A Graph Parsing Algorithm and Implementation

C. L. McCreary and A. Reed
Department of Computer Science and Engineering
Auburn University

1 Introduction

This paper presents an algorithm for decomposing directed acyclic graphs (DAGs) into a heirarchy of subgraphs we call clans. The resulting parse tree is being used to partition and schedule program dependence graphs for efficient concurrent execution on a parallel system. The clans can be identified as able or unable to support parallel execution, and their connections with other clans is completely characterized in the parse tree.

2 Definitions and Concepts

2.1 Graph definitions and notations

A *directed graph* G = (V,E) consists of a set V of vertices and a set $E \subseteq VxV$ of directed edges such that each edge $e \in E$ is associated with a unique pair, (v,w) of vertices. The set of edges is written E(G) and the set of vertices is V(G) or simply G. For $(v,w) \in E(G)$, v is called a *parent* of w and w is a *child* of v. A *path* from v_0 to v_n in directed graph G is a sequence of edges $\{(v_0,v_1),(v_1,v_2),...,(v_{n-1},v_n)\}$ in E(G). If there is a path from v to w in directed graph G, then v is said to be an *ancestor* of w and w is a *descendant* of v.

For any directed graph G, the *source* of G is the set of nodes S where for each $s \in S$ there is no incoming edge. $S = \{s \in G \mid (v,s) \notin E(G) \text{ for any } v \in G\}$. The *sink* of G is the set of nodes M such that for each $m \in M$, there is no outgoing edge.
$M = \{m \mid (m,v) \notin E(G) \text{ for any } v \in G\}$.

A *transitive reduction* or *Hasse graph* of directed graph G is a graph H(G) where

(i) there is a directed path from u to v in H(G) if and only if there is a directed path from u to v in G, and

(ii) there is no graph with fewer edges than H(G) that satisfies condition (i).

Intuitively, we can describe a Hasse graph as a graph with no short-cuts. If there is a path from u to v containing at least one node different from u or v, then no (u,v) edge exists in the Hasse graph. The Hasse graph of an directed acyclic graph is unique [1]. The parse will be found using the Hasse graph corresponding to the input DAG.

2.2 Clans

The central focus of our decomposition algorithm is the clan. Let G be a dependency graph. A subset $X \subseteq G$ is a clan iff for all x, $y \in X$ and all $z \in G - X$,

(a) z is an ancestor of x iff z is an ancestor of y, or

(b) z is a descendant of x iff z is a descendant of y.

An alternate description of a clan depicts it as a subset of nodes where every element not in the subset is related in the same way (i.e. ancestor, descendant or neither) to each member in the subset. Trivial clans include singleton sets and the entire graph. In Figure 1, sets {2,3,4}, {2,3,4,5}, {1,2,3,4}, and {3,4} are the nontrivial clans. C={2,3,4} is a clan since node 1 is an ancestor of each element of C and 5 is a descendant of each element of C. The set {2,5} is not a clan since 3 and 4 are ancestors of 5 but not ancestors of 2.

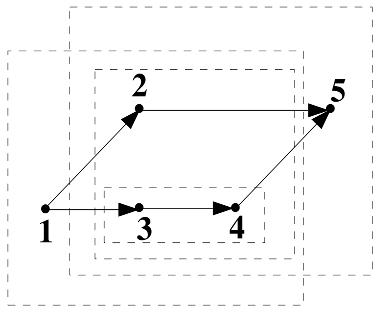

Figure 1
Clans

We can classify a graph as (i) *primitive* if the only clans in G are ∅, G and singleton sets; (ii) *independent* if every subset is a clan; or (iii) *linear* if for every pair of nodes x and y in the graph, x is an ancestor or descendant of y.

Independent graphs are sets of isolated nodes. Figure 2 is an example of a primitive graph. Clans in a linear graph are sequences of one or more nodes $v_i, v_{i+1}, \ldots, v_{j-1}, v_j$ where for k<i, v_i is an ancestor of v_k..

