Data Flow Equations for Explicitly Parallel Programs

Dirk Grunwald and Harini Srinivasan
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

CU-CS-605-92 July 1992

Data Flow Equations for Explicitly Parallel Programs

Dirk Grunwald*and Harini Srinivasan
Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, CO - 80309

July 1992

Abstract

We have extended the standard monotone dataflow system for the *reaching definitions* problem to accommodate explicitly parallel programs; this information is used in many standard optimization problems. This paper considers the `parallel_sections` construct, both with and without explicit synchronization; a future paper considers the `parallel_do` construct. Although work has been done on analyzing parallel programs to detect data races, little work has been done on optimizing such programs; the equations in this paper should form the basis for extensive work on optimization.

1 Introduction

In this paper, we describe a *data flow framework* [1] for computing the reaching definition information in parallel programs with a view of being able to optimize such programs. It is desirable to be able to perform classical code optimization on parallel programs for two main reasons: first, parallel programming and the use of explicitly parallel constructs is becoming common in scientific and numerical programs. Second, we are interested in being able to achieve good execution rates of such programs on high performance architectures. Though extensive work has been done on analyzing parallel programs for potential *races* [6, 4, 2], little work has been done on analyzing parallel programs with an aim of being able to perform code optimizations. Midkiff and Padua [7] point out the difficulties in optimizing explicitly parallel programs. Our work focuses on developing an intermediate representation for optimizing parallel programs.

Reaching definition information, i.e., the set of definitions reaching each use of a variable in a program is vital for various code optimizations; some of them include constant propagation, induction variable analysis, common subexpression elimination, dead code elimination etc. Most

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

```
(1)     j = 0
(1)     k = 1
(2)     loop
(3)         if (condition) then
(4)             j = j + 1
            else
(5)             k = 5
(6)         endif
(6)         l = k + 4
(7)     endloop
```

(a) Sequential Program

```
(1)     j = 0
(1)     k = 1
(2)     loop
(3)         Parallel Sections
(4)         Section A
(4)             j = j + 1
(5)         Section B
(5)             k = 5
(6)         End Parallel Sections
(6)         l = k + 4
(7)     endloop
```

(b) Parallel Program

Figure 1: Example sequential and parallel programs.

commercial compilers use the bit vector intermediate representation, and we feel the extensions presented in this paper can be easily incorporated in compilers to analyze explicitly parallel programs.

In our work we consider the parallel extensions to FORTRAN as specified by the Parallel Computing Forum [9], which is the basis of the ANSI committee X3H5 standardization effort. As mentioned earlier, the performance of parallel programs on existing and future high performance architectures depends to a great extent on the ability to perform aggressive code optimizations, particularly scalar optimizations across parallel constructs. Most of the existing compilers for parallel programs do not perform scalar optimizations across parallel constructs. Instead, they restrict optimizations to specific sequential sections of code in the parallel program.

Consider the sequential and parallel programs in Figure 1; these two programs have very similar control flow structures. The variable '`j`' in 1(a) is not an induction variable, because the `if .. then` may not be executed for each iteration of the loop. However, in the parallel program, '`j`' *is* an induction variable since both branches of the `Parallel Sections` statement always execute for all iterations of the loop, but this could not be automatically detected without adequate dataflow information. Detecting such induction variables is useful for strength reduction, data dependence analysis and other optimizations. Likewise, dataflow information would show that the variable '`k`' has the value 5 at the end of the parallel construct during each iteration.

In section 2, we explain data flow analysis and explain how the reaching definition information is computed in sequential programs. Section 3 describes the parallel constructs and the semantics of the parallel constructs considered in this paper. In Section 4, we describe data structures

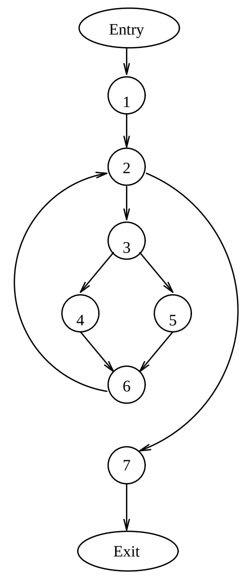

Figure 2: Control Flow Graph for the sequential program in Figure 1.

to represent such constructs. Section 5 extends the data flow equations for computing reaching definition information in sequential programs, to handle parallel constructs and in Section 6 we extend these equations to handle synchronization in the form of `post/wait` statements.

2 Global Data Flow Analysis

The problem of global data flow analysis can be explained as follows [8]: given the control flow structure, we must discern the nature of the data flow (which definitions of program quantities can affect which uses) within the program. Data flow problems are often posed as a system of equations based on the Control Flow Graph of the program. A Control Flow Graph, CFG, of a program is a directed graph, $\langle V, E, V_0 \rangle$, where V is the set of vertices representing basic blocks in the program, E is the set of edges representing flow of control in the program and V_0 is the unique node representing the entry into the program. We say a node P is the predecessor of node Q if there is an edge in the CFG from P to Q. For each vertex in the CFG, we define some *basic* attributes, which can be defined unambiguously from an analysis of the program. Then we define *inherited* and *synthesized* attributes in a set of data flow equations, and solve these equations.

