a parallel merge point i.e., at least one of the the incoming flow edges are consistency edges. In this case, we use ψ-functions to denote ϕ-functions that occur in parallel merge nodes. All statements that apply to ϕ-functions apply similarly to ψ-functions; we simply introduce this notation to refer to ϕ-functions occurring in parallel merge nodes. A set of nodes with ψ-functions is maintained for later use.

4. We rename each variable using the standard algorithm (see [Cytron et al. 89]) such that each use of a variable has exactly one reaching definition.

5. We remove spurious ψ-functions; this may introduce simple assignments (e.g., $f_2 = f_1$) that can be removed using the standard algorithm.

The algorithms to compute the dominance frontiers, place ϕ-functions and to rename variables are as in the sequential case (see [Cytron et al. 89]). We record all nodes where ψ-functions are introduced, and then visit each node in step (5) to remove spurious ψ-functions.

4.1 Removing Spurious ψ-functions

A ψ-function introduces a new value name to indicate that the value of variable may have changed. Applying the standard SSA algorithms to PFG's may produce spurious ψ-functions. These arise when more than one value is visible at a merge point, and some subset of the values are not definitions within the parallel code leading to the merge point.

Determining if a ψ-function is spurious involves examining the reaching definitions of each argument of the function. We simply remove those arguments that do not correspond to parallel updates of the variable. If the resulting ψ-function has just one argument i.e., the variable is updated on only one path, then the ψ-function is spurious and can be removed. Spurious ψ-functions reduce to trivial assignments; this is done in order to preserve the SSA properties. Such trivial assignments are later removed by dead code elimination.

A *ϕ-region* corresponding to a parallel merge node X is a subgraph of the Parallel Flow Graph defined by all paths from $idom(X)$ to X. The node $idom(X)$ represents the closest node that joins all paths leading *Entry* to X. If a data value reaches node X, it is either generated by some predecessor of $idom(X)$ or in one of the paths from $idom(X)$ to X. At least one definition *must* be defined in the ϕ-region, otherwise no ϕ-function would have been placed at the node X. In the following we assume the ϕ-region has vertices V_r and edges E_r where V_r and E_r are the nodes and edges that appear on all paths from $idom(X)$ to X.

```
a_0 = 0
b_0 = 0
c_0 = 0
d_0 = 0
f_0 = 0.9
Parallel Sections
Section (A)
   a_1 = 0.5
   if (P) then
      b_1 = a_1**2
      f_1 = 1;
   else
      b_2 = a_1/(a_1+5)
   endif
   f_2 = φ(f_1, f_0);
   b_3 = φ(b_1, b_2)
Section (B)
   c_1 = 8
   d_1 = 3.9
   if (Q) then
      d_2 = d_1 + c_1**4
   endif
   d_3 = φ(d_2, d_1)
Section (C), Wait (A)
   a_2 = a_1 - 0.2
   f_3 = a_2 * b_3
End Parallel Sections
f_4 = ψ(f_3, f_0)
a_3 = ψ(a_2, a_1)
b_4 = ψ(b_3, b_0)
c_2 = ψ(c_1, c_0)
d_4 = ψ(d_3, d_0)
```

(a) SSA Form with Spurious ψ-functions

```
a_0 = 0
b_0 = 0
c_0 = 0
d_0 = 0
f_0 = 0.9
Parallel Sections
Section (A)
   a_1 = 0.5
   if (P) then
      b_1 = a_1**2
      f_1 = 1;
   else
      b_2 = a_1/(a_1+5)
   endif
   f_2 = φ(f_1, f_0);
   b_3 = φ(b_1, b_2)
Section (B)
   c_1 = 8
   d_1 = 3.9
   if (Q) then
      d_2 = d_1 + c_1**4
   endif
   d_3 = φ(d_2, d_1)
Section (C), Wait (A)
   a_2 = a_1 - 0.2
   f_3 = a_2 * b_3
End Parallel Sections
f_4 = f_3
a_3 = a_2
b_4 = b_3
c_2 = c_1
d_4 = d_3
```

