LABORATORY FOR
COMPUTER SCIENCE

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation

Vivek Sarkar

CSG Memo 402
November 14, 1997

To appear in the Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing, held August, 1997 in Minneapolis, Minnesotta.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Analysis and Optimization of Explicitly Parallel Programs using the Parallel Program Graph Representation

Vivek Sarkar

Abstract

As more and more programs are written in explicitly parallel programming languages, it becomes essential to extend the scope of sequential analysis and optimization techniques to explicitly parallel programs. Since the definition of a program dependence graph (PDG) is strongly tied to its underlying sequential program, the PDG is an inadequate intermediate representation for analysis and optimization of explicitly parallel programs. In this paper, we propose the use of Parallel Program Graphs (PPGs) as a general parallel program representation for analysis and optimization of explicitly parallel programs. PPGs are comprised of parallel control flow edges and synchronization edges, and can represent a broad class of deterministic parallel programs. We highlight the main differences between PPGs and PDGs and show how PPGs are strictly more general than PDGs. We also present a solution to the reaching definitions analysis problem for PPGs to illustrate how PPGs can be used to perform analysis and optimization of explicitly parallel programs.

1 Introduction

Major changes in processor architecture over the last decade have created a demand for new compiler optimization technologies. Optimizing compilers have risen to this challenge by steadily increasing the uniprocessor performance gap between optimized compiled and unoptimized compiled code to a level that already exceeds the performance gap between two successive generations of processor hardware. These traditional optimizations [2] have been developed in the context of sequential programs — the assumption of sequential control flow is intrinsic to the definition of basic optimization data structures such as the control flow graph (CFG), and pervades all the optimization algorithms.

As more and more programs are written in explicitly parallel programming languages, it becomes essential to extend the scope of sequential analysis and optimization techniques to explicitly parallel programs. This extension is necessary for maintaining single-processor performance in parallel programs and also for adapting the parallelism in the program to the target parallel machine. By an explicitly parallel programming language, we mean a programming language that contains primitives for creating, terminating, and synchronizing concurrent (logical) threads of activity. The primitives may be in the form of syntactic extensions (*e.g.,* the cobegin-coend construct in Concurrent Pascal [14], or the parallel constructs in the proposed ANSI X3H5 standard [18]), directives (*e.g.,* HPF [15], OpenMP [20]), or library calls (*e.g.,* start(), run(), wait() in Java [13], and similar calls in pthreads [16]). In this paper, we assume that the source primitives in the parallel program have already been translated to the *parallel program graph* (PPG) representation [23, 24] and focus our attention of performing compiler analysis and optimization on PPGs.

The *memory consistency model* assumed for accesses to shared variables can have a profound impact on the semantics of an explicitly parallel program. We distinguish among three (increasingly general) classes of parallel programs:

1. *Deterministic parallel programs* — these parallel programs always produce the same output across multiple runs with the same input. Examples include parallel programs with no data races that are constructed out of deterministic parallel control structures such as doall loops and cobegin-coend, and directed synchronizations such as post-wait.

2. *Nondeterministic data-race-free parallel programs* — these parallel programs may produce different outputs across multiple runs with the same input. Examples include parallel programs from class 1 augmented with undirected synchronization such as acquire-release. However, all accesses to shared data must be protected by control sequencing or by data synchronization (directed or undirected) so that each run of the parallel program is guaranteed to be free of *data races* (programs in class 2 are also referred to as *data-race-free* [1] and as *properly labeled* [11] in the literature).

3. *Nondeterministic parallel programs with data races* — these programs may be nondeterministic and, more importantly, are allowed to contain data races *i.e.,* to contain concurrent data accesses that are not protected by synchronization. The semantics of the data races is usually specified by a strong memory consistency model such as sequential consistency [17].

Compiler analysis and optimization becomes progressively harder for the three classes of parallel programs listed above. The results presented in this paper apply only to deterministic parallel programs (class 1). We intend to extend these results to nondeterministic data-race-free parallel programs (class 2) in future work. Programs in classes 1 and 2 have the property that extra reordering constraints on data accesses only need to be imposed at synchronization boundaries.

However, nondeterministic parallel programs with data races (class 3) pose serious obstacles to compiler analysis and optimization. The strong memory consistency semantics usually assumed for programs in class 3 requires a default compilation approach in which the ordering of *all* shared data accesses in the program is preserved; in general, it is illegal to even reorder two consecutive read accesses with no intervening write in a strong memory consistency model.

There are three approaches that can be taken to deal with class 3 parallel programs. First, the data races in class 3 parallel programs can be viewed as standard programming practice whose semantics is defined by a strong consistency model. The price of this approach is lack of code optimization — by default, accesses to shared data must be treated like accesses to volatile data. Even though there are cases in which the compiler can reorder data accesses for simple classes of programs in class 3 (*e.g.,* see [19]), it is hard to avoid the default serialization of all read/write data accesses in a general case such as separate compilation of procedures that contain pointer-based data references. The second approach can be to view the data races in class 3 parallel programs as access anomalies or errors, similar to the assumptions made by weak hardware memory consistency models (*e.g.,* [1, 11]); this means that strong memory consistency semantics will be guaranteed only for parallel programs in classes 1 or 2. The third approach can be to define a weaker model such as Location Consistency [9, 10] as a uniform memory consistency model for all three classes of parallel programs. The weaker model does not consider data races in class 3 programs to be errors, but instead gives them a different semantics that allows the same flexibility in reordering data accesses as is available for reordering data accesses in class 1 and class 2 parallel programs.

It is unclear at this time which one of these three will become the established approach for dealing with class 3 parallel programs. In the first approach, the analysis and optimization techniques required for class 3 programs will be very different from the PPG-based techniques introduced in this paper. In the second and third approaches, the PPG-based techniques will be directly relevant. We believe that a significant drawback of the first approach is that it will penalize all parallel programs (not just class 3 programs), because the compiler will have to consider the possibility that any procedure that it compiles *might* be called from a class 3 parallel program.

The rest of the paper is organized as follows. Section 2 gives an overview of the PPG representation introduced in [23, 24]. Section 3 contains a few example PPGs. Section 4 highlights the main differences between PPGs and program dependence graphs (PDGs) [7]. Section 5 presents our solution to the reaching definitions analysis problem for PPGs. Section 6 outlines related work, and section 7 contains our conclusions.

2 Parallel Program Graphs

This section contains an overview of the PPG representation introduced in [23, 24]. The *parallel program graph* (PPG) is a general intermediate representation that can represent deterministic parallel programs and that subsumes program dependence graphs (PDGs) [7] and control flow graphs (CFGs) [2]. Analogous to control dependence and data dependence edges in PDGs, PPGs contain *control* edges that represent parallel flow of control and *synchronization* edges that impose ordering constraints on execution instances of PPG nodes. PPGs also contain *MGOTO* nodes [6] that are a generalization of *REGION* nodes in PDGs.

