Abstract

In this paper, we present a concurrent execution semantics for Parallel Program Graphs (PPGs), a general parallel program representation that includes Program Dependence Graphs (PDGs) and sequential programs. We believe that this semantics is natural to the programmer's way of thinking, and that it also provides a suitable execution model for efficient implementation on real architectures. To demonstrate the robustness of our semantics, we prove a Reordering Theorem which states that a PPG's semantics does not depend on the order in which parallel nodes are executed, and an Equivalence Theorem which states that the semantics of a sequential program is identical to the semantics of its PDG.

1 Introduction

The *Program Dependence Graph* (PDG) [10] is a popular representation of control and data dependences derived from a sequential program. PDGs have been shown to be useful for solving a variety of problems, including optimization [10], vectorization [5], code generation for VLIW machines [12, 13], merging versions of programs [14], and automatic detection and management of parallelism [2, 3, 8]. A PDG node represents an arbitrary sequential computation e.g. a basic block, a statement or an operation. An edge in a PDG represents a *control dependence* or a *data dependence*. PDGs reveal the inherent parallelism in a program written in a sequential programming language by removing artificial sequencing constraints from the program text.

In this paper, we introduce the *Parallel Program Graph* (PPG) as a more general intermediate representation of parallel programs than PDGs. PPGs contain *mgoto* edges that represent parallel flow of control, and *synchronization* edges that impose ordering constraints on execution instances of PPG nodes. As they execute, the PPG nodes perform read and write accesses to a shared memory. If read and write accesses to the same location are not properly guarded by mgoto edges or by synchronization edges, then the PPG's execution may incur an access anomaly. 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 undefined (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 cannot be used for synchronization; the mgoto edges and synchronization edges are the only mechanisms available in the PPG for coordinating execution instances of PPG nodes.

Given the widespread use of the PDG, there is a strong motivation to understand its parallel execution semantics. The semantics of PDGs has been examined in past work by Selke and by Cartwright and Felleisen. Both those approaches assumed a restricted programming language (the language W), and presented a value-oriented semantics for PDGs. The value-oriented nature of the semantics makes it inconvenient to model store-oriented operations such as an update of an array element or of a pointer-dereferenced location. Also, the control structures in the language W are restricted to `if-then-else` and `while-do`; the semantics did not cover programs with arbitrary control flow. However, PDGs can be used to represent arbitrary (unstructured) control flow and arbitrary data read and write accesses, as found in imperative programming languages like Fortran, C and Pascal. This generality leads to many subtleties and potential ambiguities in interpreting a PDG's parallel execution.

In this paper, we resolve these subtleties and ambiguities by defining a parallel imperative semantics for PPGs (and hence for PDGs) in their full generality. To demonstrate the robustness of our semantics, we prove a Reordering Theorem which states that a PPG's semantics does not depend on the order in which parallel nodes are executed, and an Equivalence Theorem which states that the semantics of a sequential program is identical to the semantics of its PDG. Our semantics accounts for the possibility of program exceptions like race conditions and deadlock, and that the theorems also take these possibilities into account.

We believe that the semantics presented in this paper is natural to the programmer's way of thinking, and lends itself to efficient concurrent execution models for PPGs. The semantics is defined with respect to a global *scheduling system.* The scheduling system allows a programmer (or a debugging system) to follow the progress of a PPG's execution in a step-by-step manner, if so desired. Parallel nodes may be executed in any order—the Reordering Theorem guarantees that all node orderings yield the same semantics. The global scheduling system is defined as a centralized scheduling algorithm, so as to provide a simple and convenient mental abstraction for the programmer; a distributed scheduler would probably be the preferred way to actually implement such a scheduling system on a parallel architecture. Our semantics is *imperative* — the execution of a PPG node is defined by read and write accesses into a shared global memory. This is in contrast to the value-oriented semantics proposed for PDGs in [18], where a PDG node is viewed as a function that consumes input values from its input data dependence edges and produces output values on its output data dependence edges. It is well known that a value-oriented model can be inefficient to implement in practice, because the extra copying overhead required for store-oriented operations (like the update of an array element) can be prohibitive. It is also awkward to force the programmer to think of store-oriented operations in a value-oriented way. Our semantics for PPGs is *strict* — the entire program execution is assumed to return $\perp$ (error) if any statement in the program returns $\perp$. This is in contrast to the non-strict semantics proposed for PDGs in [7] (similar to the non-strict semantics of functional languages). We believe that a strict semantics is more natural for PPGs because the source programming languages typically used for PDGs (e.g. Fortran, Pascal, C) also have a strict semantics. Further, a strict semantics is more efficient to implement than a non-strict semantics because it does not require support for demand-driven evaluation. In summary, we believe that our parallel, imperative and strict semantics is more appropriate for PPGs and PDGs than other semantics that have been proposed [18, 7] because it is a logical extension of the semantics of the PDG's source languages, because it is applicable to general PDGs as they are used in practice, and because it can be implemented more efficiently.

