Incremental Graph Parsing

C.L.McCreary[1], S.S. Roth
Department of Computer Science and Engineering
Auburn University, Alabama 36849
mccreary@eng.auburn.edu

Abstract

A Program Dependence Graph is a useful representation of the dependencies within a code module, with the nodes representing code segments and the edges showing dependency constraints. Through this representation, inherent parallelism in the program structure can be exploited by various heuristics. This paper presents an incremental graph decomposition algorithm which parses the graph into a structure called a Parse Tree that aids in the analysis. The algorithm presented executes in $O(n^2)$ time.

Keywords: modular decomposition, graph parsing, graph theory

Introduction

One of the basic problems in parallelizing programs is the recognition of program dependences and the partitioning of the program into segments that can be executed concurrently. To solve this problem, many researchers create an intermediate representation of the program which is some form of Program Dependence Graph (PDG) where nodes represent code segments and edges show dependency constraints. Through some heuristic, nodes are grouped into grains that are executed serially on a single processor. Our approach is to analyze the PDG by identifying all possibilities for concurrent execution as well as the dependences which force serial execution. A structure we call the *Parse Tree* is created to aid in the analysis.

1.This research is supported by National Science Foundation Grant CCR-9203319

This paper discusses an incremental modular decomposition of directed acyclic graphs (DAGs) into a hierarchy, the parse tree, of subgraphs called clans. Clans are similar to other structures developed for undirected graph decomposition such as *autonomous sets* [1] and *modules* [3]. The algorithm takes $O(n^2)$ time to create the resulting hierarchy of clans from a DAG which represents the PDG.

Benefits of this algorithm include the clear representation of dependencies within the code, and therefore a more obvious indication of opportunities for parallelization. The parse tree also preserves the original meaning of the PDG by destroying no dependencies in the original graph and adding a minimal number of new dependency edges to preserve a well-structured parse tree and clear conditions for concurrent execution. Representations created by the algorithm are unique to each PDG.

Description of Clans

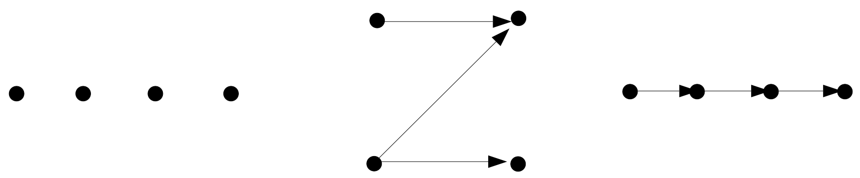
The formal description of a clan structure is:

> A set of vertices C in graph G is a clan iff for all nodes x, y in C and z in G-C:
> (i) z is an ancestor of x iff z is an ancestor of y
> (ii) z is a descendant of x iff z is a descendant of y

A trivial clan is one that contains singleton sets and the entire graph.

There are three types of clans : Primitive, Independent, and Linear. Let C be a clan and let $\{C_1, C_2, \ldots, C_k\}$ be a partition of C where each C_i is a subclan of C. The *quotient graph* of C is the graph with vertices $C_1, C_2, \ldots, C_k$. (C_i, C_j) is an edge when (x,y) is an edge of C and $x \in C_i$ and $y \in C_j$. Clan C is classified as (i) independent if every subgraph of its quotient graph is a clan; (ii) linear if for every pair of vertices

C_i and C_j in the quotient graph, C_i is either an ancestor or descendant of C_j or (iii) primitive if the only clans of the quotient graph are trivial clans [Figure 1]. Independent clans lend themselves to parallelization because each subclan has no dependence on any of the others. Because of this, each subclan can be placed on a separate processor. Linear clans require serial execution, because each subclan depends on its ancestors. Primitive clans can be broken down further into their independent and linear components before it becomes clear which segments should be parallelized.