Definition 2.1. A *Parallel Program Graph* $PPG = (N, E_{cont}, E_{sync}, TYPE)$ is a rooted directed multigraph in which every node is reachable from the root using only *control* edges. It consists of:

1. N, a set of nodes.

2. $E_{cont} \subseteq N \times N \times \{T, F, U\}$, a set of labeled *control* edges. Edge $(a, b, L) \in E_{cont}$ identifies a control edge from node a to node b with label L.

   Labels T and F represent true and false outcomes in conditional control edges from a *PREDICATE* node. For consistency, we also use a label, U, for unconditional control edges.

3. $E_{sync} \subseteq N \times N \times \mathit{SynchronizationConditions}$, a set of *synchronization* edges. Edge $(a, b, f) \in E_{sync}$ defines a synchronization from node a to node b with synchronization condition f.

4. *TYPE*, a node type mapping. *TYPE*(n) identifies the type of node n as one of the following: *START*, *PREDICATE*, *COMPUTE*, *MGOTO*.

   The *START* node and *PREDICATE* node types in a PPG are just like the corresponding node types in a CFG or a PDG. The *COMPUTE* node type is more general because a PPG compute node may either have an outgoing control edge as in a CFG or may have no outgoing control edges as in a PDG. A node with *TYPE* = *MGOTO* is used as a construct for creating parallel threads of computation — a new thread is created for each successor of an *MGOTO* node. Only *MGOTO* nodes can have multiple successors with the same label. □

We distinguish between a PPG node and an *execution instance* of a PPG node (i.e., a dynamic instantiation of that node). Given an execution instance I_a of PPG node a, its *execution history*, $H(I_a)$, is defined as the sequence/trace of PPG node and label values that caused execution instance I_a to occur. A PPG's execution begins with a single execution instance (I_{start}) of the *start* node, with $H(I_{start}) = \langle\rangle$ (an empty sequence).

A *control* edge in E_{cont} is a triple of the form (a, b, L), which defines a transfer of control from node a to node b for branch label (or branch condition) L. The semantics of *control* edges is as follows. If *TYPE*(a) = *PREDICATE*, then consider an execution instance I_a of node a that evaluates node a's branch label to be L: if there exists an edge $(a, b, L) \in E_{cont}$, execution instance I_a creates a new execution instance I_b of node b (there can be at most one such successor node) and then terminates itself. The execution history of each I_b is simply $H(I_b) = H(I_a) \circ \langle a, L\rangle$ (where $\circ$ is the sequence concatenation operator). If *TYPE*(a) = *MGOTO*, then let the set of outgoing control edges from node a (all of which must have label = U) be $\{(a, b_1, L), \ldots, (a, b_k, L)\}$. After completion, execution instance I_a creates a new execution instance (I_{b_i}) of each target node b_i and then terminates itself. The execution history of each I_{b_i} is simply $H(I_{b_i}) = H(I_a) \circ \langle a, L\rangle$.

A *synchronization* edge in E_{sync} is a triple of the form (a, b, f), which defines a PPG edge from node a to node b with synchronization condition f. In general, $f(H_1, H_2)$ can be any computable Boolean function on execution histories. However, in practice, there will only be a few limited classes of synchronization conditions of interest (such as the *control-independent* synchronization conditions defined at the end of this section). Given two execution instances I_a and I_b of nodes a and b, $f(H(I_a), H(I_b))$ returns *true* if and only if execution instance I_a must complete execution before execution instance I_b can be started. Note that the synchronization condition depends only on execution histories, and not on any program data values. Also, due to the presence of synchronization edges, it is possible to construct PPGs that may deadlock (as in any explicitly parallel program).

A PPG node represents an arbitrary sequential computation. The control edges of a PPG specify how execution instances of PPG nodes are created (unraveled), and the synchronization edges of a PPG specify how execution instances need to be synchronized. A formal definition of the execution semantics of *MGOTO* edges and *synchronization* edges is given in [23].

We would like to have a definition for synchronization edges that corresponds to the notion of *loop-independent* data dependence edges in sequential programs [3]. If there is a loop-independent data dependence from node a to node x with respect to a common loop L that contains a and x, then for each iteration of loop L that executes *both* nodes a and x it must be the case that the instance of a iterations of loop L that do not execute both a and x. This is the notion we are trying to capture with our definition of *control-independent*, discussed below.

Consider an execution instance I_x of PPG node x with execution history,

$$H(I_x) = \langle u_1, L_1, \ldots, u_i, \ldots, u_j, \ldots, u_k, L_k \rangle \text{ where } u_k = x \ .$$

We define $\mathit{nodeprefix}(H(I_x), a)$ (the *nodeprefix* of execution history $H(I_x)$ with respect to the PPG node a) as follows. If there is no occurrence of node a in $H(I_x)$, then $\mathit{nodeprefix}(H(I_x), a) = \perp$. If $a \neq x$, $\mathit{nodeprefix}(H(I_x), a) = \langle u_1, L_1, \ldots, u_i, L_i \rangle, u_i = a,\ u_j \neq a,\ i < j \leq k$; if $a = x$, then $\mathit{nodeprefix}(H(I_x), a) = H(I_x) = \langle u_1, L_1, \ldots, u_i, \ldots, u_j, \ldots, u_k, L_k \rangle$.

A *control path* in a PPG is a path that consists of only control edges. A node a is a *control ancestor* of node x if there exists an acyclic control path from START to x such that a is on that path.

A synchronization edge (x, y, f) is *control-independent* if a necessary (but not sufficient) condition for $f(H(I_a), H(I_b)) = \mathit{true}$ is that $\mathit{nodeprefix}(H(I_x), a) = \mathit{nodeprefix}(H(I_y), a)$ for all nodes a that are control ancestors of both x and y. The reaching definitions analysis presented in section 5 is currently restricted to PPGs that contain only control-independent synchronization edges.

3 Examples of PPGs

In this section, we give a few examples of PPGs to provide some intuition on how PPGs can be built in a compiler and how they execute at run-time. The reaching definitions analysis in section 5 can work for all three examples described in this section.