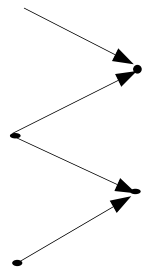

Figure 2
A Primitive Clan

2.3 Graph-grammars

String grammars are a special case of graph-grammars. A string is isomorphic to a linear graph, and in a production of the string grammar, the replacement string is connected to the host string in the same way as the replaced string. For example, if the production rule ab → cde, is applied to the string xabc, the string xcdec is produced.

In a sequential graph rewriting system or graph-grammar, graphs are generated from some initial graph by productions where a subgraph of the host graph, the *mother* graph, is replaced by another subgraph, the *daughter* graph. The main problem of graph grammars is specifying how to embed the daughter graph in the host graph, that is, specifying the edges that should be added to connect the daughter graph to the host graph and determining how edges incident to the mother graph should be modified in the derived graph.

For the graph-grammar we define here, the reconnection rule or embedding rule is *hereditary.* An embedding is called *hereditary* if the mother graph consists of a single node and each source node in the daughter graph of a production is connected to the parents of the mother node in the host graph and each sink of the daughter is connected to the children of the mother node. More formally, let the mother graph be node u. For each source vertex v in the daughter graph, (w,v) is an edge in the resultant graph whenever (w,u) is an edge in the host graph, and for each sink vertex y of the daughter graph, (y,w) is an edge in the resultant graph whenever (u,w) is an edge in the host graph. For example if the host graph is the Hasse graph H in Figure 3 and the production maps node x into the graph D, the resulting graph is H'.

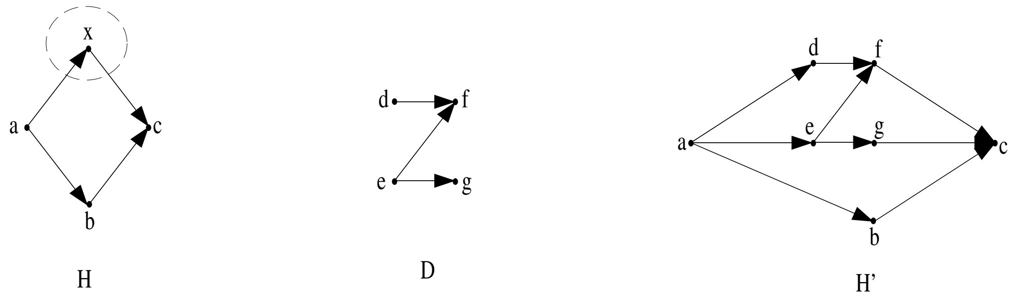

Figure 3
A CGS production

A *clan generation system (CGS)* is a tuple, (z, P, H),where z is a node called the axiom or start node. P is a set of pairs (v,D) where v is a graph vertex and D is a primitive, independent or linear Hasse graph. P represents the set of productions and H is the hereditary rule of reconnection. Let us call a production of this system a *CGS-production.* Applications of CGS-productions preserve the properties of Hasse graphs. Furthermore, the daughter graph becomes a clan in the resultant graph. In a CGS, all graphs resulting from the productions on Hasse graphs are also Hasse graphs.

3 Graph Decomposition Algorithm

Our goal is to write an algorithm to parse or equivalently, find a derivation tree for a given DAG. This involves finding the right hand sides or daughter graphs of the CGS-productions on the corresponding Hasse graph. Since each daughter D is a clan in the derived graph, we will look for clans.

Ehrenfeucht and Rosenberg [2,3,4] prove that every Hasse graph can be decomposed into (built up from) the three types of graphs: independent, linear, and primitive. They further describe a canonical derivation for a given graph that is unique. A derivation is called *canonical* if a CGS-production with linear (independent) daughter graph D is never followed directly by another production where D is linear (independent). This restriction in no way affects the class of graphs that can be derived, for any two consecutive productions of the same form can be combined into a single production. Any Hasse graph yields a unique decomposition when the decomposition is canonical. The purpose of this paper is to describe an algorithm for creating the CGS parse tree of any input DAG.