The rest of the paper is organized as follows. Section 2 defines the *PPG representation* as well as the *virtual parallel architecture* that serves as the target execution model for PPGs. Section 3 defines the global *scheduling system* for PPGs. Section 4 presents the *concurrent execution semantics* for PPGs, and also proves the *Reordering Theorem*. Section 5 relates PDGs to PPGs, and also proves the *Equivalence Theorem*. Section 6 discusses related work, and Section 7 contains the conclusions of this paper and an outline of possible directions for future work.

2 Definition of the Parallel Program Graph (PPG)

Definition 2.1 A *Parallel Program Graph* (PPG) $G = (N, E_{mgoto}, E_{sync}, start)$ consists of a set of nodes N, a set of *mgoto* ("multiple goto") edges E_{mgoto}, a set of *synchronization* edges E_{sync}, and a designated start node $start \in N$.

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 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 $H(I_{start}) = <>$ (an empty sequence).

An *mgoto* edge in E_{mgoto} is a triple of the form (a, b_i, L), which defines a PPG edge from node a to node b_i with branch label (or branch condition) L. In general, there may be multiple outgoing *mgoto* edges from node a with branch label L, $\{(a, b_1, L), \ldots, (a, b_k, L)\}$. The semantics of *mgoto* edges is as follows. Consider an execution instance I_a of node a that evaluates node a's branch label to be 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 I_{b_i} is simply $H(I_{b_i}) = H(I_a) \circ < a, L >$ (where $\circ$ is the sequence concatenation operator).

A *synchronization* edge in E_{sync} is a triple of the form (a, c, f), which defines a PPG edge from node a to node c with synchronization condition f. $f(H_1, H_2)$ is a Boolean function on execution histories. Given two execution instances I_a and I_c of nodes a and c, $f(H(I_a), H(I_c))$ returns *true* if and only if execution instance I_a must complete execution before execution instance I_c can be started. Note that the synchronization condition depends only on execution histories, and not on any program data values. $\square$

A PPG node represents an arbitrary sequential computation. The mgoto edges specify how execution instances of PPG nodes are created (unravelled), and the synchronization edges specify how execution instances need to be synchronized. A formal definition of the execution semantics of *mgoto* edges and *synchronization* edges is presented later in Sections 3 and 4.

The parallel computation model assumed in this paper can be viewed as a *virtual parallel architecture*. This abstract architecture consists of an unbounded number of asynchronous virtual processors that communicate through a shared global memory. In our model, all read and write accesses to the shared memory (i.e. all inter-processor communications) are assumed to be non-atomic transactions. Non-atomicity is less restrictive than atomicity, and makes it easier and more efficient to implement the virtual parallel architecture's memory access model on a real architecture. The non-atomicity assumption has an important effect on the semantics of parallel reads and writes to the same location. If two write accesses with different values are issued in parallel to the same memory location, the resulting value is undefined in our model[1]. We say that both write accesses incur a *write-write hazard*. In other shared-memory models, it is usually assumed that the resulting value must be one of the two values being written. If a read access to a memory location is issued in parallel with a write access that changes the value in the location, the result of the read operation is undefined in our model. We say that the read access incurs a *read-write hazard*. The semantics of write-write and read-write hazards are formally specified in Section 4 (Definition 4.1).

A desirable consequence of the non-atomicity assumption is that there is a logical separation between inter-processor synchronization and inter-processor communication in the architecture. Inter-processor synchronization is specified by the PPG's mgoto edges and synchronization edges. Inter-processor communication is specified by the read/write shared-memory accesses in the PPG's nodes. Due to non-atomicity, it is not possible to perform any synchronization by using shared-memory accesses.

[1]Note that the result of two parallel write accesses with the *same* value is well-defined.

3 Scheduling Model for PPGs

In this section, we present a scheduling system for PPGs that *unravels* a PPG into a *partial order*, $\prec$, defined on execution instances of PPG nodes, based on the semantics of mgoto edges and synchronization edges in the PPG. The partial order is a conceptual tool used to define the scheduling system. We do not suggest that the partial order be actually constructed as a data structure when scheduling execution instances of PPG nodes on real architectures; instead, more efficient scheduling/synchronization mechanisms should be used as appropriate for the target architecture to enforce the constraints imposed by the partial order (e.g. fork/join operations, counting semaphores, guards, etc.).