This section discusses the data flow equations to solve the reaching definitions problem in sequential programs [1]. The reaching definition problem is to find the set of definitions of a variable '`v`' that can reach a particular *use* of '`v`'. This is also referred to as the *ud-chaining* problem in the literature. In the later sections of the paper, we explain how these equations can be extended to

Node	*Gen*	*Kill*	*In*	*Out*
Entry	{ }	{ }	{ }	{ }
(1)	j_1, k_1	j_4, k_5	{ }	j_1, k_1
(2)	{ }	{ }	j_1, k_1	j_1, k_1
(3)	{ }	{ }	j_1, k_1	j_1, k_1
(4)	j_4	j_1	j_1, k_1	j_4, k_1
(5)	k_5	k_1	j_1, k_1	j_1, k_5
(6)	l_6	{ }	j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5
(7)	{ }	{ }	j_1, k_1	j_1, k_1
Exit	{ }	{ }	j_1, k_1	j_1, k_1
Entry	{ }	{ }	{ }	{ }
(1)	j_1, k_1	j_4, k_5	{ }	j_1, k_1
(2)	{ }	{ }	j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5
(3)	{ }	{ }	l_6, j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5
(4)	j_4	j_1	l_6, j_1, k_1, j_4, k_5	l_6, j_4, k_1, k_5
(5)	k_5	k_1	l_6, j_1, k_1, j_4, k_5	l_6, j_1, j_4, k_5
(6)	l_6	{ }	j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5
(7)	{ }	{ }	j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5
Exit	{ }	{ }	j_1, k_1, j_4, k_5	l_6, j_1, k_1, j_4, k_5

Table 1: Table showing two iterations of the data flow equations to solve the reaching definitions problem for the sequential program in Figure 1(a).

solve the reaching definitions problem across parallel constructs and synchronization constructs in explicitly parallel programs.

2.1 Reaching Definitions

We say a definition of a variable '`v`' *reaches* a point p in the program if there is a path in the CFG from that definition to p, such that no other definitions of '`v`' appear on the path. To determine the definitions that can reach a given point in a program, we first assign a distinct number to each definition. Our problem is to be able to find for each node n of the CFG, *In*(n), the set of definitions that reach the beginning of n.

Formally, a definition d of a variable name '`v`' reaches a node n if there is a path $n_1, n_2, \ldots, n_k, n$ in the flow graph such that

1. d is within n_1,
2. d is not subsequently killed in n_1 (i.e., '`v`' is not redefined) and
3. d is not killed in any of $n_2, \ldots, n_k$.

One way of calculating *In*(n) is to determine all generated definitions and then to propagate each definition from the point of generation to n. An easy way of doing this is to solve the following set of 2N simultaneous equations for a CFG of N nodes:

$$\begin{aligned} Out(n) &= (In(n) - Kill(n)) \cup Gen(n)] \\ In(n) &= \bigcup_{\mathrm{p} \in \mathrm{pred(n)}} Out(p)\,. \end{aligned}$$

Out(n) is similar to *In*(n) but pertains to the point immediately after the end of the basic block. *Kill*(n) is the set of definitions outside of n that define variables that also have definitions within n and *Gen*(n) is the set of definitions generated within n that reach the end of n. We are interested in the smallest solution possible for *In*, which is why we start with *In* as the empty set for all n. The algorithm that computes the *In* sets starts with this initial approximation and iterates through the above set of equations until a fixpoint is reached. This particular set of equations and the iterative algorithm form a monotone dataflow system. In such a system, the order of traversal of the CFG only affects the convergence rate of the different sets to their fixpoint. It has been proven that a depth first traversal of the CFG helps reduce the number of iterations to five in most practical cases [1].

The CFG for the sequential program in Figure 1 is given in Figure 2. Variable '`j`' is defined at nodes (1) and (4); call these j_1 and j_4 respectively. The reaching definitions for the use of '`j`' at node (6) are j_1 and j_4. The *In*, *Out*, *Kill* and *Gen* sets for the different nodes are given in Table 1. This table shows two iterations of the data flow equations; the third iteration is the same as the second, indicating that a fixpoint has been reached.

This example illustrates how the *In* and *Out* sets are computed for sequential programs, given the *Gen* and *Kill* sets. In the following sections, we derive an analogous procedure for explicitly parallel programs.

3 Parallel Constructs

In this paper, we only consider the **`Parallel Sections`** construct [9]. The **`Parallel Sections`** construct is used to specify parallel execution of explicitly identified sections of code. Each section of code is interpreted as a parallel thread, and must be data independent except where an appropriate synchronization mechanism is used. The **`Parallel Sections`** construct can also be nested, appear in the body of a loop and so on.

We consider synchronization between threads in the form of *event synchronization*, described by a binary *event variable*. Operations are available to indicate that an event has occurred (**`post`**), to

```
(Entry)     event( ev )
(Entry)     x = 2
(Entry)     y = 5
(1)         loop
(2)             Parallel Sections
(3)             Section A
(3)                 if (condition) then
(4)                     x = 7
(4)                     post( ev )
                    else
(5)                     x = 8
(5)                     post( ev )
(6)                 endif
(6)                 z = y * 7
(7)             Section B
(7)                 Parallel Sections
(8)                 Section B1
(8)                     wait( ev )
(8)                     x = x * 32
(9)                 Section B2
(9)                     z = y * 54
(10)                End Parallel Sections
(11)            End Parallel Sections
(11)            y = x * z
(12)        endloop
```

Figure 3: Parallel Program with `Parallel Sections` and event synchronization

ensure that an event has occurred (`wait`), and to indicate that an event has not occurred (`clear`). In our work, we only consider `post` and `wait` statements.

When a `post` statement is executed, the appropriate shared variables are made consistent and the value of the event is set to "posted", no matter what its value was previously. When a `wait` statement is executed, the appropriate shared variables are made consistent and the thread waits for the event to be marked "posted".

An example parallel program with `Parallel Sections` construct and event synchronization is shown in Figure 3. Section `A` and `B` execute in parallel. Within section `B`, there is a nested `Parallel Sections` construct where sections `B1` and `B2` can execute in parallel. The event variable '`ev`' will be posted in one of the branches of the if-construct, depending on the value of '`condition`'. The execution of Section `B1` can not proceed until at least one of the `post` occurs. Note that the `Parallel Sections` is inside a loop. This example is purely illustrative; in particular, the event variable '`ev`' is not cleared between iterations of the loop, and thus, this example would not execute properly. We refer to this example in §6, to show the interaction of loops and synchronization variables. Note that this is a *sequential* loop; analysis of parallel loops is a topic of future papers.