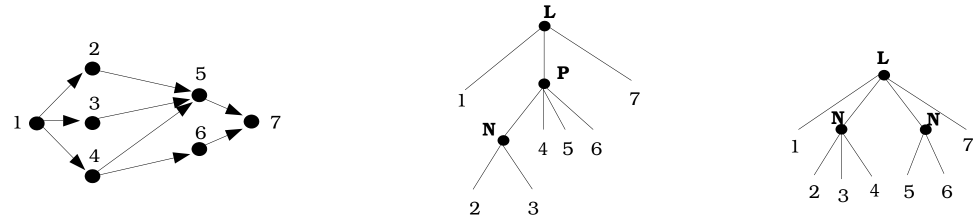


(a) Independent (b) Primitive (c) Linear

Types of Clans
Figure 1

Objective of the Algorithm

The objective of the algorithm is to create a clear representation of the dependency structure of the PDG by creating a hierarchy of clans, alternating by level between Linear and Independent clans. Figure 2 shows a DAG and its corresponding parse trees. Linear, primitive, and independent clans are labeled L, P, and N respectively. To maintain the alternating parse tree, primitives require the addition of edges to the PDG. By maintaining the internal structure of the graph as far into the process as possible we attempt to minimize the number of additional edges. We can also guarantee that no dependences from the original PDG will be removed.



A graph and its corresponding parse trees with and without primitives
Figure 2

The Algorithm

The algorithm takes as input a directed acyclic graph with nodes labeled in topological order from 0 to n-1. Each node, x, is inspected in order and added to the parse tree by examining it's relationship to the current parse tree.

All nodes in the current parse tree are examined and the nodes are relabeled each time a new node, x, is added. The leaf nodes of the parse tree correspond to DAG nodes and are labeled A (I) meaning that the leaf is (not) an ancestor of x. Because nodes are added in topological order, it is impossible for x to be an ancestor of a tree node. There are three possible labels for interior nodes : A, I, and Φ. An 'A' label denotes that all of the interior node's children are ancestors of x, while an 'I' label denotes that all of the interior node's children are independent of x. A **'**Φ' label denotes that some of the children of the interior node are ancestors of x and that some are independent of x. If any child of an interior node is labeled **'**Φ**'**, then the interior node is automatically labeled **'**Φ**'** as well.

Primitive Clans are the main stumbling block in exploiting opportunities for parallelism. This is because it is unclear how to divide a primitive section of the graph into clusters that can be executed serially or in parallel. A Primitive can be recognized by any clan in the parse tree that contains a child labeled 'Φ'. The 'Φ' marking indicates that the child is neither a true ancestor of nor is completely independent of x.

Linear clans must list their children in the order of their dependency on one another, with all children dependent on the leftmost child, and none dependent on the rightmost child. All ancestors of child C must be placed to its left, and all descendants to its right.

Lemma 1:Children of a Linear clan are labeled with nodes marked 'A' to the left of nodes marked 'Φ' to the left of nodes marked 'I'.

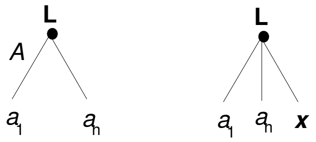
Proof: If a child C_i of the Linear is marked 'A' then every node in C_i is an ancestor of x. All nodes to the left of C_i must also be ancestors of x because they are ancestors of C_i. If a child C_j of the Linear is marked 'I', then no child to the right of C_j has a node that is an ancestor of x. For if $y \in C_k$ to the right of C_i is an ancestor of x, some $z \in C_j$ is an ancestor of y because C_j is linearly connected to C_k. But then z is an ancestor of x, contradicting the 'I' label.■

In an effort to preserve as much of the original graph structure as possible, edges are never deleted the PDG, inly added. In some instances where the majority of the parse tree nodes are ancestors, and those nodes that are independent are several levels above the node to be added in the PDG, dependencies can be over-serialized and destory the graph's original structure. Due to this, the labeling of the parse tree can be altered. If the node being examined is not the root node of the parse tree and the number of Independent children that it contains is less than 20% of the total number of ancestor children in that branch, then the entire branch is labeled 'A' instead of 'Φ'. As only independent children are "ignored" in this manner, no dependencies are destroyed. This preserves the structure of the graph as well as the inherent parallelism.