Figure 1 outlines a scheduling algorithm for scheduling PPG nodes on the virtual parallel architecture described in Section 2. The initial contents of memory locations are defined by an input store, σ_i. PPG execution begins with a single execution instance of the *start* node. Each iteration of the while-loop in Figure 1 schedules an execution instance that has been created and whose predecessors have previously been scheduled. Step 1 schedules the execution instance on a virtual processor and evaluates the branch label obtained on completion. Step 2 creates new execution instances for successor nodes, as defined by the semantics of mgoto edges (Definition 2.1).

Step 2 also specifies how the partial order should be updated after creating a new execution instance, I_{b_i}. Steps 2.b and 2.c respectively update $\prec$ for all "successor" and "predecessor" execution instances of I_{b_i}. To identify successor execution instances of I_{b_i}, step 2.b enumerates each acyclic sequence of mgoto edges $S =< (x_1, x_2, L_1), \ldots, (x_k, x_{k+1}, L_k) >$ that starts with PPG node $x_1 = b_i$, examines each synchronization edge of the form (x_{k+1}, c, f), and checks if there is an execution instance I_c such that $f(H(I_{b_i}) \circ P, H(I_c)) = \mathit{true}$ (where $P =< x_1, L_1, \ldots, x_k, L_k >$ is the "history suffix" defined by S). To identify predecessor execution instances of I_{b_i}, step 2.c examines each synchronization edge of the form (d, b_i, f), then enumerates each acyclic sequence of mgoto edges $S =< (x_1, x_2, L_1), \ldots, (x_k, x_{k+1}, L_k) >$ that ends with PPG node $x_{k+1} = d$, and checks if there is an execution instance I_c such that $f(H(I_c) \circ P, H(I_{b_i})) = \mathit{true}$ (where $c = x_1$, and $P =< x_1, L_1, \ldots, x_k, L_k >$ is the "history suffix" defined by S).

Since a partial order is a reflexive, transitive, antisymmetric binary relation (by definition), all updates to $\prec$ in Figure 1 take care to preserve these properties.

4 A Concurrent Execution Semantics for PPGs

In this section, we present a concurrent execution semantics for PPGs. The concurrent execution semantics for PPGs is based on the scheduling system defined in Section 3. The operational semantics presented in this paper can be used to determine if two PPGs have the same semantics, and if a given PPG's execution leads to unsafe situations like read-write and write-write hazards, non-termination, or deadlock.

```
Create $I_{start}$, an execution instance of the start node, and initialize $\prec = \{(I_{start}, I_{start})\}$,
and $H(I_{start}) = <>$ (an empty sequence)

while $\exists$ an execution instance $I_a$ that has been created but not scheduled such that all of
$I_a$'s predecessors in $\prec$ (except $I_a$ itself) have been scheduled do

  1. Schedule execution instance $I_a$
     (let $L$ = branch label for PPG node $a$ evaluated at the end of execution instance $I_a$)

  2. for each PPG node $b_i$ such that $(a, b_i, L) \in E_{mgoto}$ do

     (a) Create $I_{b_i}$, an execution instance of $b_i$, initialized with
         $H(I_{b_i}) := H(I_a) \circ < a, L >$, and update $\prec := \prec \cup \{(I_{b_i}, I_{b_i})\}$

     (b) /* Add the pair $(I_{b_i}, I_c)$ to $\prec$ for each instance $I_c$ that must wait for $I_{b_i}$ */
         for each sequence of $k \geq 0$ $E_{mgoto}$ edges,
         $S =< (x_1, x_2, L_1), (x_2, x_3, L_1), \ldots, (x_k, x_{k+1}, L_k) >$, such that $x_1 = b_i$ and all
         $x_j$'s are distinct do
         /* Note that $S =<>$ always satisfies this condition (when $k = 0$) */

           i. $P :=< x_1, L_1, x_2, L_2, \ldots, x_k, L_k >$ /* History suffix obtained from $S$ */
          ii. for each synchronization edge $(x_{k+1}, c, f)$ in $E_{sync}$ do
              for each execution instance $I_c$ of node $c$ that has been created do
              if $f(H(I_{b_i}) \circ P, H(I_c)) = true$ then $\prec := (\prec \cup \{(I_{b_i}, I_c)\})^*$ ;
              end for
              end for

         end for

     (c) /* Add the pair $(I_c, I_{b_i})$ to $\prec$ for each instance $I_c$ that $I_{b_i}$ must wait for */
         for each synchronization edge $(d, b_i, f)$ in $E_{sync}$ do
         for each sequence of $k \geq 0$ $E_{mgoto}$ edges $S =< (x_1, x_2, L_1), \ldots, (x_k, x_{k+1}, L_k) >$
         such that $x_{k+1} = d$ and all $x_j$'s are distinct do
         /* Note that $S =<>$ always satisfies this condition */

           i. $P :=< x_1, L_1, \ldots, x_k, L_k >$
          ii. for each execution instance $I_c$ of node $c = x_1$ that has been created do
              if $f(H(I_c) \circ P, H(I_{b_i})) = true$ then $\prec := (\prec \cup \{(I_c, I_{b_i})\})^*$ ;
              end for

         end for
         end for

  end for

end while
```

Figure 1: Scheduling System Algorithm

The effect of a PPG's execution is to map an input store to a final store ($\sigma_i \mapsto \sigma_f$). For convenience, we assume that the PPG's execution has no other observable effect on the outside world. An I/O or graphics device can be modeled as a special memory location that contains the entire sequence of state changes performed on the device, instead of just the final value written.