(b) After removal of Spurious ψ-functions

Figure 6: SSA form of Example 1 with spurious ψ-function

Theorem 1 *Let $v_n = \psi(v_1, ..., v_{n-1})$ be a ψ-function at a parallel merge node X_n. Let X_i be the node corresponding to the reaching definition of v_i $\forall$ i = 1,2,...,n-1. Then,*

1. *Node X_i is in the ϕ-region of X_n only if idom(X_n) dominates X_i.*
2. *If idom(X_n) does not strictly dominate X_i then we can remove the argument corresponding to X_i (v_i) from the ψ-function.*

Proof: By definition of the ϕ-region of X_n, X_i is a node in the ϕ-region if and only if X_i is on some path from the entry point of the ϕ-region to X_n. By definition of dominance relation, $idom(X_n)$ dominates X_i. If $idom(X_n)$ does not strictly dominate X_i, then the value v_i is defined in $idom(X_n)$ or outside the ϕ-region. By definition of dominance frontier, a ψ-function would have been placed at X_n only if there exists some other definition, v_j, of variable v within the ϕ-region, excluding $idom(X_n)$. A ψ-function would have been placed only if there were an incoming consistency edge, meaning that the variable v_j reaches along a consistency edge to X_n. Parallel semantics require that values be made consistent along this consistency edge. Hence, v_j must reach the parallel merge point X_n; in other words, v_i will be eclipsed by v_j. Therefore, v_i can be removed from the ψ-function.

□

Once redundant value names have been removed from a ψ-function, it may be a candidate for removal:

Theorem 2 *When the application of theorem 1 has reached a fixpoint, and if a ψ-function at a parallel merge point has exactly one remaining argument, it is a spurious ψ-function and can be removed.*

Proof: By theorem 1, an argument of the ψ-function is removed if it is defined outside the ϕ-region under consideration. If there is only one argument remaining in the ψ-function after applying theorem 1 on all the arguments then the variable is defined exactly once inside the ϕ-region. Since a ψ-function is required at a parallel merge point only if there is a merge of

```
for each ψ-function, ψ_i do
    X ← node(ψ_i)
    n ← num-arguments(ψ_i)
    for each argument v_i of ψ_i do
        X_i ← node(v_i)
        if (idom(X) ⋡ X_i) then
            remove-argument(ψ_i, v_i)
        endif
        n ← n − 1
    endfor
    if ( n ≤ 1 ) then
        remove-ψ(ψ_i)
    endif
endfor
```

Figure 7: Algorithm to remove spurious ψ-functions

more than one update of the same variable inside the ϕ-region, this ψ-function is spurious and can be removed.

□

Figure 7 gives the algorithm to remove spruious ψ-functions; the algorithm uses theorems 1 and 2. The function `remove-argument` removes an argument from the corresponding ψ-function; `remove-`ψ replaces the ψ-function with a trivial assignment; `node(`ψ_i`)` returns the node that contains the ψ-function, ψ_i. The algorithm assumes knowledge of all the ψ-functions and the nodes they appear in the parallel SSA form of the program.

For example, in Figure 4(a), the *idom*((10)) is node (6); any value used by node (10) is either produced in node (6) or a predecessor of node (6). The value of `f` is modified in node (4), resulting in a ϕ-function in node (6). This is a standard ϕ-function introduced by the standard SSA algorithm. This value, f_2 reaches node (10) because section (C) has a `wait` clause for section (A). No ψ-function is needed because there is a single consistency edge reaching node (10); if the redundant consistency edge removed in §3 remained, the process in the next example would remove it.