The language standard does not define the memory consistency model for the target architecture. Rather, it allows a range of implementations including copy-in/copy-out semantics. We assume copy-in/copy-out semantics in the compiler, because it provides more opportunity for optimization. For example, within a single thread, we are free to load copies of variable values into registers or propagate subexpressions and the like, disregarding the actions of other threads. This does not imply that we implement a pure copy-in/copy-out program. Rather, we use this as one model of memory consistency because it is convenient for compiler optimizations and allowed by the language standard. Correct programs should obey copy-in/copy-out semantics as well as other memory consistency models allowed by the language standard.

At a *fork* point, i.e., a `Parallel Sections` statement, every branch of the fork (each thread) gets its own copy of the shared variables. Each thread modifies its own local copy and at the *join* point, i.e., the `End Parallel Sections` statement, the copies from the different threads are merged with the global values. In the presence of post/wait synchronization, the thread that waits for an event to occur updates its copy with the values from all the threads posting that event. Multiple copies of a variable may *potentially* reach a wait statement, either because of multiple posts executed by different threads or because of one or more posts (executed by different threads) and the `wait`ing thread defines that variable prior to the the `wait` statement. Some decision has to be made at run time as to which value will reach the `wait` statement. However, at the compiler level, we allow more than one value to reach that point and the presence of multiple values at such wait statements indicates *potential* anomalies.[1]Similarly at a join node, multiple values for a variable reaching that node indicates a potential anomaly in the `Parallel Sections` construct.

4 Parallel Flow Graph

In this section, we describe the *Parallel Flow Graph*[12], a data structure used to represent control flow, parallelism and synchronization in explicitly parallel programs. The Parallel Flow Graph (PFG) is similar to the Synchronized Control Flow Graph [4] and the Program Execution Graph [2]. A PFG is basically a directed graph with nodes representing extended basic blocks in the program and edges representing either sequential control flow, parallel control flow or synchronization. An *extended basic block* is a basic block that may have at most one `wait` statement at the start of the basic block and at most one `post` or branch statement at the end of the basic block. A *sequential control flow edge* represents sequential flow of control within sequential parts of the program. A *parallel control flow edge* represents parallel control flow, as at fork and join points in the program. Finally, a *synchronization edge* is an edge from a `post` statement to a corresponding `wait` statement.

The PFG for the parallel program in Figure 3 is shown in Figure 4. Nodes (2) and (7) represent fork nodes and nodes (11) and (10) are the respective join nodes. Sequential, parallel and synchronization edges are identified in this figure as indicated.

5 Data Flow Equations for `Parallel Sections`

In Section 2, we reviewed the data flow equations from [1] to compute the reaching definition information at any point in a sequential program. In this Section, we extend these equations to handle the `Parallel Sections` construct. The extensions are based on the following fundamental concepts:

- At parallel branch points, such as fork nodes, *all* the branches execute; in the case of sequential branch points, e.g., if-statements, only *one* of the branches will be executed.
- A value defined at a point prior to a parallel construct does not reach the corresponding parallel merge point if it is *always* killed in *at least one* of the branches. In contrast, for sequential branches, the value would need to be always killed along *every* branch.
- The compiler must assume that a conditionally defined value in a parallel section may reach the parallel merge point. These definitions do not kill the definitions prior to the `Parallel Sections` statement. In actuality, only one definition reaches the merge point, but determining the actual reaching definition is undecidable. Thus, the compiler must be conservative and assume that both definitions reach.

[1]Note that multiple values reaching a wait statement do not necessarily mean there *are* anomalous updates; for example, the `post` statements may have been conditionally executed.