Once the root node has been labeled, the new node, x, is inserted into the parse tree according to the root label. Several modifications to the tree require recursive calls to be made to lower branches in the tree. This process continues until all nodes in the graph have been examined, and the final n-ary parse tree has been completed. The total complexity of the algorithm is $O(N^2)$, with the labeling of the graph taking O(n) time, as the labeling is only done once per iteration.

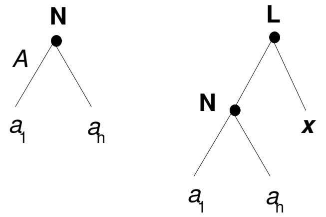
```
The Algorithm:
INCREMENTAL_PARSE(G)
// INPUT: A directed acyclic dependency graph, G, with topologically sorted nodes
              labeled 0 to n-1
// OUTPUT: A N-ary tree representing the relationships between the nodes in the
                graph.

    LOOP until Node_Number = (n-1), where n = number of nodes contained in G.
       INSERT(Node_Number, T);
    END LOOP
END INCREMENTAL_PARSE;

INSERT(Node_Number, T);
     Label each node in the Parse Tree, T, according to its relationship to x, the
        node to be added.

    CASE Root is marked:
         A:
```

A_L: If the node is a Linear Subclan, then add x, as the rightmost branch.



Adding x to a Linear Marked 'A'

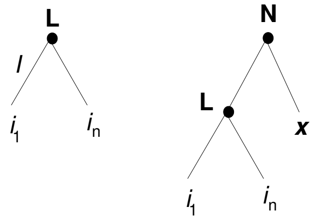
A_N: If the node is an Independent Subclan, then create a new Linear root with the original Independent as the left child, and x as the right child.



Adding x to an Independent Marked 'A'

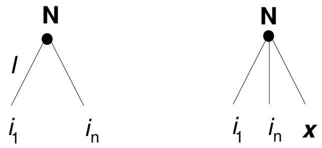
I:

I_L: If the node is a Linear Subclan, then create a new Independent root with the original Linear node as the left child, and x as the right child.



Adding x to a Linear Marked 'I'

I_N: If the node is an Independent Subclan, then add x as the rightmost child.

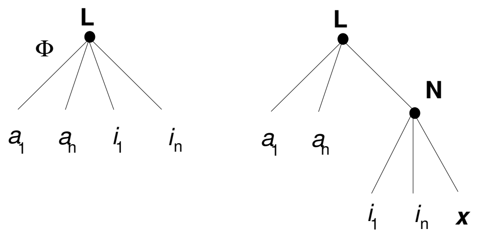


Adding x to an Independent Marked 'I'

Φ:

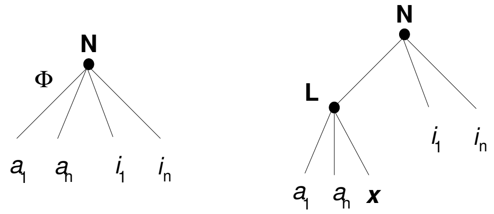
If the Subclan does not contain any Φ children:

Φ_L: If the node is a Linear Subclan, create an Independent child containing all children of the Linear marked I and x as the rightmost child in the new Independent.



Adding x to a Linear marked Φ

Φ_N: If the node is an Independent Subclan, create a Linear child containing all children of the Independent marked A and x as the rightmost child.

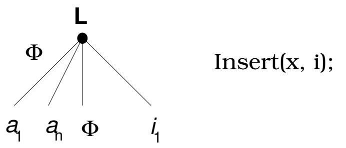


Adding x to an Independent Marked Φ

If the Subclan contains one Φ child:

Φ1L: If the Subclan is a Linear:

(a) If the Linear contains at least one child marked A and at least one marked I, then add x by calling the algorithm recursively on the child marked I.