Figure 1 is an example of a PPG obtained from a thread-based parallel programming model that uses `create` and `terminate` operations. This PPG contains only control edges; it has no synchronization edges. The `create` operation in statement S2 is modeled by an *MGOTO* node with two successors: node S6, which is the target of the `create` operation, and node S3, which is the continuation of the parent thread. The `terminate` operation in statement S5 ends the current thread, and is modeled by making node S4 a leaf node *i.e.,* a node with no

```
S1:  . . .
S2:  create S6
S3:  . . .
S4:  . . .
S5:  terminate
S6:  . . .
. . . .
```

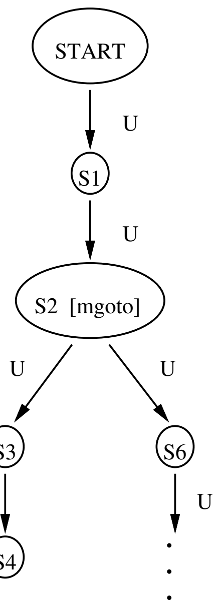


Figure 1: Example of PPG with only control edges

```
S1:  if (   ) mgoto S2, S3
S5:  . . .
. . . .
terminate
S2:  . . .
     goto S4
S3:  . . .
     goto S4
S4: . . .
     terminate
```

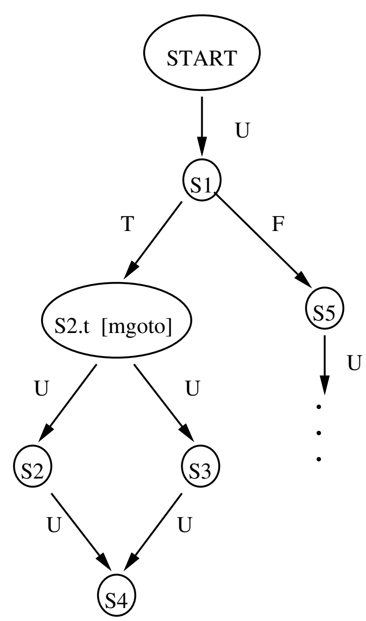


Figure 2: Example of unstructured PPG with only control edges

outgoing control edges. (Any other linguistic construct that identifies the end of a thread's execution can be modeled in the same way.)

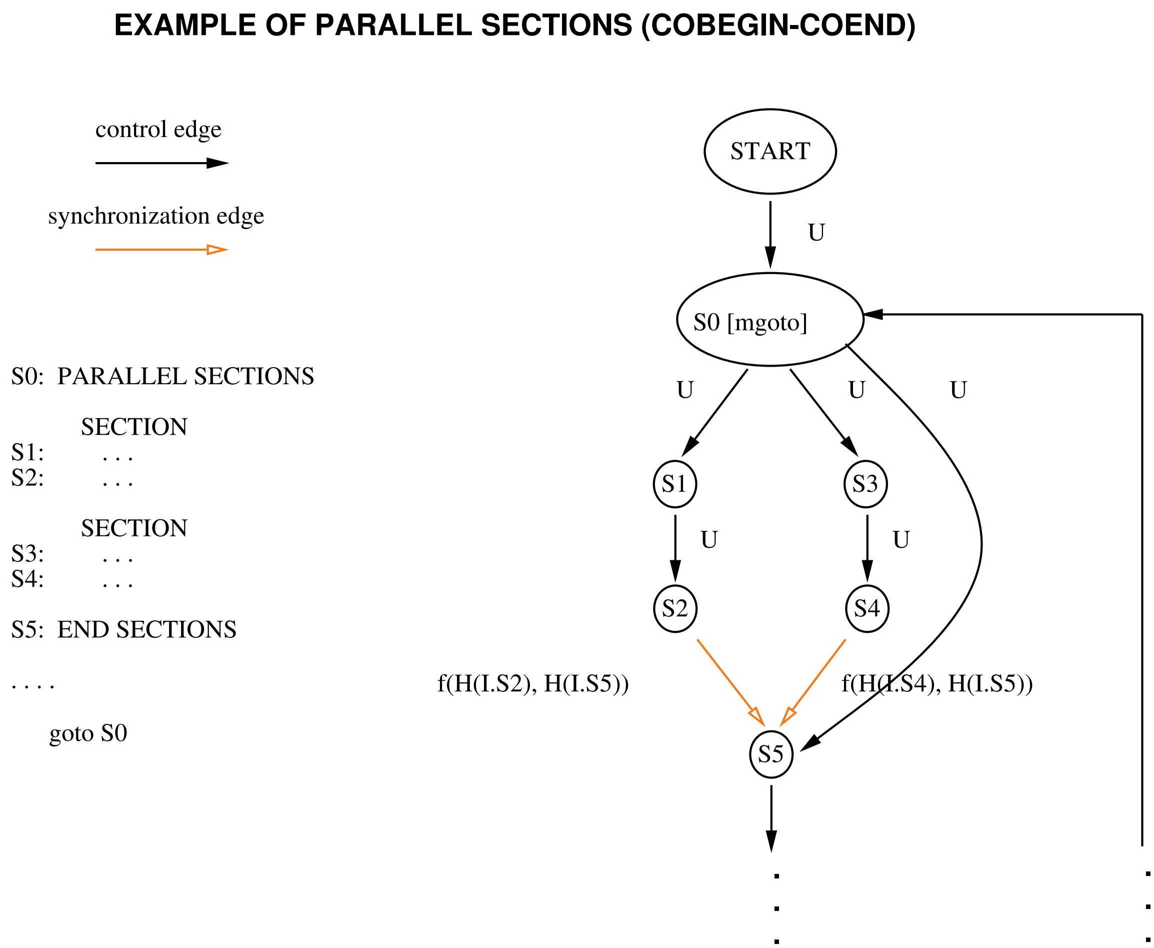
Figure 2 is another example of a PPG that contains only control edges and is obtained from a thread-based parallel programming model. This is an example of "unstructured" parallelism because of the `goto S4` statements which cause node S4 to be executed twice when predicate S1 evaluates to true. The PPG in Figure 2 is also an example of a *non-serializable* PPG [24]. Because node S4 can be executed twice, it is not possible to generate an equivalent sequential CFG for this PPG without duplicating code or inserting Boolean guards. With code duplication, this PPG can be made serializable by splitting node S4 into two copies, one for parent node S2 and one for parent node S3.

Figure 3 is an example of a PPG that represents a `parallel sections` construct [20], which is similar to the `cobegin-coend` construct [14]. This PPG contains both control edges and synchronization edges. The start of a `parallel sections` construct is modeled by an *MGOTO* node with three successors: node S1, which is the start of the first parallel section, node S2, which is the start of the second parallel section, and node S5, which is the continuation of the parent thread. The `goto S0` statement at the bottom of the code fragment is simply modeled by a control edge that branches back to node S0.