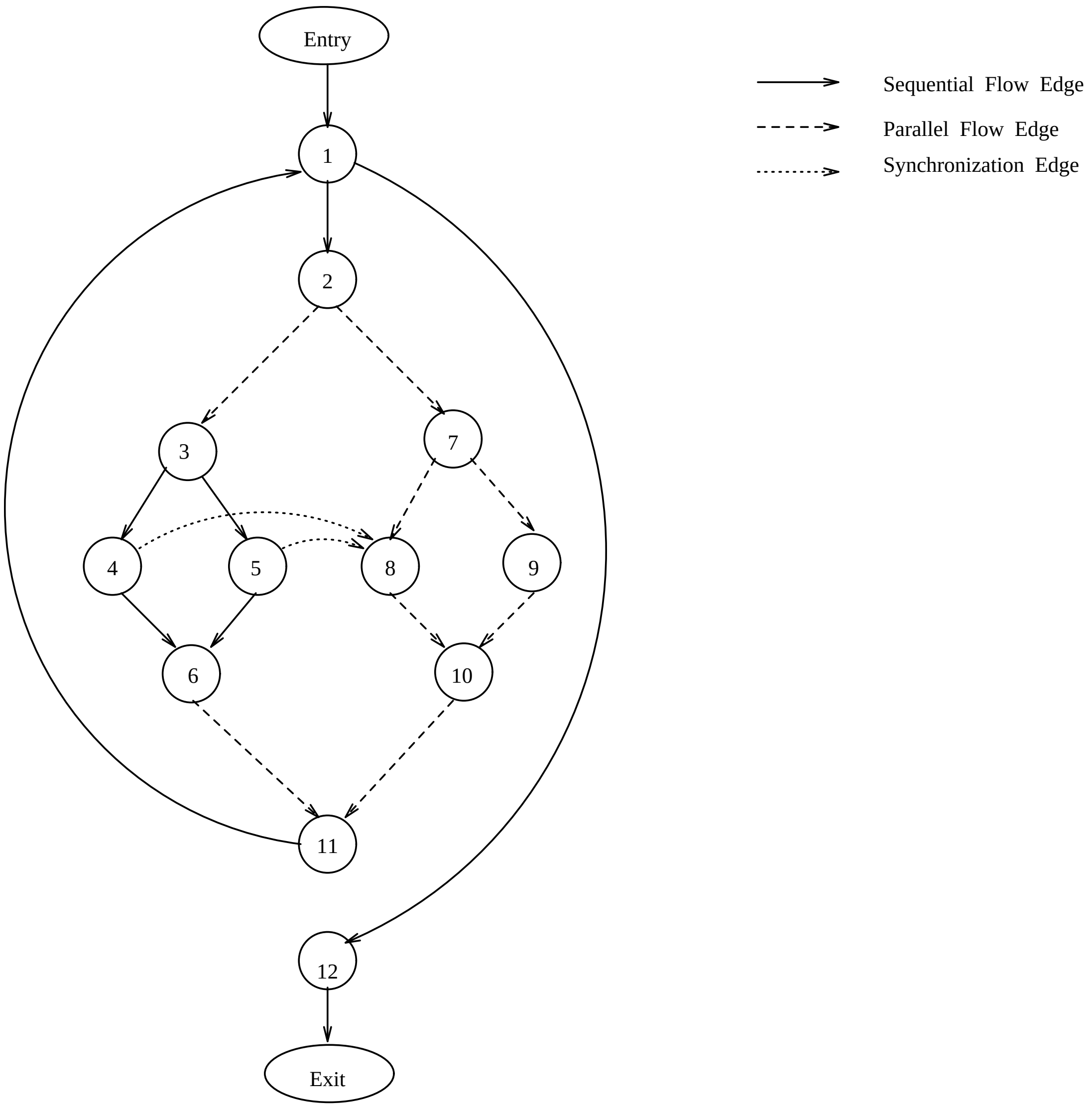

Figure 4: Parallel Flow Graph for the example parallel program.

```
(1)   a = 0
(1)   b = 1
(2)   if (condition) then
(3)       a = a + 1
(3)       b = 7
      else
(4)       b = 5
(4)   endif
(5)   c = a * b
```

(A) Example Sequential Program

```
(1)   a = 0
(1)   b = 1
(2)   Parallel Sections
(3)   Section A
(3)       a = a + 1
(3)       b = 7
(4)   Section B
(4)       b = 5
(4)   End Parallel Sections
(5)   c = a * b
```

(B) Example Parallel Program

Figure 5: Example sequential and parallel programs.

```
(1)      a = 0
(1)      b = 1
(1)      c = 2
(2)      Parallel Sections
(3)      Section A
(3)          a = a + 1
(3)          b = 7
(4)      Section B
(4)          Parallel Sections
(5)          Section B1
(5)              b = 5
(6)          Section B2
(6)              if (P) then
(7)                  c = 6
(8)              endif
(9)          End Parallel Sections
(10)     End Parallel Sections
(10)     d = a * b + c
```

Figure 6: Example parallel program to illustrate data flow equations.

These concepts are illustrated by the sequential and parallel programs in Figure 5 and by the program in Figure 6 on page 10. The values of the variable '`a`' reaching the sequential and parallel merge points (i.e., the **`endif`** and **`End Parallel Sections`** statement respectively) in Figure 5 are different. In the case of the sequential program, the values of the variable '`a`' reaching the **`endif`** statement is either the value defined before the **`if`** test or the value defined in the **`then`**-part of the **`if`**-construct. However, at the parallel merge point, the only reaching value of '`a`' is the value defined in **`Section A`**. In Figure 6, the variable '`c`' is defined conditionally in **`Section B`**. Therefore, this value and the value of '`c`' defined prior to the outer **`Parallel Sections`** construct reach the parallel merge points. The sequential data flow equations (Section 2) will not be able to handle such cases. The new data flow equations for parallel programs must still be able to say that the values of '`b`' in Figure 5 reaching the join node are either from **`Section A`** or **`Section B`**. As mentioned earlier, more than one value of a variable reaching a parallel merge point indicates a potential anomaly in the program.

To handle the above situation, we introduce two new sets to the data flow framework of Section 2: the *ACCKillin* and *ACCKillout* sets. These sets accumulate definitions that occur outside a parallel construct and that are killed along specific parallel branches in the parallel construct. The *ACCKillin* set at a node is the set propagated by its predecessors and *ACCKillout* set at the node is its *ACCKillin* set updated by the definitions killed in this node, excluding the definitions generated in this node, i.e., its *Kill* set minus the *Gen* set. In our first example (Figure 5), the accumulated kill set at the end of **`Section A`** is the value of '`a`' defined prior to the parallel construct because the definition of '`a`' inside **`Section A`** will always kill the previous definition.

Parallel sections can be nested, but the information represented by the *ACCKillout* set pertains to a *single* parallel block. For example, in Figure 5, the *ACCKillin* set at the entry to the parallel program is empty. At node (1), the *ACCKillout* set includes 'a_3' since it is in its *Kill* set. However, if we propagate this set via **`Section B`**, that does not define '`a`', to the parallel merge node, the *ACCKill* set at this node will contain this definition. However, 'a_3' always reaches the parallel merge point and should not be in the *ACCKill* set of any of its parallel predecessors. Therefore, we clear *ACCKillout* at fork nodes and use this empty set in computing the accumulated kill sets inside the corresponding parallel block. We must also preserve the current value across internal nested parallel blocks because a join node must have access to the *ACCKillout* set from the corresponding fork node. Thus, fork nodes store the *ACCKillout*, computed from its *Gen* and *Kill* sets in another set, *ForkKill*, and a 'technical edge' between corresponding fork and join nodes makes this information available to the join node. At join nodes, the *In* set will exclude definitions from the *ACCKillout* sets of *all* the parallel predecessors of this node.

We propagate the *ACCKill* sets by computing the *ACCKillin* set at a merge node as the union of the *ACCKillout* sets of its parallel predecessors and the intersection of the *ACCKillout* sets of its sequential predecessors.