From now on all graphs are assumed to be Hasse graphs. If a primitive or independent clan were to replace a node using a CGS-production, each source of the clan would then have the same parents as the mother node, and each sink of the clan would then have the same children as the mother node. This leads us to look for nodes with the same parents that will be the sources of the clans and nodes with the same children that will be the sinks of the clans. The entire clan will then include the sources, the sinks, and all the nodes "in between." An edge entering a potential clan at a non-source node, or an edge leaving a potential clan at a non-sink node violates the clan definition. In summary we need to find subgraphs of G with the properties:

A. any ancestor of one source of the subgraph is an ancestor of all elements of the subgraph.
B. any descendent of one sink of the subgraph is a descendant of all elements of the subgraph.
C. all children of non-sink elements of the subgraph must be contained in the subgraph and all parents of non-source elements of the subgraph must be contained in the subgraph.

For any set of nodes S, we will denote the sets of parents, children, ancestors, and descendants of S by P(S), C(S), A(S), and D(S), respectively. To simplify many of the equations needed, let F*(S) denote the set of nodes $F(S) \cup S$.

These properties indicate that we should search for groups of nodes with the same parents and groups of nodes with the same children. The first group of nodes we will call siblings and the second group we will call mates. Since any set of siblings is a potential set of sources for a clan, we partition all nodes into sets S where $x,y \in S$ if and only if the set of parents of x is the same as the set of parents of y (x and y are siblings). Note that $\mathcal{S} = \{S\}$ is a partition of the nodes of the graph. Similarly we partition all nodes into sets M where $x,y \in M$ if and only if the set of children of x is the same as the set of children of y (x and y are mates). $\mathcal{M} = \{M\}$ is a partition of the nodes of the graph. By considering only subgraphs with sources from some $S \in \mathcal{S}$ and sinks from some $M \in \mathcal{M}$ and such that conditions A, B, and C hold, we will find clans. The following theorem gives an alternate characterization of a clan.

Theorem 3.1: Let H be a Hasse graph with $\mathcal{S}$ and $\mathcal{M}$ partitions of the nodes into sets of siblings and sets of mates, respectively. A subgraph of nodes in set G is a clan if and only if

(i) $G = D^*(S) \cap A^*(M)$ for some $S \subseteq S' \in \mathcal{S}$ and $M \subseteq M' \in \mathcal{M}$

(ii) $D^*(S) - (D^*(M) \cup A^*(M)) = \varnothing$
and (iii) $A^*(M) - (D^*(S) \cup A^*(S)) = \varnothing$.

Part (i) corresponds to properties A and B and parts (ii) and (iii) grant the fulfillment of property C. If (ii) is violated, it is said that an *illegal exit* from the clan has occurred. If (iii) is violated, an *illegal entry* into the clan has occurred.

4 Parsing a DAG

4.1 Algorithm Input and Output

Input to the algorithm is a DAG G. The output is a parse tree whose nodes are clans corresponding to those from the canonical derivation. A subset of clans is sought since the total number of clans may be exponential. For example, if G is the independent graph on n nodes, (i.e. the set of n isolated nodes), then every subset of nodes is a clan and there are 2^n clans of G. If G is a linear graph on n nodes, every sequence of k nodes where k < n is a graph, and the total number of nodes is $\Sigma(n-k+1) = O(n^2)$. Since the canonical derivation involves only O(n) clans, an appropriate subset of all clans must be found.

The clans we find will be classified as primitive, independent or linear according to their *quotient* graph. Let F be a clan of G and $\{F_1, F_2, ..., F_n\}$ be a partition of F where each F_i is a clan of G. Then the *quotient graph* of F, denoted by $F/F_1...F_n$ is the graph where the nodes are $F_1...F_n$ and the edges are (F_i,F_j) whenever there exist $x \in F_i$ and $y \in F_j$ such that $(x,y) \in E(F)$. In the parse tree, the children of F are denoted $F_1...F_n$ and $F/F_1...F_n$ is a linear, independent or primitive graph.