A store, σ, is a mathematical representation of the machine's memory, which maps memory location addresses to values ($\sigma : addr \mapsto val$). $\sigma[l]$ represents the value stored in location l. As in [7], we use the notation $\sigma[l \leftarrow v]$ to represent an updated store that is identical to the original store σ, except that $\sigma[l \leftarrow v]$ has the value v in location l. Both $\sigma[l]$ and $\sigma[l \leftarrow v]$ are assumed to be strict functions so that $\sigma[l] = \perp$ if $(\sigma = \perp) \vee (l = \perp)$, and $\sigma[l \leftarrow v] = \perp$ if $(\sigma = \perp) \vee (l = \perp) \vee (v = \perp)$.

We begin by defining the semantics of a single execution instance of a PPG node. A PPG node's computation consists of reading some values from the store, evaluating a result, and writing the result into a single location in the store[2]. We assume that the node's evaluation is deterministic, so that it always computes the same result value from the same set of input values. The operational semantics of an execution instance of PPG node n is defined with respect to two parameters that provide the context for the execution instance:

1. An input store, σ_i.
2. A set of "parallel writes", $PAR_{writes} = \{(l_1, v_1), (l_2, v_2), \ldots\}$ = the set of location-value pairs for which a write access may be performed in parallel with the given execution instance of node n.

σ_i is required for specifying node n's input values. PAR_{writes} is required for defining the parallel semantics of read-write and write-write hazards, based on the non-atomic memory access model assumed in Section 2.

Definition 4.1 The *operational semantics of an execution instance of PPG node n* is defined by the store mapping $n : \sigma_i \mapsto \sigma_f$ with respect to a given set of parallel writes

[2]For convenience, we assume that a PPG node writes into a single location in the store. The semantics can be easily extended to handle nodes that write into multiple locations in the store. To represent such a node in the current semantics, the node should be expanded into multiple nodes, one for each write location.

PAR_{writes}, where

$$\sigma_f = \begin{cases} \perp & \text{if the execution instance of node } n \text{ performs a read access} \\ & \text{on location } l\text{, and } \exists\,(l_j, v_j) \in PAR_{writes} \text{ s.t. } (l_j = l) \wedge (v_j \neq \sigma_i[l]) \\ & \text{(read-write hazard)} \\ \perp & \text{if the execution instance of node } n\text{'s computation incurs an exception} \\ & \text{(e.g. non-termination, reading a } \perp \text{ value from } \sigma_i\text{, overflow, etc.)} \\ \sigma_i[l \leftarrow \perp] & \text{if the execution instance of node } n \text{ attempts to write value } v \text{ into} \\ & \text{location } l\text{, and } \exists\,(l_j, v_j) \in PAR_{writes} \text{ s.t. } (l_j = l) \wedge (v_j \neq v) \\ & \text{(write-write hazard)} \\ \sigma_i[l \leftarrow v] & \text{otherwise, assuming that the execution instance of node } n \text{ writes} \\ & \text{value } v \text{ into location } l \end{cases}$$

□

To obtain the operational semantics of a PPG, we define a *scheduled sequence* to be the sequence of node execution instances scheduled in step 1 of the while loop in Figure 1, assuming that the PPG terminates successfully for the given input store. For a given PPG and input store, there may be several possible scheduled sequences, based on which "ready" execution instance is chosen each time step 1 is performed.

Definition 4.2 Let $< N_1, \ldots, N_k >$ be *any* scheduled sequence of PPG node execution instances obtained for a given PPG G and input store σ_i, and let

- $\prec$ be the partial order on $\{N_1, \ldots, N_k\}$, defined by the scheduling system algorithm in Figure 1.
- $PAR_{writes}(N_j) = \{(l, v) | \exists\, N_i \text{ s.t. } (N_i \not\prec N_j) \wedge (N_j \not\prec N_i) \wedge$ (execution instance N_i writes value v into location l)$\}$, be the set of parallel writes for each execution instance N_j, $1 \leq j \leq k$.

Then, *the operational semantics of PPG G* is defined to be the store mapping $G : \sigma_i \mapsto \sigma_f$, where $\sigma_f = N_k(\ldots N_1(\sigma_i)\ldots)$, and the $PAR_{writes}(N_j)$ set defined above is used in determining the store mapping for each execution instance N_j, $1 \leq j \leq k$ (as specified in Definition 4.1).