In sequential programs, we define *Kill*(n) to be the set of all the definitions of variables outside n for those variables defined in n; these are the definitions that will be overridden when the variable is defined in node n. This is appropriate for sequential programs or a single thread of control because assignments can not occur in parallel.

By comparison, in the case of parallel programs, where we can have multiple simultaneous threads of execution, we distinguish between the *Kill* set and the *ParallelKill* set. The *Kill* set for node n contains all killed definitions from nodes that can not execute at the same time as node n. Similarly, the *ParallelKill* for n contains all definitions from nodes that *can* execute at the same time. For example, in Figure 5(B), the *Kill* set of `section B` contains the definition '$\mathtt{b}_1$' (the definition of b from node 1), while the *ParallelKill* set contains the definition '$\mathtt{b}_3$'.

We would expect both definitions '$\mathtt{b}_3$' and '$\mathtt{b}_4$', but *not* '$\mathtt{b}_1$', to reach the join node (node 5). Definition '$\mathtt{b}_1$' should not reach because there are assignments to '`b`' that are guaranteed to occur later in the execution order. Both '$\mathtt{b}_3$' and '$\mathtt{b}_4$' should reach the join node because the compiler can not assume a particular execution order or memory semantics. Indeed, this indicates a potential data anomaly or race condition in this particular program. We segregate the kill sets into *Kill* and *ParallelKill* sets to distinguish between these cases. *ParallelKill*(n) can be computed by traversing the PFG and including those definitions d_i of variables '`v`' such that '`v`' has a definition in n and d_i occurs in a node that *can* execute in parallel with n. This can be done by traversing the parallel flow edges and the sequential flow edges in all `Section`s that have the same fork node and join node as the `Section` S_n corresponding to n but not S_n itself. Thus, as in the sequential dataflow problem, *Kill* and *ParallelKill* can be computed directly and need not be computed using an iterative algorithm.

The *ACCKill* sets accumulate information about definitions that are killed *within* a sequential thread, and we include the *Kill* sets in the *ACCKillin* and *ACCKillout* sets. We do not include the *ParallelKill* set, because that set represents information about other threads where the temporal ordering of definitions is undefined. When computing the *Out* set for each node, we must consider all killed definitions, i.e. the union of the *Kill* and *ParallelKill* sets.

The data flow equations for the reaching definitions problem in programs that have the `Parallel Sections` construct is given in Figure 7. In those equations, *par_pred* refers to the set of parallel flow predecessors of the node; *seq_pred* refers to the set of sequential flow predecessors of the node and *pred* refers to the set of all predecessors (both parallel and sequential flow predecessors) of the node. The reaching definition information,i.e., the *In* set at each node, is defined by the fixpoint of the equations in Figure 7.

$$
\begin{aligned}
\mathit{Out}(n) &= \mathit{In}(n) - \mathit{Kill}(n) - \mathit{ParallelKill}(n) \cup \mathit{Gen}(n)\\
\mathit{In}(n) &= \bigcup_{\mathrm{p} \in \mathrm{pred(n)}} \mathit{Out}(p) - \bigcup_{\mathrm{p} \in \mathit{par\_pred}(\mathrm{n})} \mathit{ACCKillout}(p)
\end{aligned}
$$

$$
\mathit{ACCKillout}(n) = \begin{cases} \emptyset & (n \text{ is a fork node})\\ (\mathit{ACCKillin}(n) + \mathit{Kill}(n)) - \mathit{Gen}(n) & (n \text{ is a join node, with corre-}\\ \quad +(\mathit{ForkKill}(f) - \mathit{Out}(n)) & \text{sponding fork node } f)\\ (\mathit{ACCKillin}(n) + \mathit{Kill}(n)) - \mathit{Gen}(n) & (\text{otherwise}) \end{cases}
$$

$$
\mathit{ACCKillin}(n) = \bigcup_{\mathrm{p} \in \mathit{par\_pred}(\mathrm{n})} \mathit{ACCKillout}(p) + \bigcap_{\mathrm{p} \in \mathit{seq\_pred}(\mathrm{n})} \mathit{ACCKillout}(p)
$$

$$
\mathit{ForkKill}(n) = \begin{cases} (\mathit{ACCKillin}(n) + \mathit{Kill}(n)) - \mathit{Gen}(n) & (n \text{ is a fork node})\\ \emptyset & (\text{otherwise}) \end{cases}
$$

Figure 7: Dataflow Equations for Programs with Parallel Sections

For the parallel program given in Figure 6, the *In*, *Out*, *Gen*, *Kill*, *ParallelKill* and the accumulated kill sets are given in Figure 8. The system of equations converges on the second iteration. The figure shows the first iteration (which is the same as the second). Note that $\mathit{ACCKillout}(10)$ contains b_1. This indicates that b_1 is killed by one or more of the parallel branches – in this case, it is killed by both sections `A` and `B` (via `Section B1`). By comparison, even though '`c`' is defined in node 7, the definition is conditional on '`P`', and thus c_1 does not appear in $\mathit{ACCKillout}(10)$. The set $\mathit{Out}(10)$ contains definitions b_3 and b_5, indicating a potential anomaly. In the case of '`b`', this is an actual anomaly.

In the next section, we extend these data flow equations to consider event synchronization between parallel `Sections`.