4.2 Algorithm Description

The algorithm consists of three distinct parts: the preprocessing section, the clan recognition section, and the parse tree construction section. The preprocessing section finds relevant features of the graph that are used by the decomposition section. It first computes the Hasse graph of G, H(G), and then pre-computes sets of parents, children, ancestors, and descendants for each node. It also partitions the nodes of H(G) into sets of siblings, $\mathcal{S}$, and sets of mates, $\mathcal{M}$.

The deomposition section pairs each sibling set with each mate set to determine if the pair forms the sources and sinks of a clan. For each potential clan, $F = D^*(S_i) \cap A^*(M_j)$., three things are checked: connectivity, clan property C, and prior discovery.

If F contains more than one element, the algorithm first finds the weakly connected components of F. If there is more than one, F is a potential independent clan, and each connected component may itself be a clan. For each potential clan, the algorithm checks for illegal entry or exit in accordance with theorem 1. If F is a clan, it is compared with clans that were discovered previously. If F was discovered previously, it is discarded. If for some clan C, $C \cap F \neq \varnothing$, and neither one is a subset of the other, then the union of F and C must be a linear clan, since they must form some linear sequence. Maximal independent clans are discovered through the clan discovery process, and their connected components give their subclans.

The clan labels of linear or independent are applied as they are discovered, but not all linear clans can be identified until adjacencies between clans are determined. All unidentified clans are labeled as primitives until the parse tree is built. Then improperly labeled linears can be identified.

The parse tree is built by comparing clan size and using set inclusion. The parse tree root is always the entire graph, the leaves are individual nodes, and tree descendants are always subclans of their ancestors.

4.3 The Algorithm

Input: A DAG, G

Output: Parse tree

BEGIN

{Preprocessing}

1. Compute the Hasse graph H(G)
2. For each node $x \in H(G)$ find: the sets of parents P(x), children C(x),ancestorsA(x),and descendants D(x).
3. Partition nodes n_i into sets $S_1, S_2, ..., S_n$ where $n_i, n_j \in S_k$ if and only if $P(n_j) = P(n_i)$. $\mathcal{S} = \{S_1, S_2, ..., S_n\}$
4. Partition nodes n_i into sets $M_1, M_2, ..., M_m$ where $n_i, n_j \in M_k$ if and only if $C(n_i) = C(n_j)$. $\mathcal{M} = \{M_1, M_2, ..., M_m\}$

{Find clans}

/*** two lists are used: the list of unique clans and the candidate list which holds clans until they can be put on clan list or discarded as duplicates**/

For each pair S_i, M_j where $S_i \in \mathcal{S}$ and $M_j \in \mathcal{M}$ do

Begin {for each pair}

Let $F = D^*(S_i) \cap A^*(M_j)$.

If F contains more than one element then

1. For each connected component F_k of F with source S_k and sink M_k check for illegal entry or illegal exit using formulas (ii) and (iii) of section 3.1.
2. Place each connected component with more than one element on candidate list
3. If F has more than one legal connected component then
   label the union of legal connected components as Independent and place on candidate list
4. For each candidate F do
   begin {for each candidate}
   a. remove F from candidate list
   b. For each F_i on clan list.
      begin {for each clan on clan list}
      if $F = F_i$ then
         if F is labeled then F_i is given the label of F
         exit {clan list loop}, continue with next candidate
      else if $F \cap F_i \neq \emptyset$ and not ($F \subseteq F_i$ or $F_i \subseteq F$) then
         delete F_i from clan list
         $F := F \cup F_i$
         label F as linear
      end {if}
   end {for each clan on clan list}
   c. Place F on clan list
   d. If F is not labeled, label F as primitive
   end {for each candidate}

END {for each pair}

{Build parse tree }

Root := largest clan from the clan list

For each remaining clan F in the clan list, from largest to smallest

For smallest tree node, T, for which $F \subseteq T$,

1. parent(F) := T
2. child(T,k) := F where children are placed in topological ordering.