The semantics of `parallel sections` requires a synchronization at the `end sections` statement. This is modeled by inserting a synchronization edge from node S2 to node S5 and another synchronization edge from node S4 to node S5. The synchronization condition f for the two synchronization edges is defined in Figure 3 as a simple function on the execution histories of execution instances I.S2, I.S4, I.S5 of nodes S2, S4, S5 respectively. For example, the synchronization condition for the edge from S2 to S5 is defined to be true if and only if $H(I.S2) = concat(H(I.S5), \langle S1, U \rangle)$; this condition identifies the synchronization edge as being control-independent. In practice, a control-independent synchronization can be implemented by simple semaphore operations without needing to examine the execution histories at run-time [12].

Note that node S5 in Figure 3 has to wait for nodes S2 and S4 to complete before it can start execution. For a multithreaded runtime system that supports suspensive threads, a more efficient PPG for this example can be obtained by reducing the three-way branching from the *MGOTO* node by two-way branching *e.g.,* by deleting the control edge from node S0 to node S5, and then replacing the synchronization edge from node S2 to node S5 by a control edge. This transformed PPG would contain the same amount of ideal parallelism as the original PPG but would create two threads instead of three, and would perform one synchronization instead of two, for a given instantiation of the `parallel sections` construct. This idea is an extension of the *control sequencing* transformation that was used in the PTRAN system to replace a data synchronization by sequential control flow [5, 22]. (Note that it is illegal to replace both synchronization edges coming into S5 by control edges, because the resulting PPG would then incorrectly execute node S5 twice for a given instantiation of the `parallel sections` construct.)

Further, if the granularity of work being performed in the parallel sections is too little to justify creation of parallel threads, the PPG in Figure 3 can be transformed into a sequential CFG, which is a PPG with no *MGOTO* nodes and no synchronization edges



f(H(I.S4), H(I.S5)) := H(I.S4) = concat(H(I.S5), <S3,U>)

f(H(I.S2), H(I.S5)) := H(I.S2) = concat(H(I.S5), <S1,U>)

Figure 3: Example of structured PPG with control and synchronization edges

(this transformation is possible because the PPG in Figure 3 is serializable). Therefore, we see that PPG framework can can support a wide range of parallelism from ideal parallelism to useful parallelism to no parallelism. As future work, it would be interesting to extend existing algorithms for selecting useful parallelism from PDGs (*e.g.,* [21]) to operate more generally on PPGs.

4 Comparing PPGs to PDGs

The main distinction between PPGs and PDGs lies in the complete freedom in connecting PPG nodes with control and synchronization edges to represent a wide spectrum of sequential and parallel programs. In contrast, PDGs have several structural constraints arising from the fact that the PDG was designed to be a "maximally parallel" representation of a sequential program. Examples of the structural constraints for PDGs include:

- *No-post-dominator condition:* Let node P be a postdominator of node N in the CFG that the PDG was derived from. Then there cannot be a directed path of control dependence edges from N to P in the PDG.

  (The example PPG in figure 1 and most CFGs violate this condition. A PDG would not allow the presence of an unconditional control edge, like the edge from $S3$ to $S4$ in figure 1.)

- *Predicate-ancestor condition:* if there are two disjoint control dependence paths from (ancestor) node A to node N in a PDG, then A cannot be a *region* node *i.e.,* A must be a predicate node.

  (The example PPG in figure 2 violates this condition, and hence cannot be viewed as a legal PDG.)

Given these structural constraints, it is hard to manipulate the PDG representation as an independent entity. Instead, program transformation is usually performed by maintaining a tight link between a PDG and its corresponding sequential program (CFG).

Figure 4 contains a simple example to illustrate how the CFG and the PDG have to be updated in a consistent manner whenever a program transformation is performed. CFG #1 and PDG #1 contain the original CFG and PDG for this example. As shown in PDG #1, we assume that the only data dependences are $1 \rightarrow 3$, $1 \rightarrow 4$, and $3 \rightarrow 4$ (*e.g.,* assume that node 1 contains a def of variable X, node 3 contains a use and a def of variable X, and node 4 contains a use of variable X). A desirable transformation for this PDG might be to combine nodes 1, 3 and 4 into a single thread since there is no useful parallelism among them. However, it is not possible to replace the $R1 \rightarrow 3$ control dependence edge in PDG #1 by a $1 \rightarrow 3$ control dependence edge, because the $1 \rightarrow 3$ edge violates the *no post-dominator condition* for PDGs. The only transformation that can be made in the PDG towards our goal of serializing nodes 1, 3, 4 is to interchange nodes 1 and 2 as shown in CFG #2 and PDG #2. A separate data structure will then need to be maintained to store the desired