If the scheduling system does not yield a complete scheduled sequence in which all created execution instances are scheduled (due to some exception), then $G(\sigma_i)$ is defined to be $= \perp$. □

Lemma 4.3 Consider any two scheduled sequences $S_1 = < N_{1,1}, N_{1,2}, \ldots >$ and $S_2 = < N_{2,1}, N_{2,2}, \ldots >$ that may be obtained for a given PPG G and input store σ_i, and consider any PPG node a for which there is an execution instance $N_{1,j}$ in S_1 and an execution instance $N_{2,k}$ in S_2 such that $H(N_{1,j}) = H(N_{2,k})$. Let the resulting stores after executing $N_{1,j}$ in S_1 and $N_{2,k}$ in S_2 be $\sigma_1 = N_{1,j}(\ldots N_{1,1}(\sigma_i)\ldots)$ and $\sigma_2 = N_{2,k}(\ldots N_{2,1}(\sigma_i)\ldots)$ respectively. Then $\sigma_1 \neq\perp$ and $\sigma_2 \neq\perp$ implies that execution instances $N_{1,j}$ and $N_{2,k}$ performed read/write operations on the same locations and with the same values. □

Proof (by induction on length of scheduled sequences): [Sketch] Since $\sigma_1 \neq \perp$ and $\sigma_2 \neq \perp$, no read access in execution instance $N_{1,j}$ incurred a read-write hazard in S_1 (and the same for execution instance $N_{2,k}$ in S_2). Therefore, the read accesses performed by $N_{1,j}$ ($N_{2,k}$) were properly guarded by the PPG's mgoto and synchronization edges in scheduled sequences S_1 (S_2), and all writes to $N_{1,j}$'s ($N_{2,k}$'s) input locations must have been performed before execution instance $N_{1,j}$ ($N_{2,k}$) was scheduled. Since execution instances $N_{1,j}$ and $N_{2,k}$ have the same execution history, each input location accessed by $N_{1,j}$ and $N_{2,k}$ must contain the same value in the two cases (by induction on their predecessors). With the same input location values, execution instances $N_{1,j}$ and $N_{2,k}$ of node a must compute the same result and attempt to store it in the same output location, in both S_1 and S_2. Since the condition for an execution instance incurring a write-write hazard depends on the PAR_{writes} set and not on the ordering of predecessors in the scheduled sequences, execution instance $N_{1,j}$ must store the same output value in σ_1 as execution instance $N_{2,k}$ in σ_2 (the output value will be $\perp$ in the case of a write-write hazard). $\Box$

Theorem 4.4 [Reordering Theorem] For a given PPG G and input store σ_i, the final store $\sigma_f = G(\sigma_i)$ obtained (according to Definition 4.2) is the same for all possible scheduled sequences that can be chosen by the scheduling system. $\Box$

Proof (by contradiction, and induction on length of scheduled sequences): [Sketch] Assume that there exist two scheduled sequences $S_1 = < N_{1,1}, N_{1,2}, \ldots >$ and $S_2 = < N_{2,1}, N_{2,2}, \ldots >$ that yield different final stores $\sigma_1 \neq \sigma_2$. Two cases arise:

1. If some execution instance in S_1 (say, execution instance $N_{1,j}$ of node a) causes σ_1 to be $= \perp$ then a corresponding execution instance of node a in S_2 (say, $N_{2,k}$) with $H(N_{1,j}) = H(N_{2,k})$ must force $\sigma_2 = \perp$.

2. If $\sigma_1 \neq \perp$ and $\sigma_2 \neq \perp$ then each pair of execution instances in S_1 and S_2 with the same execution history and the same source PPG node must have the same input and output values (by Lemma 4.3), which means that S_1 and S_2 must take the same conditional branches, eventually yielding $\sigma_1 = \sigma_2$.

Contradiction in both cases. $\Box$

5 Relating Program Dependence Graphs (PDGs) to Parallel Program Graphs (PPGs)

A Program Dependence Graph (PDG) consists of a set of nodes connected by *control dependence* and *data dependence* edges, and is derived from a sequential program [10]. A control dependence edge is a triple of the form (a, b, L) which defines a control dependence from node a to node b with branch label L. A data dependence edge is a triple of the form (a, b, C) which defines a data dependence from node a to node b with context information C that identifies the execution instances of a and b that are involved in the data dependence.

By default, the context information C states that the dependence must hold for each *plausible* pair of execution instances I_a and I_b i.e., for each (I_a, I_b) pair such that I_b is executed after I_a in the original sequential program. However, in many important cases, the context information C contains *direction vectors* [20] or *distance vectors* [15], which give us more information about the execution instances involved in loop-carried data dependences. Direction vectors and distance vectors are defined with respect to a nest of single-entry loops enclosing nodes a and b in the sequential program.