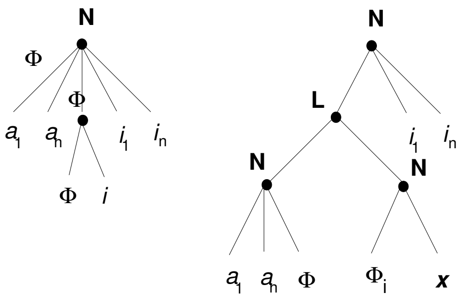
Adding x to a Linear with one Φ child

(b) Else recursively call the algorithm on the child marked Φ.

Φ1N: If the Subclan is Independent:

(a) If the Independent contains at least one child marked A:

1. All children marked I remain placed as they are.
2. Create a Linear containing two Independent children:
   - N_1 contains the children on the original Independent marked A and the grandchildren on the Independent marked Φ.
   - N_2 contains the grandchildren of the original Independent marked I contained in the Φ child and the new node, x, as the rightmost branch. If the child marked Φ contains no grandchildren on the original Independent marked I, then x becomes a singleton child of the Linear formed in the transformation.

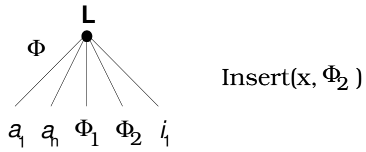


Adding x to an Independent with one Φ child

(b) Else call the algorithm recursively on the child marked Φ**.**

If the Subclan contains more than one child marked Φ:

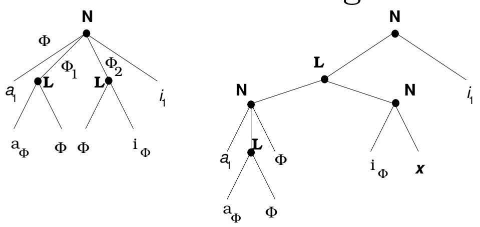
Φ2L: If the Subclan is a Linear, call the algorithm recursively on the last child marked Φ.



Adding x to a Linear with two children marked Φ

Φ_{2N}: If the Subclan is an Independent:

1. All children marked I remain placed as they are.
2. Create a Linear containing two Independent children:
   - N_1 contains the children and grandchildren of the original independent marked A and the grandchildren of the original independent marked Φ, with any linear connections between them preserved.
   - N_2 contains all grandchildren of the original independent marked I and x as the rightmost branch.



Adding x to an Independent with two children marked Φ

END INSERT;

Lemma 2: The algorithm INSERT preserves DAG dependences as it places x in the tree.