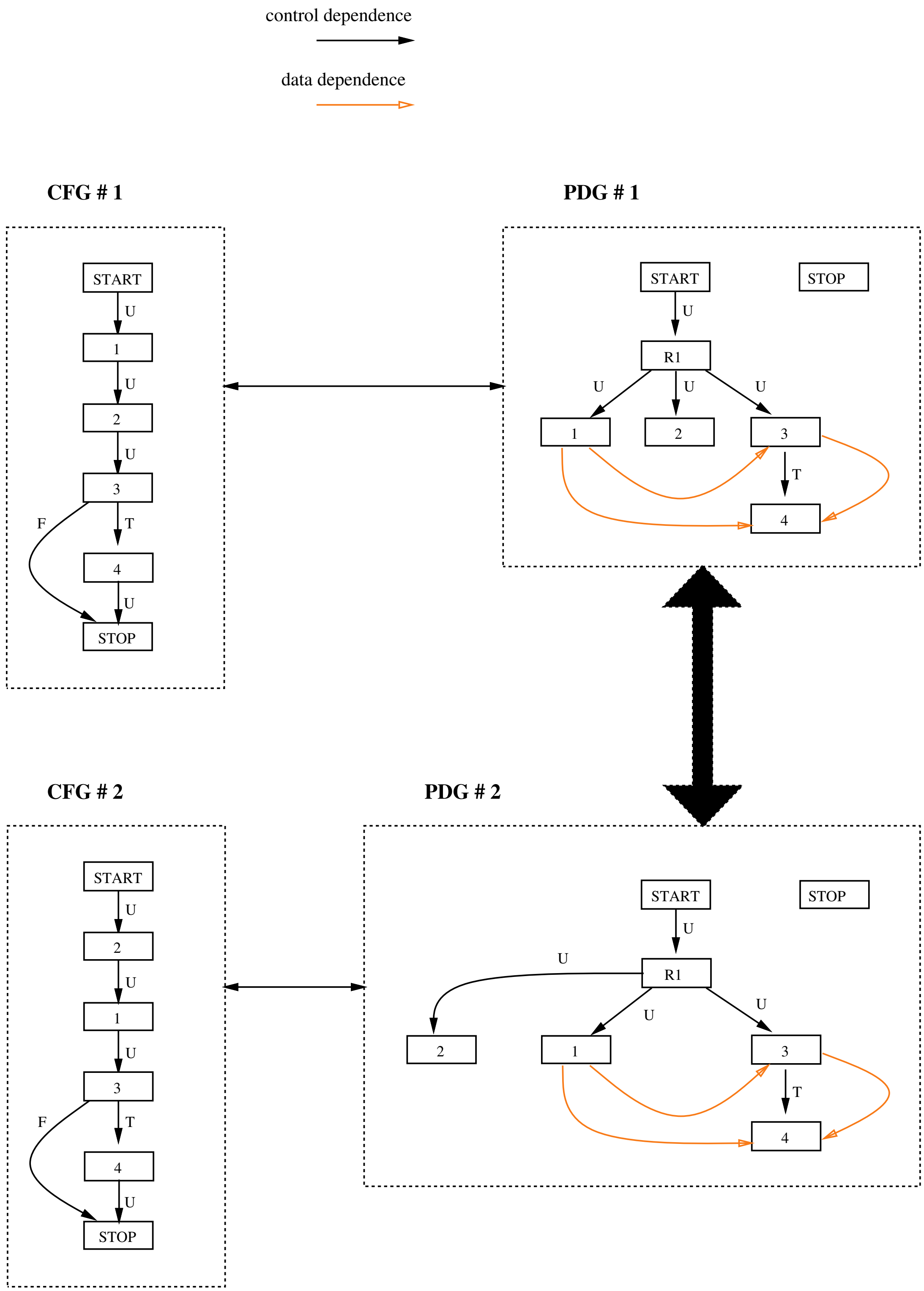


Figure 4: Restrictions on transforming PDGs

thread mapping or task partition as in [21]. Even for this simple code motion transformation, it is necessary to store both CFG #2 and PDG #2 after the transformation.

There are no such structural constraints imposed on PPGs. For example, any CFG can be viewed as a PPG that has no synchronization edges (each CFG edge is treated as a PPG control edge). This PPG is a completely sequential representation of the program. Similarly, any PDG can be viewed as a PPG by treating each PDG control dependence edge as a PPG control edge, each PDG data dependence edge as a PPG synchronization edge, and each PDG region node as a PPG $MGOTO$ node. This PPG is a "maximally parallel" representation of the program. PPGs can also represent a wide spectrum of intermediate parallelism granularities.

Figure 5 illustrates how the desired program transformation for the example in figure 4 can be implemented in the PPG framework. PPG #1 contains the original CFG, which can be viewed as a PPG with no parallelism. PPG #2 contains the PDG derived from the original CFG *i.e.,* PDG #1 from figure 4. This PDG can be viewed as a PPG by treating each control dependence edge as a PPG control edge, each data dependence edge as a PPG synchronization edge, and each region node as a PPG $MGOTO$ node. PPG #3 in figure 5 shows the result of serializing nodes 1, 3, and 4. This was our desired program transformation. Note that PPG #3 contains parallelism (expressed by the $MGOTO$ node), but does not contain any synchronization edges.

We conclude this section with a brief discussion of memory consistency semantics. Since a PDG is derived from a sequential program, its control and data dependences ensure that there are no data races. It is possible to create a PPG that exhibits a data race, however. If read and write accesses to the same location are not properly guarded by control edges or by synchronization edges, then the PPG's execution may exhibit a data race. As discussed in section 1, the results presented in this paper apply only to deterministic parallel programs (class 1). This means that we assume a deterministic and weak memory consistency semantics for PPGs as follows. If a read access is performed in parallel with a write access that changes the location's value, then the result of the read access is assumed to be undefined (*i.e.,* all memory accesses are assumed to be non-atomic). Similarly, the result of two parallel write accesses with different values is also undefined. Note that non-atomicity implies that memory accesses alone cannot be used for synchronization; the control edges and synchronization edges are the only mechanisms available in the PPG for coordinating execution instances of PPG nodes. More details on this PPG memory consistency model can be found in [23]. We intend to extend the results of this paper to nondeterministic data-race-free parallel programs (class 2) in future work.

5 Reaching Definitions Analysis on PPGs

Section 5.1 reviews reaching definitions analysis for sequential programs *i.e.,* for CFGs. Section 5.2 presents our solution for reaching definitions analysis for PPGs. Section 5.3 illustrates our solution with an example.