{Add tree leaves}
For each tree node N
add leaf nodes N - $\cup$ (elements in children of N)

{Correctly label linears}
For each primitive node
label as linear if for each successive pair of children C_i, C_{i+1}, $C(C_i) = C_{i+1}$ and $C_i = P(C_{i+1})$

END

Example:

When we apply the algorithm to the graph of Figure 4, the resulting clans and their classification are

C1 = {a,b,c}I; C2= {j,k}I; C3 = {d,e,f,g}P;
C4 = {i,h,j,k}I; C5 = {C1, C3}L;
C6 = {C3, C4}L and C7 = {h,C2}L

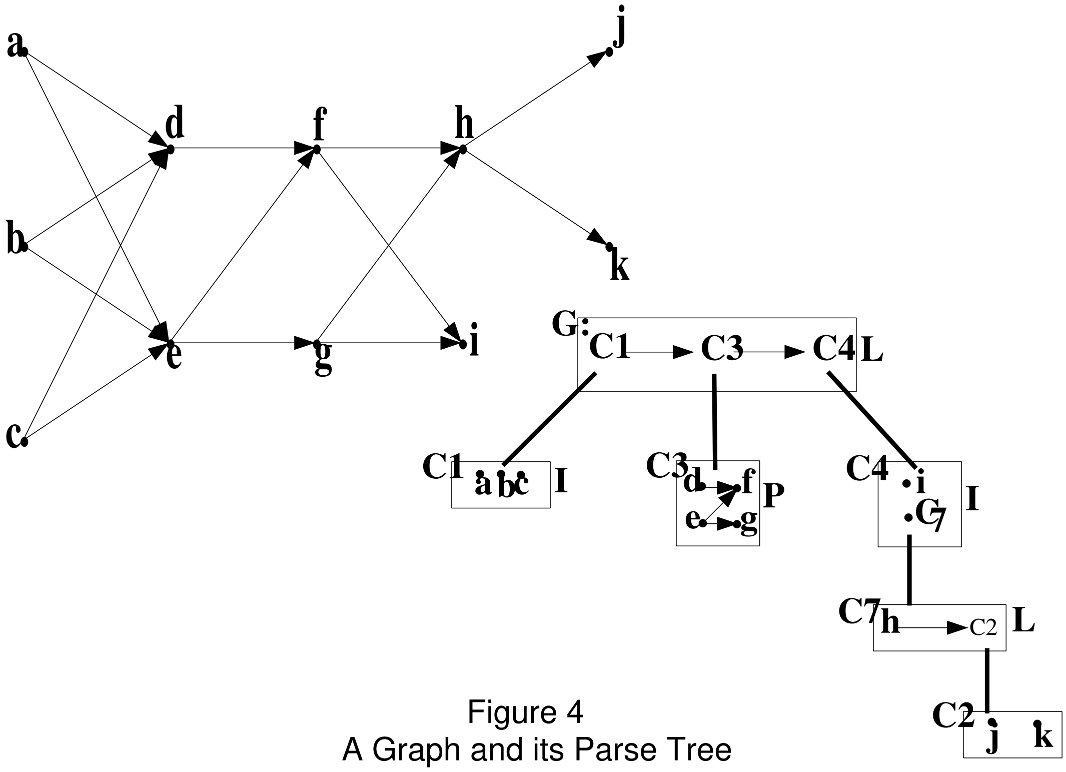

Figure 4
A Graph and its Parse Tree

5. Decomposition of Primitive Clans

Primitives can be arbitrarily large and present a problem when trying to characterize the structure of the underlying graph. In our application of parsing to partition a graph for parallelization, the main concern was to determine code sections that could be parallelized and separate them from sections that had to be executed sequentially. In that application, a program dependence graph represents the data and control dependence found in programs. The nodes represent code sections and the edges indicate an odering or dependence between the nodes. In this application, nodes in a linear clan must be executed sequentially and nodes of independent clans may be executed in parallel. If primitives are not decomposed further, a conservative approach would force the sequential exectuion of all nodes in the primitive clan even though dependences do not require serial ordering. A method is given for subdividing primitive clans that does not violate the dependence requirements.