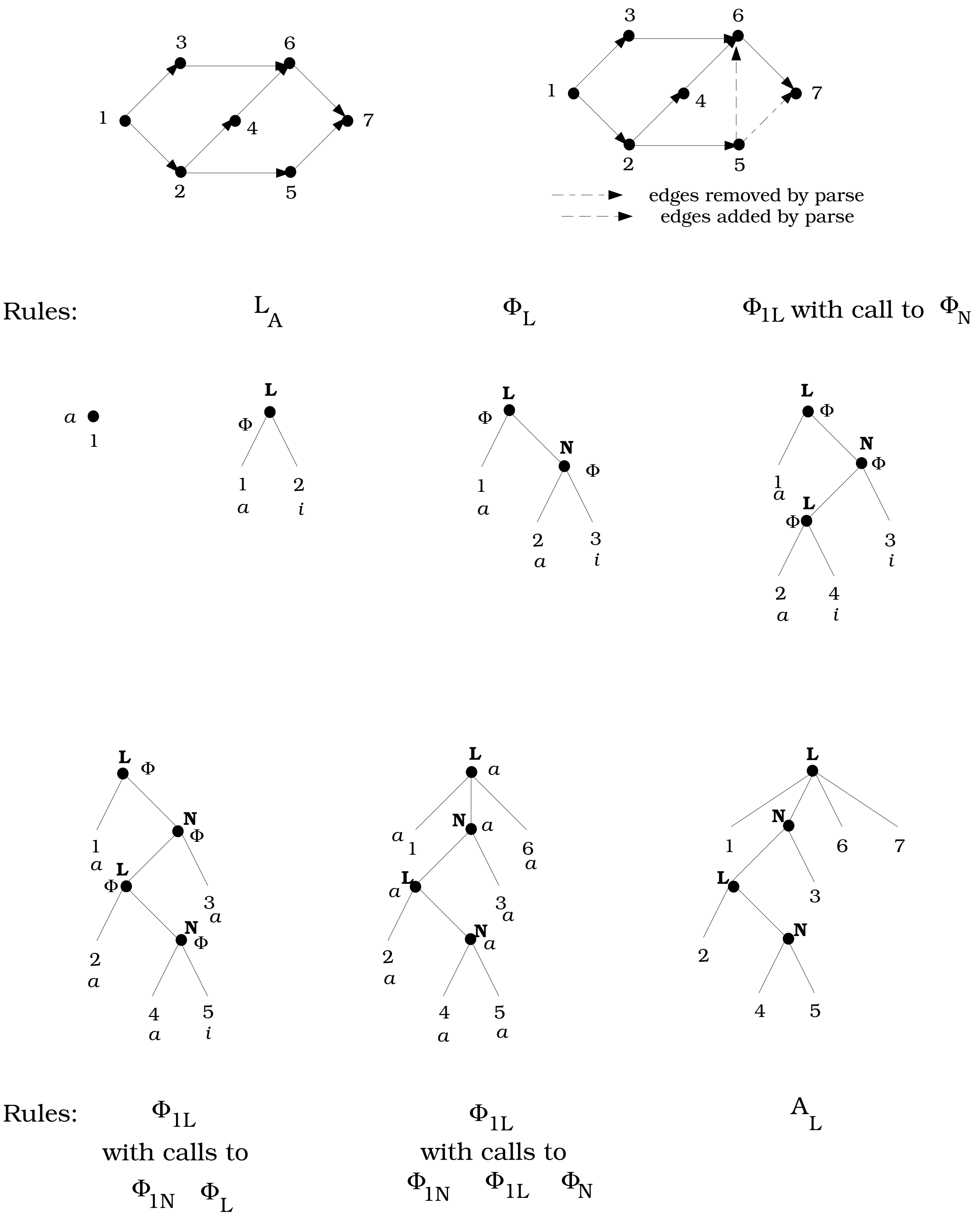
An example graph and its parse is shown in Figure 3. The labels with the figures denote which rule had been applied. For example, when adding node 4, the root node is a Linear Clan labeled 'Φ', and it contains one child marked 'Φ'. Therefore, we apply rule Φ_{1L}, which requires a recursive call to the Insert subroutine with the Independent marked 'Φ' as the root. After this call has been made, we apply rule Φ_N because the Independent has no children marked 'Φ'. On applying this rule, a Linear child is created containing nodes 2 and 4.

Summary

A clear representation of the opportunities for parallelism in a program is needed to simplify the decision-making process for scheduling. The clan hierarchy represented in a parse tree can provide this representation in a time $O(n^2)$, a time equal to or better than the times taken to create the similar structures of [1] and [3]. Clan structures are ideal for a parallelization application in that they preserve the original dependencies and resolve cases where it is unclear whether or not to run the cluster concurrently or serially. This research is currently being used to form the dependency analysis part of a automated parallelizing compiler funded by NSF.

References

1. Buer, H. and R.H. Mohring. A Fast Algorithm for the Decomposition of Graphs and Posets. *Mathematics of Operations Research.* Vol. 8, No. 2, May 1983, pp. 170-184.
2. McCreary, C.L. and D.H. Gill. Automatic Determination of Grain Size for Efficient Parallel Processing. *Communications of the ACM*, Vol. 32, No. 9, Sept. 1989, pp.1073-1078.
3. Muller, J.H. and J.Spinrad. Incremental Modular Decomposition. *Jounral of the ACM*, January 1989, pp. 1-19.



A sample parse
Figure 3