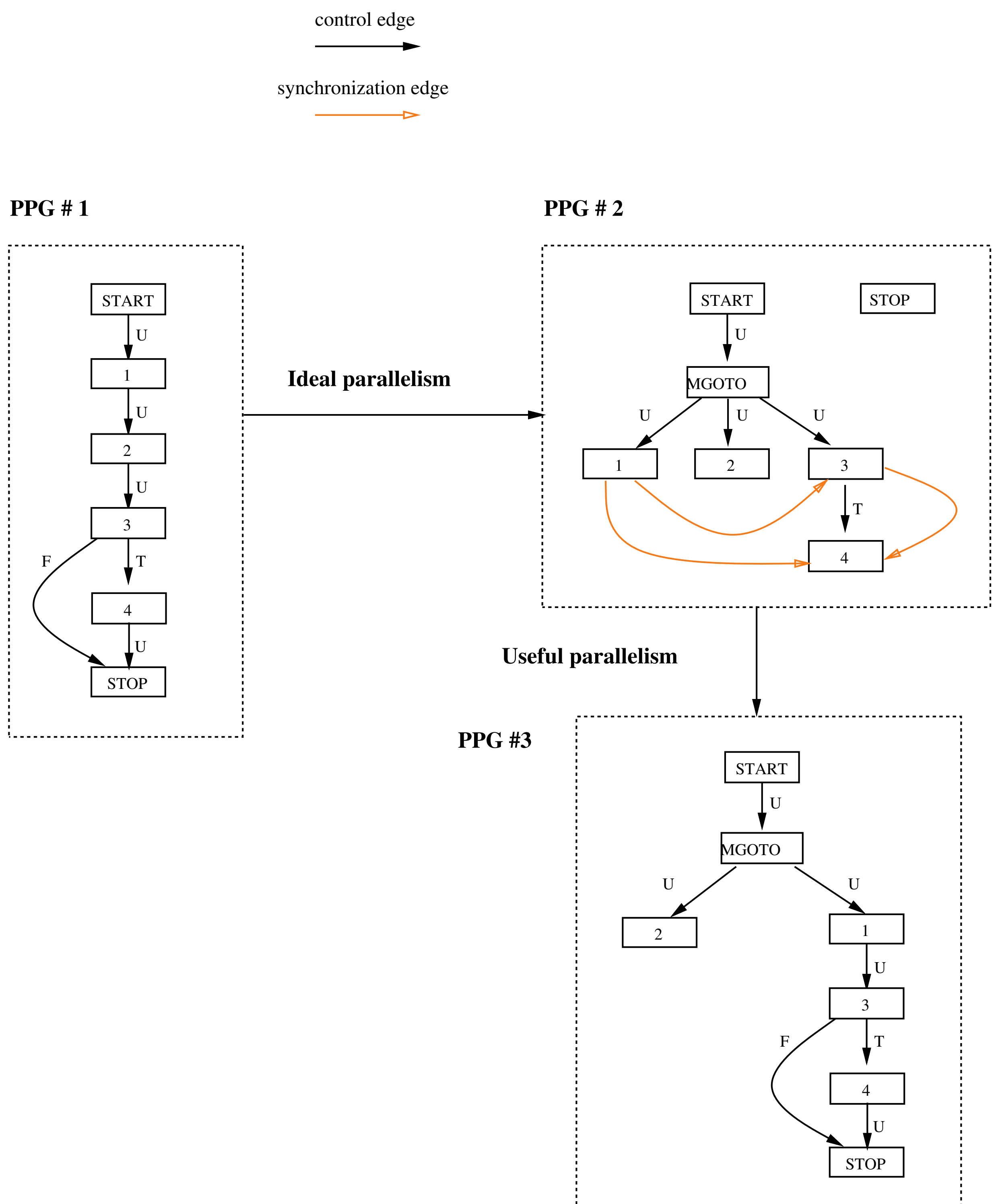


Figure 5: Example of transforming PPGs

5.1 Reaching Definitions Analysis on CFGs

Recall that a definition d of variable V *reaches* a program point P in a CFG if there is a directed path in the CFG from d to point P such that d is not "killed" along that path [2] *i.e.,* there is no other definition of variable V along that path. Following the usual convention, let $\text{REACH}_{in}(n)$ and $\text{REACH}_{out}(n)$ denote the set of definitions that reach the start and end respectively of CFG node (basic block) n. Also, let $Kill(n)$ denote the set of definitions d that get *killed* in node n *i.e.,* for which there exists another definition d' in node n to the same variable as d (if d also appears in n then d' must follow d). And let $Gen(n)$ denote the set of definitions d that get *generated* in node n *i.e.,* d appears in n and is not killed in n.

For each CFG node n, $Kill(n)$ and $Gen(n)$ can be computed by a local examination of the statements/instructions in basic block n. The following equations can then be used to relate the $\text{REACH}_{in}(n)$ and $\text{REACH}_{out}(n)$ sets [2]:

$$
\begin{aligned}
\text{REACH}_{out}(n) &= (\text{REACH}_{in}(n) - Kill(n)) \bigcup Gen(n) \\
\text{REACH}_{in}(n) &= \bigcup_{p \,\in\, \text{pred}(n)} \text{REACH}_{out}(p)
\end{aligned}
$$

These equations form the basis for the classical iterative algorithm for reaching definitions in which the $\text{REACH}_{in}(n)$ sets are initialized to $\emptyset$, and the $\text{REACH}_{out}(n)$ and $\text{REACH}_{in}(n)$ are iteratively recomputed till no change occurs.

In preparation for our solution to the reaching definitions analysis problem for PPGs, we also formalize the concept of *redefinition*. We say that a definition d is *redefined* at program point P if there is a path from d to P and d is killed along *all paths* from d to P. The sets $\text{REDEF}_{in}(n)$ and $\text{REDEF}_{out}(n)$ denote the set of definitions that are redefined at the start and end respectively of CFG node (basic block) n.

We observe that if definition d is redefined at point P then it cannot reach point P, and vice versa. Therefore, the REACH and REDEF sets at any program point must be disjoint. Further, if a definition d belongs to neither REACH_P nor REDEF_P for some program point P, then it must be the case that there is no CFG path from d to P. To prepare for reaching definitions analysis on PPGs, we provide the following equations which can be used to compute the REDEF sets using the REACH_{in} set:

$$
\begin{aligned}
\text{REDEF}_{out}(n) &= (\text{REDEF}_{in}(n) - Gen(n)) \bigcup \\
&\quad\ (Kill(n) \cap \text{REACH}_{in}(n)) \\
\text{REDEF}_{in}(n) &= \bigcap_{p \,\in\, \text{pred}(n)} \text{REDEF}_{out}(p)
\end{aligned}
$$

5.2 Reaching Definitions Analysis on PPGs

We now extend the equations for the REACH_{in} and REACH_{out} sets in PPGs as follows. As mentioned earlier, this analysis is currently restricted to PPGs that contain only control-independent synchronization edges:

$$
\begin{aligned}
\mathrm{REACH}_{out}(n) &= (\mathrm{REACH}_{in}(n) - \mathit{Kill}(n)) \bigcup \mathit{Gen}(n) \\
\mathrm{REACH}_{in}(n) &= \left(\bigcup_{p \,\in\, \mathrm{pred}(n)} \mathrm{REACH}_{out}(p) \right) - \mathrm{REDEF}_{in}(n)
\end{aligned}
$$

The first equation for $\mathrm{REACH}_{out}(n)$ is identical to the sequential case. For the second equation, the set $\mathrm{pred}(n)$ refers to all PPG predecessors of node n (control edge predecessors and synchronization edge predecessors). As we will see in the example program in section 5.3, it is important to subtract out $\mathrm{REDEF}_{in}(n)$ to ensure that the $\mathrm{REACH}_{in}(n)$ set does not conservatively (imprecisely) include extra definitions. This subtraction was not necessary in the equation for the sequential case in section 5.1 because $\mathrm{REDEF}_{in}(n)$ and $\cup_{p \,\in\, \mathrm{pred}(n)} \mathrm{REACH}_{out}(p)$ are completely disjoint for a (sequential) CFG thus making the subtraction a no-op in the sequential case.

The equations for the REDEF sets for a PPG are now as follows:

$$
\begin{aligned}
\mathrm{REDEF}_{out}(n) &= (\mathrm{REDEF}_{in}(n) - \mathit{Gen}(n)) \bigcup \\
&\quad\; (\mathit{Kill}(n) \cap \mathrm{REACH}_{in}(n)) \\
\mathrm{REDEF}_{in}(n) &= \bigcup_{p \,\in\, \mathrm{sync\_pred}(n)} \mathrm{REDEF}_{out}(p) \quad \bigcup \\
&\quad\; \bigcap_{p \,\in\, \mathrm{control\_pred}(n)} \mathrm{REDEF}_{out}(p)
\end{aligned}
$$

The first equation is identical to the sequential case. Also, as in the sequential case, the second equation includes the intersection of the REDEF_{out} sets for node n's control edge predecessors in $\mathrm{REDEF}_{in}(n)$. The main new addition in the second equation lies in including the *union* of the REDEF_{out} sets for node n's *synchronization* edge predecessors in $\mathrm{REDEF}_{in}(n)$. The union operation is used because an instance of node n must wait for *all* the appropriate instances of its synchronization edge predecessors to complete execution.