Now, consider node (11); *idom*(11) is node (2). Figure 6(a) shows the ψ-function introduced after the **end parallel sections** statement by the standard SSA algorithm. The variable f is modified in node (10), and node (11) is in the dominance frontier of node (10), causing a ψ-function to be introduced; since this occurs in a parallel merge node, a ψ-function is used and considered for cleanup. The reaching definitions of arguments to this ψ-function originate prior to node (2) or in the left or right hand paths from node (2) to node (11); in particular, f_3 is defined in node (10) and f_0 is defined in node (1). In Figure 4(b), we see that node (10) is a child of node (2) in the dominator tree. This indicates the definition occurs in a path from node (2) to node (11), and can not be removed from the ψ-function. By comparison, f_0 is defined in node (1), which is not a child of node (2) in the dominator tree; this means the value f_0 is not defined within the ϕ-region, and can be removed from the ψ-function. This leaves us with the assignment $f_4 = \psi(f_3)$ which can be reduced to $f_4 = f_3$; the value name f_4 will eventually be renamed to f_3 by dead-code elimination using a standard SSA algorithm. Figure 6(b) shows the final SSA form before dead code elimination.

For this algorithm to be efficient, a rapid method is needed to determine parent/child relationships in the dominator tree. Once the tree is constructed, a total ordering can be applied. Each node, n, in the tree is labeled with its position in a left-to-right ($\mathcal{L}(n)$) and right-to-left ($\mathcal{R}(n)$) pre-order traversal of the tree. Given this ordering, node x dominates node y *iff* $\mathcal{L}(x) \leq \mathcal{L}(y)$ and $\mathcal{R}(x) \leq \mathcal{R}(y)$. After this construction, determining dominance relationships takes constant time.

5 Conclusion

We have described the Parallel Flow Graph, a simple data structure to handle explicitly parallel constructs with structured parallelism. The Parallel Flow Graph is similar to the the Program Execution Graph [Balasundaram & Kennedy 89] and the Synchronized Control Flow Graph [Callahan et al. 90]. We have shown how to compute Static Single Assignment form of such parallel programs using the Parallel Flow Graph by a simple extension to the standard SSA algorithms for sequential programs. The algorithms presented here differ from those

in [Srinivasan 91] in that they do not work on a recursive data structure and the concept of *factoring* used in [Srinivasan 91] is no longer needed here.

We plan to extend this work to handle the full range of synchronization and parallel statements available in PCF Fortran.

References

[Alpern et al. 88] Alpern, B., Wegman, M., and Zadeck, F. Detecting equality of variables in programs. In [POP 88], pages 1–11.

[Balasundaram & Kennedy 89] Balasundaram, V. and Kennedy, K. Compile-time detection of race conditions in a parallel program. In *Proc. 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[Callahan et al. 90] Callahan, D., Kennedy, K., and Subhlok, J. Analysis of event synchronization in a parallel programming tool. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 21–30, Seattle, Washington, March 1990. ACM Press.

[Cytron & Ferrante 87] Cytron, R. and Ferrante, J. What's in a name? (the value of renaming for parallelism detection and storage allocation). Technical Report RC 12785(#55984), IBM T.J. Watons Research Center, 1987.

[Cytron et al. 89] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, K. An efficient method of computing static single assignment form. In *Conf. Record 16th Annual ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, TX, January 1989.

[Parallel Computing Forum 91] Parallel Computing Forum. Pcf fortran. *Fortran Forum*, 10(3), September 1991. (special issue).

[POP 88] *Conf. Record 15th Annual ACM Symp. Principles of Programming Languages*, San Diego, CA, January 1988.

[Rosen et al. 88] Rosen, B., Wegman, M., and Zadeck, F. Global value numbers and redundant computations. In [POP 88], pages 12–27.

[Srinivasan 91] Srinivasan, H. Analyzing programs with explicit parallelism. M.S. thesis 91-TH-006, Oregon Graduate Institute of Science and Technology, Dept. of Computer Science and Engineering, July 1991.

[Wegman & Kenneth Zadeck 85] Wegman, M. N. and Kenneth Zadeck, F. Constant propagation with conditional branches. In *Conf. Record 12th Annual ACM Symp. Principles of Programming Languages*, pages 291–299, January 1985.

[Wolfe & Srinivasan 91] Wolfe, M. and Srinivasan, H. Data structures for optimizing programs with explicit parallelism. Technical report, Oregon Graduate Institute, February 1991. To appear in the *First International Conf. of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.