6 Including the effect of Synchronization

We extend the data flow equations in the previous section to consider event synchronization by using the *preserved sets* formulation given in [3]. Synchronization using `post`/`wait` occurs between different threads that execute in parallel. Synchronization edges carry data flow information, i.e., they propagate values of variables from the thread that posted the event to the thread that is waiting for the event to be posted. According to [9], it must be *possible* to execute each `post` before

Node	Gen	Kill	ParKill
1	$\{a_1, b_1, c_1\}$	$\{a_3, b_3, b_5, c_7\}$	
2			
3	$\{a_3, b_3\}$	$\{a_1, b_1\}$	$\{b_5\}$
4			
5	$\{b_5\}$	$\{b_1\}$	$\{b_3\}$
6			
7	$\{c_7\}$	$\{c_1\}$	
8			
9			
10	$\{d_{10}\}$		

Node	In	Out	ACCKillIn	AccKillOut	ForkKill
1		$\{a_1, b_1, c_1\}$		$\{a_3, b_3, b_5, c_7\}$	
2	$\{a_1, b_1, c_1\}$	$\{a_1, b_1, c_1\}$	$\{a_3, b_3, b_5, c_7\}$		$\{a_3, b_3, b_5, c_7\}$
3	$\{a_1, b_1, c_1\}$	$\{a_3, b_3, c_1\}$		$\{a_1, b_1\}$	
4	$\{a_1, b_1, c_1\}$	$\{a_1, b_1, c_1\}$			
5	$\{a_1, b_1, c_1\}$	$\{a_1, b_5, c_1\}$		$\{b_1\}$	
6	$\{a_1, b_1, c_1\}$	$\{a_1, b_1, c_1\}$			
7	$\{a_1, b_1, c_1\}$	$\{a_1, b_1, c_7\}$		$\{c_1\}$	
8	$\{a_1, b_1, c_1, c_7\}$	$\{a_1, b_1, c_1, c_7\}$			
9	$\{a_1, b_5, c_1, c_7\}$	$\{a_1, b_5, c_1, c_7\}$	$\{b_1\}$	$\{b_1\}$	
10	$\{a_3, b_3, b_5, c_1, c_7\}$	$\{a_3, b_3, b_5, c_1, c_7, d_{10}\}$	$\{a_1, b_1\}$	$\{a_1, b_1\}$	

Figure 8: Data Flow Sets for one iteration on the parallel program in Figure 7.

its corresponding `wait` for a parallel program to be deadlock free and correct. If the `post` statement at a node n_p always executes before the corresponding `wait` node, n_w[2], then n_w will have to update its reaching definitions information with that from the *Out* set of the node corresponding to the `post`. Apart from updating the reaching definitions information, i.e., the *In* set at the `wait` node, n_w, we also want to be able update its accumulated kill sets, e.g., if n_w defines a variable, then all definitions of that variable reaching n_w via synchronization edges from `post` nodes, n_p, that always execute before n_w, must be included in the *ACCKillin* set of n_w. This is important because the definitions propagated by such synchronization edges are killed by the corresponding definition in n_w.

If there is a synchronization edge from n_p to n_w, we can not say that n_p always executes before n_w. It is possible that there are multiple posts of the same event variable and multiple waits for the same event variable. It is also possible that these multiple posts and waits are executed conditionally. Thus, a synchronization edge does not always imply an execution order. We are, however, interested in the *potential* execution order for computing the reaching definition information. *Preserved sets*, as defined in [3] give precisely the set of nodes that execute before a given node, defined as follows:

Definition 1 *A node $n_j \in$ Preserved(n_i) if and only if for all parallel executions x, if n_j and n_i are both executed, n_j is completed before n_i is begun.*

However, Callahan and Subhlok [3] have shown that computing this information is Co-NP Hard and have given a data flow framework to compute a conservative approximation to the Preserved sets. The approximate Preserved sets are computed as the least fixpoint of a set of data flow equations over the Parallel Flow Graph. The Preserved set for a block is defined as a function of its control flow (parallel and sequential) and synchronization predecessors.

Clearly, by using the Preserved set formulation, we can determine at a `wait` node n_w if a `post` node, n_p, always completes execution before n_w begins, i.e., if $n_p \in \text{Preserved}(n_w)$. We use a new data flow set, called the *SynchPass* set, that propagates definitions via synchronization edges.

If n_p executes before n_w, we propagate the definitions from n_p to n_w (and thus all nodes that execute after n_w in the same thread), because we know those definitions must have occurred before the synchronization occurred. Any definitions that occur in node n_w (and nodes subsequently executed by that thread) will kill the previous definitions in the thread executing n_w. These definitions will also kill any definitions that occur *before* n_p executes in the thread corresponding to n_p, but not necessarily those definitions occurring *after* n_p executed in that thread (e.g., because the thread executing n_p may have already completed execution, as it does not wait for the `wait`

[2]We say a `wait` node starts executing when the wait statement is successful and the code following the `wait` in this node starts execution

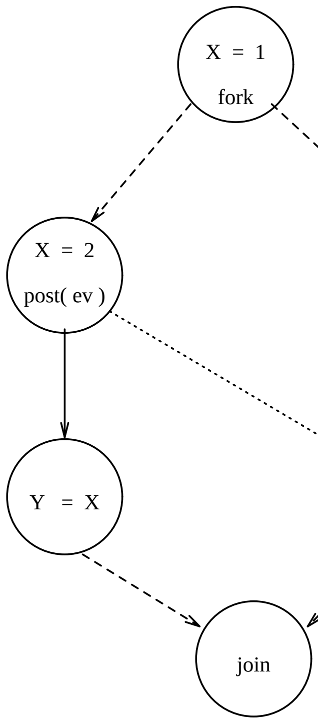

Figure 9: Synchronization Example

statement to occur). This means that the join node must realize that the definitions in n_w occur *after* the definitions passed in from n_p; this is the role of the *ACCKill* sets.

For example, consider the **`Parallel Sections`** in the PFG shown in Figure 9. The fork node defines a value for '`x`'. This value reaches the predecessor of the wait node and the post node. The definition in the fork node is in the *ACCKillout* set for the post node, indicating that some branch of the **`Parallel Section`** has killed that value. However, only the value from the wait node should reach the join node, because that definition *must* occur after the assignment in the post node and the fork node. We get the execution ordering information from the Preserved set. The value of '`Y`' following the post node is not specified by the language definition. One could argue that '`Y`' should have the value '3'; however, we have chosen to assume copy-in/copy-out semantics, and would thus believe that '`Y`' has the value '2' – ideally, an error message would be issued concerning this data race. For this example, the data flow formulation for Preserved sets given in [3] will be able to determine the Preserved sets of the wait node accurately. However, since this data flow formulation is conservative, we may not be always able to compute the *exact* Preserved sets for any node. This would result in a conservative approximation to the reaching definitions information in our data flow framework. For example, in the absence of the Preserved sets information in figure 9, we would derive the *Out* set of the join node to contain the definitions from both the post and the wait node. This is a conservative, yet correct, approximation to the reaching definition information at the join node. In the worst case, the effect of synchronization is lost at parallel merge points, i.e., in the absence of any Preserved set information our data flow equations would compute multiple reaching definitions at respective parallel merge nodes. This simply reduces the opportunity or effectiveness of some optimizations.