As a final note, we observe that even though the four equations shown above appear to be mutually recursive, we can avoid recursion at any given program point by evaluating the sets in the following order,

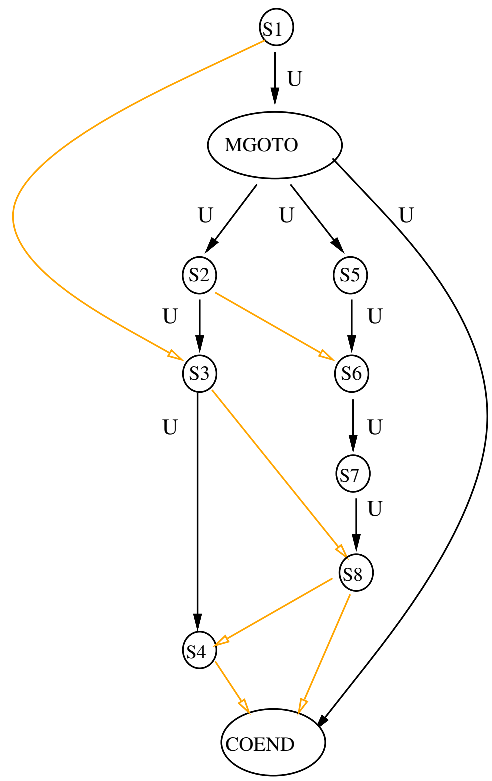
$$
\mathrm{REDEF}_{in}(n), \mathrm{REACH}_{in}(n), \mathrm{REDEF}_{out}(n), \mathrm{REACH}_{out}(n).
$$

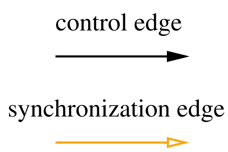
5.3 Example

In this section, we use an example PPG to illustrate the reaching definitions analysis equations from section 5.1. The example program and its PPG is shown in figure 6. The example program was taken from [27] (page 79) where it was included as an example illustrating the difficulty of doing data flow analysis for a program with explicit synchronization.

The notation X_1, X_2, X_4, X_7 is used to identify distinct definitions of the same scalar variable, X. In addition to the implicit synchronization in the cobegin-coend construct, this example

```
S1:   X1 := ...
      post(ev1)
      cobegin
S2:          X2 := ...
             post(ev2)
S3:          wait(ev1)
             ...
             post(ev3)
S4:          wait(ev8)
             X4 := ...
      \\
S5:          ...
S6:          wait(ev2)
             ...
S7:          X7 := ...
S8:          wait(ev3)
             ...
             post(ev8)
      coend
```





Node n	$\text{REDEF}_{in}(n)$	$\text{REACH}_{in}(n)$	$\text{REDEF}_{out}(n)$	$\text{REACH}_{out}(n)$
S1	$\emptyset$	$\emptyset$	$\emptyset$	$\{X_1\}$
cobegin	$\emptyset$	$\{X_1\}$	$\emptyset$	$\{X_1\}$
S2	$\emptyset$	$\{X_1\}$	$\{X_1\}$	$\{X_2\}$
S3	$\{X_1\}$	$\{X_2\}$	$\{X_1\}$	$\{X_2\}$
S5	$\emptyset$	$\{X_1\}$	$\emptyset$	$\{X_1\}$
S6	$\{X_1\}$	$\{X_2\}$	$\{X_1\}$	$\{X_2\}$
S7	$\{X_1\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{X_7\}$
S8	$\{X_1, X_2\}$	$\{X_2\}$	$\{X_1, X_2\}$	$\{X_7\}$
S4	$\{X_1, X_2\}$	$\{X_7\}$	$\{X_1, X_2, X_7\}$	$\{X_4\}$
coend	$\{X_1, X_2, X_7\}$	$\{X_4\}$	$\{X_1, X_2, X_7\}$	$\{X_4\}$

Figure 6: Example parallel program with explicit synchronization and the reaching definitions analysis sets for its PPG

has explicit synchronization using the `post` and `wait` operations on event variables `ev1`, `ev2`, `ev3` and `ev8`.

Figure 6 also includes the solutions for the REDEF and REACH sets for all the PPG nodes that would be obtained by an iterative data flow analysis program using the equations from section 5.2. Notice that $\text{REACH}_{in}(S3)$ has been computed as $\{X_2\}$, which is a precise solution. In contrast, the analysis technique from [27] would conservatively include X_1 in $\text{REACH}_{in}(S3)$ due to the presence of the synchronization edge from $S1$ to $S3$ (even though the synchronization edge is redundant). Note that our analysis also correctly identifies X_4 as the only definition that reaches the `coend` statement.

6 Related Work

The work that is most closely related to ours is by Ferrante, Grunwald, and Srinivasan on compile-time analysis and optimization of explicitly parallel programs [8] which builds on earlier work by Srinivasan on optimizing explicitly parallel programs [27]. They too provide data flow equations for the reaching definitions analysis problem. However, the program representation assumed in their work is a parallel flow graph (PFG) which is more restrictive than a PPG. Any PFG can be converted to a PPG, but the converse is not true. Some specific differences between PFGs and PPGs are as follows. First, the PFG assumes a copy-in/copy-out semantics for accessing shared variables in parallel constructs, whereas the PPG allows direct access of shared variables everywhere. Second, the PFG assumes that the parallelism is well-structured (a nesting of cobegin–coend and doall statements) and that the structure is visible in the representation (their analysis algorithms use the lexical nesting structure of parallel constructs); in contrast, PPGs can represent both structured and unstructured parallelism. (This difference is analogous to special-case optimization of structured sequential programs using an abstract syntax tree representation vs. general optimization of sequential programs using a CFG representation). Third, a PFG distinguishes between parallel control flow and sequential control flow edges, whereas they are treated uniformly in a PPG (explicit synchronizations and synchronizations associated with parallel control flow are also treated uniformly in a PPG). Finally, as discussed in section 5.3, the analysis algorithms in [8, 27] become imprecise in the presence of synchronization edges that cross lexical levels such as the edge from $S1$ to $S3$ in figure 6.