It is easy to build a PPG from a given PDG. The same set of nodes is used in both cases. The control and data dependence edges in the PDG correspond to mgoto edges and synchronization edges in the PPG. A control dependence edge from the PDG can directly be used as an mgoto edge in the PPG. A data dependence edge from the PDG can be used as a synchronization edge in the PPG by translating the context information C to an appropriate synchronization condition f on execution histories. For convenience, we will use the notation $PPG(G)$ to refer to the PPG corresponding to PDG G.

However, not all PPGs are derivable from PDGs, which is why PPGs are more general than PDGs. This issue has been studied in [9, 4]. It was shown that there is a class of PPGs for which it is not possible to generate an equivalent sequential control flow graph without duplicating code or inserting boolean guards. We call this class of PPGs *non-serializable*. A PPG for which it is possible to generate a sequential control flow graph (without code duplication or insertion of boolean guards) is called *serializable*. Figure 2 shows a non-serializable PPG (with five mgoto edges and no synchronization edges). Note that node 4 will be executed twice if predicate nodes 1, 2, and 3 all evaluate to *true*. This PPG can be made serializable by splitting node 4 into two copies, one for predicate node 2 and one for predicate node 3.

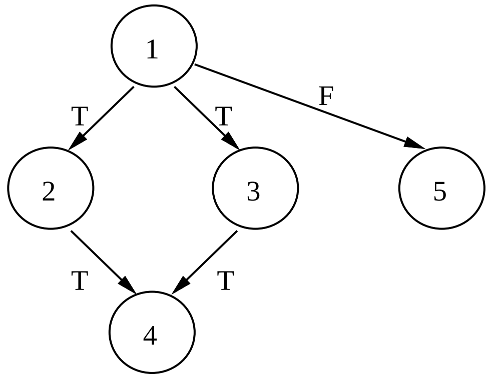

Figure 2: Example of a non-serializable PPG

Just as we defined $PPG(G)$ to be the PPG corresponding to PDG G, we define $PPG(P)$ to be the PPG corresponding to a sequential program P. A sequential program is defined by its *control flow graph* [1]. The control flow graph itself can be viewed as a degenerate PPG in which there are no synchronization edges and in which all mgoto edges are sequential (i.e., for a given node a and branch label L, there is at most one node b such that $(a, b, L) \in E_{mgoto}$). Let $< N_1, \ldots, N_k >$ be the execution instance sequence obtained by executing a sequential program P as a PPG on input store σ_i, assuming that program P terminates successfully. Then, the operational semantics of sequential program P will be the composition of the individual execution instance mappings, $P(\sigma_i) = N_k(\ldots N_1(\sigma_i)\ldots)$, assuming that $PAR_{writes}(N_j) = \emptyset \ \forall 1 \leq j \leq k$. If program P does not terminate successfully for input store σ_i then $P(\sigma_i) = \perp$.

We conclude this section with the *Equivalence Theorem* which states that the semantics of a sequential program, P, is identical to the semantics of its PDG, G, by using the execution semantics defined in Section 4 to show that $PPG(P)$ and $PPG(G)$ must have the same semantics.

Theorem 5.1 [Equivalence Theorem] A sequential program, P, and its corresponding PDG, G, have identical semantics i.e. $P(\sigma_i) = G(\sigma_i)$, for all input stores σ_i. $\square$

Proof: [Sketch] The execution instance sequence $< N_1, \ldots, N_k >$ obtained from executing the sequential program P should be a valid scheduled sequence for PDG G. We assume that PDG G correctly expresses all the control and data dependences in program P, which implies that no scheduled sequence for G (including $< N_1, \ldots, N_k >$) incurs a read-write hazard or a write-write hazard. Therefore, the PAR_{writes} sets in Definition 4.2 have no impact on PDG G's operational semantics. So, using Definition 4.2 and Theorem 4.4, we have $P(\sigma_i) = N_k(\ldots N_1(\sigma_i)\ldots) = G(\sigma_i)$. $\square$

6 Related Work

The idea of extending PDGs to PPGs was introduced by Ferrante and Mace in [9], and further extended by Alpern, Ferrante, and Simons in [4]. Several restrictions were imposed on the PPGs defined in [9, 4]: a) the PPGs could only contain control flow edges but no loop-carried data dependence (synchronization) edges, b) only a special kind of node (FORALL) was allowed to transfer control to more than one target node, c) only FORALL nodes were allowed to be *merge* nodes i.e., to have multiple incoming edges, d) only a predicate node could be the source of multiple non-intersecting paths that reach the same merge node, and e) only reducible loops were considered. The PPGs defined in this paper have none of these restrictions. Restrictions a) and e) limited 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 [20]. Restrictions b) and c) are minor; in their PPGs, a multiple-goto operation or a merge operation is explicitly represented by a FORALL operator, whereas in our PPGs, these operations are implicit in the definition of the E_{mgoto} edges. Restriction d) prohibited a FORALL operator from creating multiple execution instances of the same node e.g. the PPG shown in Figure 2 cannot be expressed in their PPG model because it is possible for node 4 to be executed twice. The Hierarchical Task Graph (HTG) proposed by Girkar and Polychronopoulos [11] is also a form of a parallel program graph applicable only to structured programs that can be represented by the hierarchy defined in the HTG.