A primitive clan can be decomposed further by augmenting edges. Specifically, for primitive clan P, add edges from the source nodes S to the union of their children. This creates a linear connection from the source nodes to the rest of the clan. The source nodes then form a *pseudo-independent* clan and the remainder of the primitive, P' = P - S, may be factored further or P' may itself be primitive, in which case the edge augmentation process can be recursively applied.

Figure 5 demonstrates the augmentation process. Adding edges (a,e), (a,f), (b,c), and (b,d) gives a linear clan with components S={a,b} (independent) and P'={c,d,e,f,g,h}. P' is itself independent with subclans {c,e,g} and {d,f,h}. Both subclans are linear clans whose left node is pseudo-independent and whose right node is a singleton.

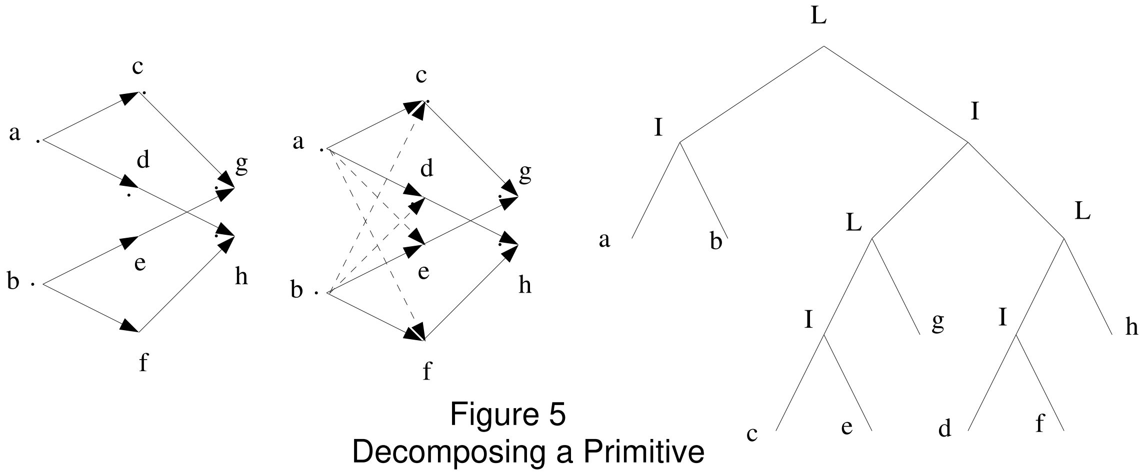

Figure 5
Decomposing a Primitive

This description of the augmenting process is only conceptual, and in fact it is not necessary to add edges in the algorithm. It is a valid process because the augmentation only imposes a stricter ordering on the edges than the original program data flow. Since the addition of edges causes all nodes in the source set to have the same children as well as the same parents, we know the source set is an independent clan. The set of sources is distinguished by marking it as a pseudo-independent clan. When a graph is fully decomposed using the augmentation method, the resulting parse tree of clans is always a bipartite graph where each independent or pseudo-independent node has linear clans as parents and children, and each linear node has parents and children that are independent or pseudo-independent.

In summary, a DAG can now be fully decomposed into a hierarchical structure consisting of independent and linear clans. The hierarchical structure is represented by a parse tree with levels alternating between independent and linear clans. The leaves of the parse tree correspond to the nodes of the original DAG.