Therefore, at `wait` nodes, we update the *SynchPass* set with the *Out* set from the corresponding synchronization predecessors in the Preserved set of this node, indicating that the definitions from those predecessors have occurred. In order to propagate the *SynchPass* information to other nodes after a wait node, we want to consider the *union* of the *SynchPass* from all the parallel predecessors (since all these predecessors always execute) and the *intersection* of the *SynchPass* from the sequential predecessors (since only one of them executes).

We update the *ACCKillin* set of each node with the definitions of variables that are propagated by synchronization edges (i.e. *SynchPass*). We only consider the definitions of *SynchPass* also defined in this node. To do this, we use the set *OtherDefs*(n), or the definitions in the program outside of n that define variables that also have definitions within n.

The data flow equations taking into account **`Parallel Sections`** constructs with event synchronization is given in Figure 10. In this figure, *synch_pred* refers to synchronization predecessor.

Figures 11 and 12 show the data flow sets for the first two iterations for the parallel program in Figure 3; the fix point is reached in the third iteration. The Preserved set of node (8) (the wait node) is the set {Entry, 1, 2, 3, 4, 5, 7}, since each of these nodes always completes execution before node (8), if they execute at all. The reaching definition information in this figure has been computed using the Preserved set information. The definitions, '$\mathtt{x}_4$' and '$\mathtt{x}_5$' will not reach the join node, (11), because the definition '$\mathtt{x}_8$' always executes after '$\mathtt{x}_4$' and '$\mathtt{x}_5$'. It is this information on execution order that we borrowed from the Preserved set formulation. Also, the *ACCKillout* set of (11) includes '$\mathtt{x}_4$' and '$\mathtt{x}_5$'. This information was propagated to node (8) by the synchronization edges since (4) and (5) were in the Preserved set of (8). The definitions '$\mathtt{z}_6$' and '$\mathtt{z}_9$' reach the merge node (11); this is an indication of a potential anomaly in the program since the two definitions occur in distinct parallel branches, i.e., threads that can execute in parallel. The importance of the *ParallelKill* set is seen in the *Out* set of nodes (6) and (9). Even though the corresponding *In* sets have both definitions of '`z`', only the definition in that node should be in its *Out* set. The reason the *In* set of (6) and (9) both have '$\mathtt{z}_6$' and '$\mathtt{z}_9$' is because of the loop around the parallel block. Since we exclude the *ParallelKill* set from the *Out* set, we are able to compute the correct *Out* sets; for example, the *Out* set of (6) does not contain '$\mathtt{z}_9$' since this definition is in its *ParallelKill* set.

$$SynchPass(n) = \begin{cases} \bigcup\limits_{p \,\in\, synch\_pred \,\wedge\, p \,\in\, \text{Preserved(n)}} Out(p) & \text{(if } n \text{ is a wait node)} \\ \bigcup\limits_{p \,\in\, par\_pred} SynchPass(p) + \bigcap\limits_{p \,\in\, seq\_pred} SynchPass(p) & \text{(otherwise)} \end{cases}$$

$$Out(n) = \begin{cases} (In(n) - Kill(n) - ParallelKill(n) \cup Gen(n)) - \\ \quad (OtherDefs(n) \cap SynchPass(n)) \end{cases}$$

$$In(n) = \begin{cases} \bigcup\limits_{p \,\in\, pred\text{(n)}} Out(p) - \bigcup\limits_{p \,\in\, par\_pred\text{(n)}} ACCKillout(p) - \bigcap\limits_{p \,\in\, synch\_pred\text{(n)}} ACCKillout(p) \end{cases}$$

$$ACCKillout(n) = \begin{cases} \emptyset & (n \text{ is a fork node}) \\ (ACCKillin(n) + Kill(n)) - Gen(n) & (n \text{ is a join node, with corre-} \\ \quad + (ForkKill(f) - Out(n)) & \text{sponding fork node } f) \\ (ACCKillin(n) + Kill(n)) - Gen(n) & \text{(otherwise)} \end{cases}$$

$$ACCKillin(n) = \begin{cases} \bigcup\limits_{p \,\in\, par\_pred\text{(n)}} ACCKillout(p) + \bigcap\limits_{p \,\in\, seq\_pred\text{(n)}} ACCKillout(p) \\ \quad + (OtherDefs(n) \cap SynchPass(n)) \end{cases}$$

$$ForkKill(n) = \begin{cases} (ACCKillin(n) + Kill(n)) - Gen(n) & (n \text{ is a fork node}) \\ \emptyset & \text{(otherwise)} \end{cases}$$

Figure 10: Dataflow Equations for Programs with Parallel Sections and Event Synchronization

Node	Gen	Kill	ParKill
Entry	$\{x_0, y_0\}$	$\{x_4, x_5, x_8, y_{11}\}$	
1			
2			
3			
4	$\{x_4\}$	$\{x_0, x_5\}$	$\{x_8\}$
5	$\{x_5\}$	$\{x_0, x_4\}$	$\{x_8\}$
6	$\{z_6\}$		$\{z_9\}$
7			
8	$\{x_8\}$	$\{x_0\}$	$\{x_4, x_5\}$
9	$\{z_9\}$		$\{z_6\}$
10			
11	$\{y_{11}\}$	$\{y_0\}$	
12			