Apart from the restrictions in the PPG model, the major difference with [9, 4] is that the problem addressed by their work was that of "linearizing" parallel code i.e., of determining when it is possible to transform a given PPG into an equivalent sequential control flow graph without duplicating code or inserting boolean guards. In this paper, we address the problem of defining the concurrent execution semantics of PPGs, whether or not the PPGs can be linearized.

As mentioned earlier, the semantics of PDGs has been examined in past work by Selke and by Cartwright and Felleisen. In [18], Selke presented an operational semantics for PDGs based on graph rewriting. In [7], Cartwright and Felleisen presented a denotational semantics for PDGs, and showed how it could be used to generate an equivalent functional, dataflow program. Both those approaches assumed a restricted programming language (the language W), and presented a value-oriented semantics for PDGs. The value-oriented nature of the semantics made it inconvenient to model store-oriented operations that are found in real programming languages, like an update of an array element or of a pointer-dereferenced location. Also, the control structures in the language W were restricted to `if-then-else` and `while-do`; the semantics did not cover programs with arbitrary control flow. In contrast, our semantics is applicable to PDGs with arbitrary (unstructured) control flow and arbitrary data read and write accesses, and more generally to PPGs as defined in this paper.

In [18], Selke presented a *graph rewriting semantics* for PDGs that is similar in spirit to the graph reduction model for executing functional programs. The program execution model in [18] is based on rewriting rules that incrementally create a new PDG from an old PDG. This model has no explicit store. Instead, values are propagated along data flow edges into the expressions that need them. [18] contains a result similar to the Reordering Theorem for the graph rewriting model, which states that the parallel rewriting of a set of redexes yields the same result as any sequential rewriting of those redexes, *for a deterministic PDG*. However, Theorem 4.4 is applicable to both deterministic and non-deterministic PPGs. This is because non-determinism manifests itself as a write-write or a read-write hazard in our model, which results in a $\perp$ value for the location or for the entire store.

In [7], Cartwright and Felleisen presented a *demand-oriented denotational semantics* for the W language, and shows how the denotational definition of a program can be used to to generate its "semantic PDG". The semantic PDG is like an executable dataflow graph that performs a demand-driven (non-strict) execution of the original program. However, the programming languages that motivate the use of the PDG (e.g., Fortran, Pascal, C) have strict semantics, which differs from the non-strict PDG execution semantics assumed in [7]. [7] contains a result stating that the execution semantics of a PDG (either *def-order* or *output*) may be different from the semantics of sequential execution (specifically, the PDG semantics *dominates* the sequential semantics). One reason for the difference is that they assumed a strict store update function for the sequential semantics and a non-strict store update function for the PDG semantics. However, the Equivalence Theorem is valid in our model because we use a strict store update function for both sequential programs and PPGs, and also because of the semantics given to read-write and write-write hazards.

The work done in [18] and [7] attempts to the bridge the semantic gap between an imperative language and a functional executional model by providing a functional-style semantics for PDGs. This research direction has also been undertaken in the field of compilers, where there is ongoing work in translating imperative programs to dataflow graphs [17, 6]. In contrast, this paper provides an imperative-style semantics for PPGs and PDGs, so as to represent in their full generality imperative-style parallel programming models that allow for arbitrary (unstructured) control flow as well as arbitrary data read and write accesses.

7 Conclusions and Future Work

In this paper, we have presented a concurrent execution semantics for Parallel Program Graphs (PPGs), a general parallel program representation that also encompasses PDGs and sequential programs. We believe that this semantics is natural to the programmer's way of thinking, and that it also provides a suitable execution model for efficient implementation on real architectures. The robustness of the semantics is demonstrated by the Reordering Theorem and the Equivalence Theorem.

We see several possible directions for future work based on the PPG representation and the concurrent execution semantics presented in this paper:

- *Extend sequential program analysis techniques to the PPG*
  All program analysis techniques currently used in compilers (e.g., constant propagation, induction variable analysis, computation of Static Single Assignment form, etc.) operate on a sequential control flow graph representation of the program. There has been some recent work in the area of program analysis in the presence of structured parallel language constructs [16, 19]. However, just as the control flow graph is the representation of choice (due to its simplicity and generality) for analyzing sequential programs, it would be desirable to use the PPG representation (possibly restricted to just mgoto edges) as a simple and general representation for parallel programs.