6 Timing and the Implementation

The parser has been implemented in C++. Classes have been created for sets, graphs, trees, and clans. The set class implements arbitrary sets as a bit vector, and consequently , set operations on sets with a universe of n elements are accomplished in time O(n/32) using 32 bit words. There is a hierarchy of classes for graphs, from the base directed graph, to DAG, and finally to Hasse graph. The adjacency matrix (or children), parents, descendents, and ancestors for graphs are represented as arrays of sets.

The running time of the algorithm is very dependent upon the structure of the graph. The major loop in finding clans is executed $T = |\mathcal{S}| * |\mathcal{M}|$ times, which can be a great as n^2. When T is large, however, there are more cases where F contains only one element, and processing is done quickly. When T is small, more potential clans are large, and more processing is required. Recursive calls to decompose primitives add complexity to timing analysis.

Actual times have been observed when the program was run at 25 Mhz on a SPARC/IPC workstation . For graphs with no primitives, times ranged from 0.17 sec. for a 15-node graph to 4.732 sec. for a 108-node graph. For graphs with primitives, a 14-node graph was parsed in 0.268 sec. and a 108-node graph was parsed in 9.66 seconds. These and other data points, indicate that the practical running time for the algorithm is $O(n^2)$. More analysis is planned.

7 Significance

The work of the paper exhibits a good (polynomial) parsing algorithm for a very general graph-grammar. Although graph-grammars have existed since the early 1970's, they have not been widely used primarily because no good parsing algorithms have existed for them. Just as the importance of string grammars has come through the discovery of LR parsers, an algorithm such as the one presented here may lead to important uses of graph-grammars.

An application of this work in the area of serial to parallel code translation is being investigated [6] and [7]. In this application, a DAG represents the program dependence graph of a program. Nodes represent executions and edges represent data or control dependencies. Independent clans show where nodes can be executed in parallel and linear clans show where serialization is necessary.

Because of overhead due to interprocessor communication, memory contention, synchronization, process initiation, and other system activities, running all processes concurrently does not necessarily produce the most efficient code. A threshold exists beyond which an increase in the number of processors actually increases parallel time. There is some optimal amount of work that must be done on a single processor for a parallel system to operate most efficiently. The key issue in the creation of an efficient parallel program is that of choosing the proper grain size. Clans are a natural grouping for grains since they are subgraphs whose nodes have identical communication requirements with the rest of the graph.

The parse tree levels represent the degree to which a computation can be made parallel. The leaves represent the highest degree of parallelism or finest grains. Internal nodes represent the aggregation of their children into coarser grains. Each level from tree bottom to top represents larger grains and lower overhead costs. By applying system metrics to the tree nodes, a proper balance between a high degree of parallel execution and low overhead can be found.

Bibliography

1. A. V. Aho, R. Garey, and J. D. Ullman, Transitive Reduction of a Directed Graph, *SIAM J. Comput.* **1** (1972), 131-137.

2. A. Ehrenfeucht and G. Rozenberg, Theory of 2-Structures, Part I: Clans, Basic Subclasses, and Morphisms, *Theoretical Computer Science 70* (1990), 277-303.

3. A. Ehrenfeucht and G. Rozenberg, Theory of 2-Structures, Part II: Representation Through Labeled Tree Families, *Theoretical Computer Science 70* (1990), 305 -342.

4. A. Ehrenfeucht and G. Rozenberg, Primitivity is Hereditary for 2-Structures, *Theoretical Computer Science 70* (1990), 343-358.

5. C. Mccreary, "An Algorithm for Parsing a Graph-Grammar," Ph.D. Dissertation, University of Colorado, Boulder, 1987.

6. C. McCreary and H. Gill, Automatic Determination of Grain Size for Efficient Parallel Processing, *Communications of the ACM*, **32** (1989).

7. C. McCreary and H. Gill, Efficient Exploitation of Concurrency Using Graph Decomposition, *Proceedings of the International Conference on Parallel Processing* , (1990).

8. D. M. Moyles and G. L. Thompson, Finding a Minimum Equivalent of a Digraph, *Journal of the ACM*, **16** (1969), 455-460.