Node	In	Out	ACCKillIn	AccKillOut	ForkKill
Entry		$\{x_0, y_0\}$		$\{x_4, x_5, x_8, y_{11}\}$	
1	$\{x_0, y_0\}$	$\{x_0, y_0\}$			
2	$\{x_0, y_0\}$	$\{x_0, y_0\}$			
3	$\{x_0, y_0\}$	$\{x_0, y_0\}$			
4	$\{x_0, y_0\}$	$\{x_4, y_0\}$		$\{x_0, x_5\}$	
5	$\{x_0, y_0\}$	$\{x_5, y_0\}$		$\{x_0, x_4\}$	
6	$\{x_4, x_5, y_0\}$	$\{x_4, x_5, y_0, z_6\}$	$\{x_0\}$	$\{x_0\}$	
7	$\{x_0, y_0\}$	$\{x_0, y_0\}$			
8	$\{x_4, x_5, y_0\}$	$\{x_8, y_0\}$	$\{x_4, x_5\}$	$\{x_0, x_4, x_5\}$	
9	$\{x_0, y_0\}$	$\{x_0, y_0, z_9\}$			
10	$\{x_8, y_0, z_9\}$	$\{x_8, y_0, z_9\}$	$\{x_0, x_4, x_5\}$	$\{x_0, x_4, x_5\}$	
11	$\{x_8, y_0, z_6, z_9\}$	$\{x_8, y_{11}, z_6, z_9\}$	$\{x_0, x_4, x_5, y_0\}$	$\{x_0, x_4, x_5, y_0\}$	
12	$\{x_0, y_0\}$	$\{x_0, y_0\}$			

Figure 11: Data Flow Sets for the program in Figure 4 : Iteration 1.

Node	In	Out	ACCKillIn	AccKillOut	ForkKill
Entry		$\{x_0, y_0\}$		$\{x_4, x_5, x_8, y_{11}\}$	
1	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_4, x_5\}$	$\{x_4, x_5\}$	
2	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_4, x_5\}$		$\{x_4, x_5\}$
3	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$			
4	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_4, y_0, y_{11}, z_6, z_9\}$		$\{x_0, x_5\}$	
5	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_5, y_0, y_{11}, z_6, z_9\}$		$\{x_0, x_4\}$	
6	$\{x_4, x_5, y_0, y_{11}, z_6, z_9\}$	$\{x_4, x_5, y_0, y_{11}, z_6\}$	$\{x_0\}$	$\{x_0\}$	
7	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$			
8	$\{x_4, x_5, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_4, x_5\}$	$\{x_0, x_4, x_5\}$	
9	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_9\}$			
10	$\{x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_4, x_5\}$	$\{x_0, x_4, x_5\}$	
11	$\{x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_8, y_{11}, z_6, z_9\}$	$\{x_0, x_4, x_5, y_0\}$	$\{x_0, x_4, x_5, y_0\}$	
12	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_0, x_8, y_0, y_{11}, z_6, z_9\}$	$\{x_4, x_5\}$	$\{x_4, x_5\}$	

Figure 12: Data Flow Sets for the program in Figure 4 : Iteration 2.

7 Conclusion & Future Work

We have presented data flow equations to compute the reaching definition information at any point in an explicitly parallel program with event synchronization. Data flow equations for computing reaching definitions information in sequential programs have been well understood and used in many current day compilers for the optimization of such programs. We believe that the data flow framework that we have presented in this paper can be used to perform rigorous scalar optimization on parallel programs and thus help achieve better execution rates of such programs on existing high performance architectures. This information will particularly benefit distributed shared memory systems, because optimizations using the dataflow information will reduce the amount of communication between processors.

This work has considered the `Parallel Sections` construct and event synchronization. We have implemented these algorithms in a dataflow tool; we hope to eventually implement them in an actual compiler such as SIGMA. In the future, we propose to extend the data flow equations to handle `Parallel Do`, another parallel construct specified by PCF FORTRAN. We anticipate the use of some technology from data dependence analysis to analyze these constructs.

Earlier work [14, 11, 13, 12] has looked at translating explicitly parallel programs to their Static Single Assignment intermediate representation [5], a more efficient dataflow representation. We hope to extend these results on `post/wait` synchronization to that representation as well.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

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

[3] D. Callahan and J. Subhlok. Static Analysis of low-level synchronization. In *Proc. of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, Madison, WA, May 1988.

[4] David Callahan, Ken Kennedy, and Jaspal Subhlok. Analysis of event synchronization in a parallel programming tool. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* [10], pages 21–30.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* [10], pages 1–10.

[7] Samuel P. Midkiff and David A. Padua. Issues in the optimization of parallel programs. In David Padua, editor, *Proc. 1990 International Conf. on Parallel Processing*, volume II, pages 105–113, St. Charles, IL, August 1990. Penn State Press.

[8] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice Hall, Englewood Cliffs, NJ, 1981.

[9] Parallel Computing Forum. PCF Parallel FORTRAN Extensions. *FORTRAN Forum*, 10(3), September 1991. (special issue).

[10] *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Seattle, Washington, March 1990. ACM Press.

[11] Harini Srinivasan. 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.

[12] Harini Srinivasan and Dirk Grunwald. An Efficient Construction of Parallel Static Single Assignment Form for Structured Parallel Programs. Technical Report CU-CS-564-91, University of Colorado at Boulder., December 1991.

[13] Harini Srinivasan and Michael Wolfe. Analyzing programs with explicit parallelism. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 405–419. Springer-Verlag, 1992.

[14] Michael Wolfe and Harini Srinivasan. Data structures for optimizing programs with explicit parallelism. In H. P. Zima, editor, *Parallel Computation: First International ACPC Conference*, volume 591 of *Lecture Notes in Computer Science*, pages 139–156. Springer-Verlag, Berlin, 1992.