- *Use semantics to prove legality of program transformations*
  Both the PPG representation and its execution semantics presented in this paper should lend themselves nicely to specifications and legality proofs for program transformations. The PPG representation is at a level that is most convenient for specifying program transformations, compare to the source-language level and the machine-instruction level. The semantics provides a complete description of the result of a PPG's execution, and avoids the kind of ambiguity problems that arise when the concurrent execution semantics for a parallel programming language is partly unspecified and left open to interpretation.

- *Hardware support features for scheduling system*
  Even though the scheduling system defined in Section 3 was presented as a theoretical abstraction, it lends itself nicely to efficient hardware implementations. The parallel

flow of control defined by *mgoto* edges is similar to lightweight threads in a multi-threaded architecture; an obvious optimization is to not create a new thread for a sequential branch, but only when control is transferred to two or more parallel branch targets. In general, it can be expensive to implement an arbitrary synchronization function for a PPG edge. However, in many cases the synchronization function can be implemented by incrementing and decrementing counting-semaphore variables, which can be performed very efficiently if hardware support is available in the form of synchronization registers. It would also be interesting to pursue hardware implementations for the restricted class of PPGs that contain only mgoto edges.

- *Develop a common compilation and execution environment for different parallel programming languages*
  The PPG can be used as the basis for a common compilation and execution environment for parallel programs written in different languages. The scheduling system defined in Section 3 can be developed into a PPG-based interpreter, debugger, or runtime system for parallel programs, that provides feedback to the user about the parallel program's execution (e.g. parallelism profiles, detection of read-write or write-write hazards, detection of deadlock, etc.). Currently, such programming tools are being developed separately for different languages, with different (ad hoc) assumptions being made about their parallel execution semantics. The PPG representation could be useful in leading to a common environment for different parallel programming languages.

Acknowledgments

The author would like to thank Barbara Simons for her detailed comments and suggestions, and Jeanne Ferrante for early discussions related to this work.

References

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

[2] Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An Overview of the PTRAN Analysis System for Multiprocessing. *Proceedings of the ACM 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, 5(5) pages 617-640.

[3] Frances Allen, Michael Burke, Ron Cytron, Jeanne Ferrante, Wilson Hsieh, and Vivek Sarkar. A Framework for Determining Useful Parallelism. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 207–215, July 1988.

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

[5] William Baxter and J. R. Bauer, III. The Program Dependence Graph in Vectorization. *Sixteenth ACM Principles of Programming Languages Symposium*, pages 1 – 11., January 11-13 1989. Austin, Texas.

[6] Micah Beck and Keshav Pingali. From Control Flow to Dataflow. Technical report, Department of Computer Science, Cornell University, October 1989. TR 89-1050.

[7] Robert Cartwright and Mathias Felleisen. The Semantics of Program Dependence. *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, June 1989.

[8] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Experiences Using Control Dependence in PTRAN. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989. In Languages and Compilers for Parallel Computing, edited by D. Gelernter, A. Nicolau, and D. Padua, MIT Press, 1990 (pages 186-212).

[9] 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.

[10] 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.

[11] 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.

[12] Rajiv Gupta and Mary Lou Soffa. A Reconfigurable LIW Architecture and its Compiler. *Proc. of the 1987 Int'l Conf. on Parallel Processing*, Aug. 1987.

[13] Rajiv Gupta and Mary Lou Soffa. Region Scheduling. *Proc. of the Second International Conferenece on Supercomputing*, 3:141–148, May 1987.

[14] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating Non-Interfering Versions of Programs. *Conf. Rec. Fifteenth ACM Symposium on Principles of Programming Languages*, pages 133–145, January 1987.

[15] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[16] Samuel Midkiff, David Padua, and Ron Cytron. Compiling Programs with User Parallelism. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[17] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. Maccabe. Gated Single-Assignment Form: Dataflow Interpretation for Imperative Languages. *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, White Plains, New York*, pages 257–271, June 1990.

[18] Rebecca Parsons Selke. A Rewriting Semantics for Program Dependence Graphs. *Sixteenth ACM Principles of Programming Languages Symposium*, January 11-13 1989. Austin, Texas.

[19] Harini Srinivasan and Michael Wolfe. Analyzing Programs with Explicit Parallelism. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing*, August 1991. To be published by Springer-Verlag.

[20] 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 This monograph is a revised version of the author's Ph.D. dissertation published as Technical Report UIUCDCS-R-82-1105, U. Illinois at Urbana-Champaign, 1982.