There has been some initial work done on analyzing explicitly parallel programs with a sequentially consistent memory model *e.g.,* [25, 19, 4]. This is a much harder problem because it is necessary to analyze all possible interleavings of memory operations to understand the reordering constraints for a program under a strong memory consistency model. In contrast, our analysis techniques are developed for the PPG in which all reordering constraints for the parallel program are captured by the control and synchronization edges (just as a PDG captures all the reordering constraints for a sequential program).

The idea of extending PDGs to PPGs was introduced by Ferrante and Mace in [6], extended by Simons, Alpern, and Ferrante in [26], and further extended by our past work in [23] and [24]. The definition of PPGs used in this paper is similar to the definition from [24];

the only difference is that the definition in [23] used *MGOTO edges* instead of *MGOTO nodes*. The PPG definition used in [6, 26] imposed several restrictions: a) the PPGs could not contain loop-carried data dependences (synchronizations), b) only reducible loops were considered, c) PPGs had to satisfy the no-post-dominator rule, and d) PPGs had to satisfy the predicate-ancestor rule. The PPGs defined in this paper have none of these restrictions. Restrictions a) and b) limit their PPGs from being able to represent all PDGs that can be derived from sequential programs; restriction a) is a serious limitation in practice, given the important role played by loop-carried data dependences in representing loop parallelism [28]. Restriction c) limits their PPGs from being able to represent CFGs. Restriction d) limits their PPGs from representing thread-based parallel programs in their full generality *e.g.,* the PPG from Figure 2 cannot be expressed in their PPG model because it is possible for node S4 to be executed twice. The Hierarchical Task Graph (HTG) proposed by Girkar and Polychronopoulos [12] is another variant of the PPG, applicable only to structured programs that can be represented by the hierarchy defined in the HTG.

7 Conclusions and Future Work

In this paper, we motivated the use of the PPG representation in analysis and optimization of explicitly parallel programs. We presented a solution for reaching definitions analysis on PPGs and showed how it is more precise and more generally applicable than the solution presented in [8, 27].

For future work, we intend to extend the results of this paper to nondeterministic data-race-free parallel programs (class 2 programs) that support mutual exclusion through acquire-release synchronization. Other possibilities for future work include extending sequential compiler analysis and optimizations algorithms (*e.g.,* common subexpression elimination, SSA construction, etc.) for use on PPGs, and extending existing algorithms for selecting useful parallelism from PDGs to operate more generally on PPGs.

Acknowledgments

The author would like to thank Professor Arvind and other members of the Computation Structures Group at the M.I.T. Laboratory for Computer Science for providing a stimulating research environment for this work, and Harini Srinivasan for her comments on this paper.

References

[1] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Transaction on Parallel and Distributed Systems*, pages 613–624, August 1993.

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

[3] John R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, Houston, TX, 1983.

[4] Jyh-Herng Chow and William Ludwell Harrison. Compile time analysis of parallel programs that share memory. *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, January 1992.

[5] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic Generation of DAG Parallelism. *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Portland, Oregon*, 24(7):54–68, June 1989.

[6] J. Ferrante and M. E. Mace. On Linearizing Parallel Code. *Conf. Rec. Twelfth ACM Symp. on Principles of Programming Languages*, pages 179–189, January 1985.

[7] J. Ferrante, K. Ottenstein, and J. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] Jeanne Ferrante, Dirk Grunwald, and Harini Srinivasan. Compile-time Analysis and Optimization of Explicitly Parallel Programs. *Journal of Parallel Algorithms and Applications*, 1997. (To appear).

[9] Guang R. Gao and Vivek Sarkar. Location Consistency: Stepping Beyond the Memory Coherence Barrier. *International Conference on Parallel Processing*, August 1995.

[10] Guang R. Gao and Vivek Sarkar. On the Importance of an End-To-End View of Memory Consistency in Future Computer Systems. *Proceedings of the 1997 International Symposium on High Performance Computing, Fukuoka, Japan*, November 1997.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th ACM International Symposium on Computer Architecture*, pages 15–27, May 1990.

[12] Miland Girkar and Constantine Polychronopoulos. The HTG: An Intermediate Representation for Programs Based on Control and Data Dependences. Technical report, Center for Supercomputing Res. and Dev.-University of Illinois, May 1991. CSRD Rpt. No.1046.

[13] James Gosling and Henry McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems Computer Company, 2550 Garcia Avenue Mountain View, CA, May 1995.

[14] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software engineering*, SE-1(2):199–206, June 1975.

[15] High Performance Fortran Forum. *High Performance Fortran. Language Specification – Version 0.4*, December 1992.

[16] IEEE. *Thread Extensions for Portable Operating Systems (Draft 6)*, February 1992. P1003.4a/6.

[17] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[18] B. Leasure. Parallel Processing Model for High Level Programming Languages. April 1994. Draft proposal by X3H5 committee of the American National Standard for Information Processing Systems.

[19] Jaejin Lee, Samuel P. Midkiff, and David A. Padua. Concurrent Static Single Assignment Form and Constant Propagation for Explicitly Parallel Programs. In *Languages and compilers for parallel computing. Proceedings of the 10th international workshop. Held Aug., 1997 in Minneapolis, MN.*, Lecture Notes in Computer Science. Springer-Verlag, New York, 1998. (to appear).

[20] OpenMP: A Proposed Industry Standard API for Shared Memory Programming, October 1997. White paper on OpenMP initiative, available at http://www.openmp.org/openmp/mp-documents/paper/paper.ps.

[21] Vivek Sarkar. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development*, 35(5/6), 1991.

[22] Vivek Sarkar. The PTRAN Parallel Programming System. In B. Szymanski, editor, *Parallel Functional Programming Languages and Compilers*, ACM Press Frontier Series, pages 309–391. ACM Press, New York, 1991.

[23] Vivek Sarkar. A Concurrent Execution Semantics for Parallel Program Graphs and Program Dependence Graphs (Extended Abstract). *Springer-Verlag Lecture Notes in Computer Science*, 757:16–30, 1992. Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing, Yale University, August 1992.

[24] Vivek Sarkar and Barbara Simons. Parallel Program Graphs and their Classification. *Springer-Verlag Lecture Notes in Computer Science*, 768:633–655, 1993. Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, Oregon, August 1993.

[25] Dennis Shasha and Marc Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. Technical report, IBM, 1987. Report RC12936.

[26] Barbara Simons, David Alpern, and Jeanne Ferrante. A Foundation for Sequentializing Parallel Code. *Proceedings of the ACM 1990 Symposium on Parallel Algorithms and Architecture*, July 1990.

[27] Harini Srinivasan. *Optimizing Explicitly Parallel Programs.* PhD thesis, Department of Computer Science, University of Colorado, 1994.

[28] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing.