Partitioning for Parallelization Using Graph Parsing

C. L. McCreary

Department of Computer Science and Engineering

Auburn University, Al 36849

e-mail: mccreary@eng.auburn.edu

(205) 844-6307

List of Figures

Partitioning for Parallelization Using Graph Parsing

C. L. McCreary*

Department of Computer Science and Engineering

Auburn University

Introduction

Programming parallel systems for efficient execution is a difficult challenge that currently consumes a great deal of effort and time. Current programming practices and tools are almost entirely architecture specific, and the efficiency of a program is primarily dependent on how compatible the program is with the hardware. The software investment is wasted when transition to a new architecture or configuration is required. Programming tools are needed to reduce the labor-intensive task of programming and re-programming parallel systems.

This paper presents a graph-theoretic scheme for partitioning and scheduling a computation for efficient parallel execution. The technique parses a graph into a hierarchy of subgraphs known as clans that can be identified as able or unable to support parallel execution. The technique is known as clan-based graph decomposition and will be referred to as CGD. Unlike standard graph partitioning techniques, CGD considers the structure of the entire graph and bases parallelization decisions on this structure. Other partitioning and scheduling techniques generally focus on only a section of the graph like the critical path or a group of tightly connected neighbors to find local optimizations.

*This research is sponsored by NSF grant no. CCR-8909253

1. Exploiting Parallelism: Automatic vs. Manual

The problem of developing programs for parallel systems has been approached from two opposite directions. One solution is the programmer-based approach where environments and languages are being devised to aid the programmer in creating his own parallel program. New programs are written in extensions of existing languages or in special parallel languages. The programmer creates parallel algorithms that are specialized to suit the hardware configuration and cannot easily be ported to other parallel systems. Much of the effort of this approach is placed in tuning the program to get good speed-up on only one system.

At the opposite end of the spectrum is the use of automatic tools to detect potential parallelism and extract it from existing sequential code. If the conversion can create efficient parallel code, the effort required to develop such systems will yield great payoffs. The advantages in terms of labor savings are obvious. The greatest drawback is that the conversion is dependent on the existing algorithm. Human insight to specify alternate formulations of the algorithm that offer greater opportunities for parallelism is not available.

Probably the best solutions lie in a middle ground where parallel programmers interact with tools to develop parallel programs. The tools can pinpoint data dependencies and query the user about their removal. They could also store information about the characteristics of the hardware that must be known in order to develop efficient programs. Temporal and spatial concerns could be removed from the user and their responsibility placed with the tools. This paper presents a solution to developing one of those tools: the automatic partitioning tool.

2. Tasks for Automatic Parallelization of Serial Code

When automatically converting a sequential program to code suitable for execution on a multiprocessor system, there are four non-trivial problems to be solved:

- the identification, through program dependence analysis, of code fragments which can be executed in parallel;
- the elimination of non-essential dependences to maximize the number of parallelizable code fragments;
- the aggregation of such fragments into processes that optimize overall performance (in terms of both communication and processor time);
- the assignment of processes to processors (scheduling).

The focus of the paper is on a method to accomplish the last two tasks: partitioning and scheduling.

Partitioning

Speedup is the ratio of the running time of a program on a single processing unit to the running time on a parallel computer. The ideal speedup for a parallel computation on a multiprocessor with *n* nodes is *n* (linear speedup). Linear speedup is virtually impossible to obtain, however, due to two problems: (1) limited available parallelism in the computation and (2) overhead. When the result of one statement is required for another statement, the statements must be executed sequentially, and a dependency is said to exist between the statements. Overhead costs are incurred through inter-processor communication, memory contention, synchronization, scheduling, and other system activities.

As higher degrees of parallelism are introduced into a computation, the expectation is that the total processing time will be decreased. Empirical tests have shown that a threshold exists beyond which an increase in the number of processors actually increases the processing time required. The problem is that two conflicting goals are at work: a high degree of parallelism and low communication costs. On the one hand, increasing the number of processors allows more instructions to be executed in the same amount of time, but when more processes are working on the same computation, additional communication is required between the processes. This key issue in the creation of an efficient parallel program is that of choosing the proper grain size. The *granularity of a parallel program* is defined to be the average size of a sequential unit of computation in the program, with no inter-processor synchronizations or communications [22]. *Partitioning* establishes the granularity of a computation.

The optimal granularity is dependent on architectural characteristics. Systems with a few powerful processors and distributed systems with expensive inter-processor communication will require a coarser grain than systems with many, but less powerful processors with fast inter-processor communication. The goal of partitioning is to minimize the elapsed time from the beginning of the program execution to the end. The partitioning tool facilitates this process by determining the optimal grain size for the algorithm as applied to a particular architecture.

3. Clan-based Graph Decomposition

Many partitioning techniques have been proposed, but most have been based on heuristics that make partitioning decisions with information only from some subgraph. Clan-based graph decomposition determines the structure of the entire underlying dependency graph before attempting to formulate the appropriate grain. Its foundations are from the work on 2-structures by Ehrenfeucht and Rosenberg[5,6,7] and is related to the work of Buer and Mohring[4]. This section explains the technique.

Program Dependence Graph

A *program dependence graph* (PDG) is a directed graph where the vertices represent blocks of statements and the edges represent essential control or data dependence[8]. The *strongly connect component graph* or *condensation graph* of the PDG can be used to represent the dependences in the program. In this interpretation, a single graph vertex could model a program loop, and the component corresponds to a strongly connected component of the PDG. Loops can also be unrolled and graph vertices can represent a single or multiple loop iterations. Edge weights will be assigned to estimate the magnitude of communication between graph vertices and vertex weights will estimate the execution requirements of the block. The intention is to leave the interpretation of the PDG flexible so that the partitioning technique can be applied in a variety of settings. The only requirement of the decomposition is that the input be a directed acyclic graph (DAG). *Clan-based graph decomposition (CGD)* is a parse of the PDG into a hierarchy of subgraphs.

Clans

The central focus of the partitioning scheme is the *clan*. Let G be a DAG. 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 vertices 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 3.1, sets {2,3}, {2,3,4,5,6}, {1,2,3,4,5,6}, and {2,3,4,5,6,7} are the nontrivial clans. C={2,3} is a clan since vertex 1 is an ancestor of each element of C and 5 and 7 are descendants of each element of C. The set {2,3,4} is not a clan since 6 is a descendant of only vertex 4. Clans are known in the literature by many names[20]. Spinrad and Muller use the term *modules* [20], and Buer and Mohring [4] call them *autonomous sets.*

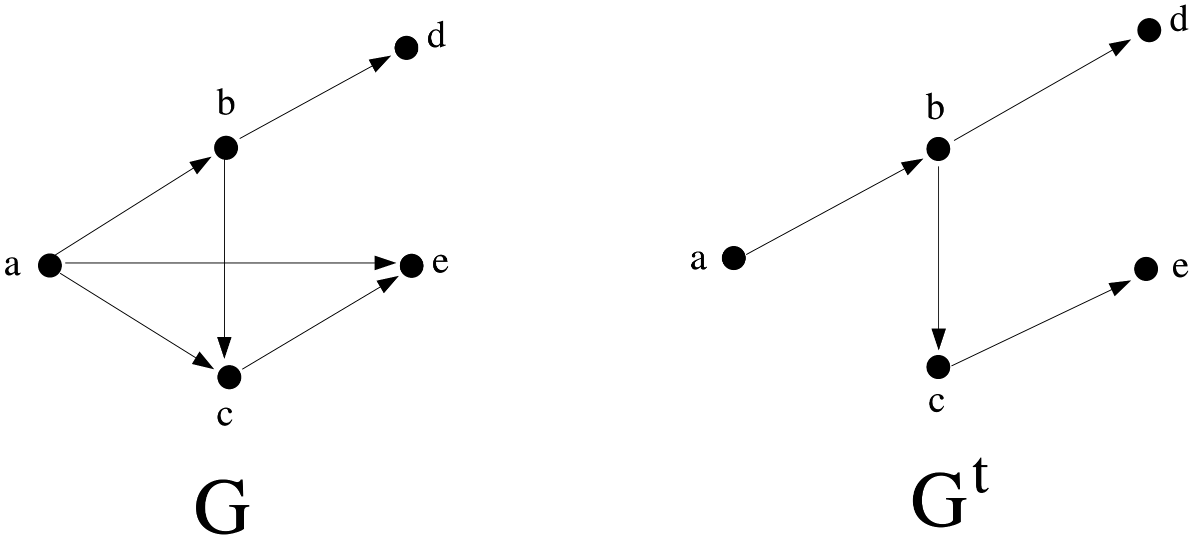


Figure 3.1 A graph and its transitive reduction

A clan *C* is classified as one of three types. It is (i) *primitive* if the only clans in *C* are the trivial clans; (ii) *independent* if every subgraph of *C* is a clan; or (iii) *linear* if for every pair of vertices *x* and *y* in *C*, *x* is an ancestor or descendant of *y*. Independent clans are sets of isolated vertices. Figure 3.2b is an example of a primitive clan. Linear clans are sequences of one or more vertices $v_i,v_{i+1},...,v_{j-1},v_j$ where for $i < k$, v_i is an ancestor of v_k. (See Figure 3.2).

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 a natural way. More specifically, the predecessor of the host becomes the predecessor of the first element of the replacement string and the successor of the host becomes the successor of the last element of the replacement string. For example, if the production rule ab --> cde, is applied to the string xaby, the string xcdey is produced. x becomes the predecessor of c and y becomes the successor of e.

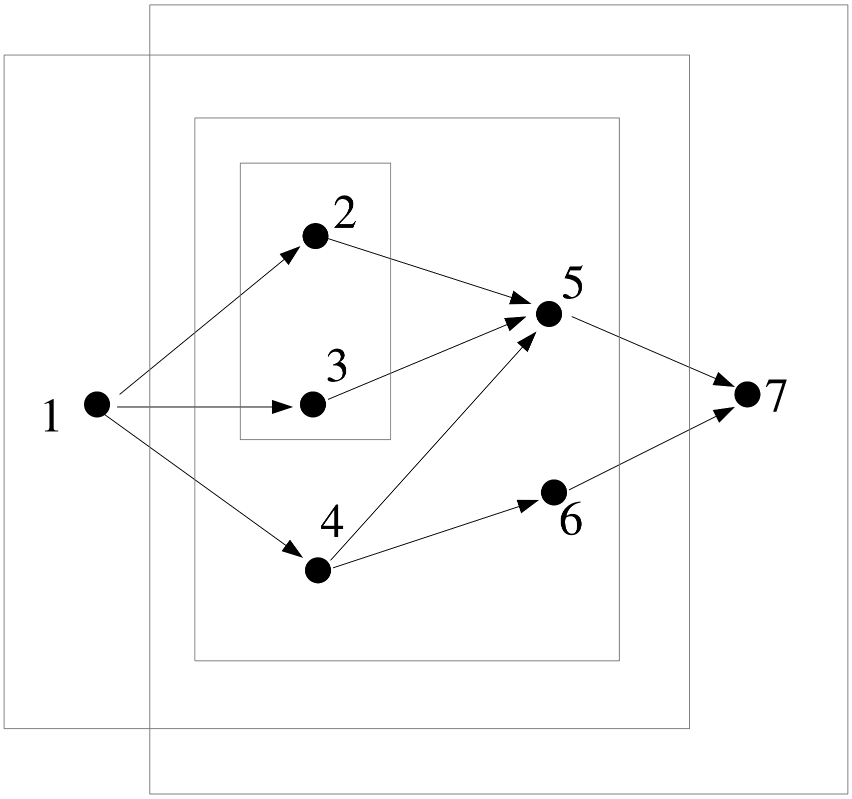
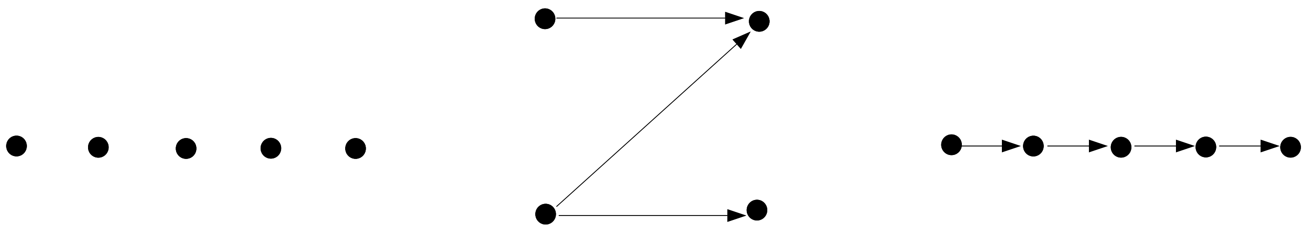


Figure 3.2 Clans

In a sequential graph rewriting system or graph-grammar, graphs are generated from some initial graph by productions where the mother graph, a subgraph of the host graph, is replaced by another graph, the daughter graph. The main problem of graph-grammars is specifying what edges 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. The specification of the edges connecting the daughter graph to the host graph is called the *embedding rule*.

For CGS, the graph-grammar defined here, the reconnection rule or embedding rule is called *heredity*. All host and daughter graphs are Hasse graphs. An embedding is called *hereditary* when the mother graph consists of a single vertex and each vertex in the daughter graph of a production is connected to the host graph in exactly the same way as its mother. More formally, let the mother graph be vertex *u*. For each vertex *v* in the set of source vertices in the daughter graph, (*w,v*) is an edge in the resultant graph whenever (*w,u*) is an edge in the host graph. For sink vertices, *t,* of the daughter graph, (*t,w*) is an edge in the resultant graph whenever (*t,w*) is an edge in the host graph.

A *graph-grammar* is a system $G = (z, P, H)$ where *z* is a vertex called the axiom or start graph, *P* is a set of productions and *H* is an embedding rule. The particular graph-grammar of the work in this paper is called the *clan generation system, CGS*. For CGS, the axiom is a single vertex; *P* is a set of pairs *(v,D)* where *v* is any graph vertex and *D* is a primitive, independent or linear clan; 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. Figure 3.3 shows example productions with the mother vertex identified as vertex *3*.



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

Figure 3.3 Types of clans

Canonical productions

A sequence of productions is called *canonical* if

(i) a production (*v,D*) with linear daughter graph is never followed directly by another production (*d,D'*) where $d \in D$ and D' is linear, and

(ii) a production (*v,D*) with independent daughter graph is never followed directly by another production (*d,D'*) where $d \in D$ and D' is independent.

Note that requiring productions to be canonical does not reduce the set of possible graphs that are generated. Figure 3.4 demonstrates this property. Canonical productions are important because any Hasse graph can be generated by a *unique* sequence of **canonical** productions, and the uniqueness leads to hope of decomposing any Hasse graph into its original components. Figure 3.5 demonstrates three different parse trees for the linear graph with four vertices. The first corresponds to the sequence of productions: (v, x->y), (x,a->b), (y, c->d). The second comes from the sequence: (v, x->d), (x,y->c), (y,a->b). The third which is the canonical productions is the single production, (v, a->b->c->d) The canonical requirement guarantees that the parse must be that shown in Figure 3.5(c).

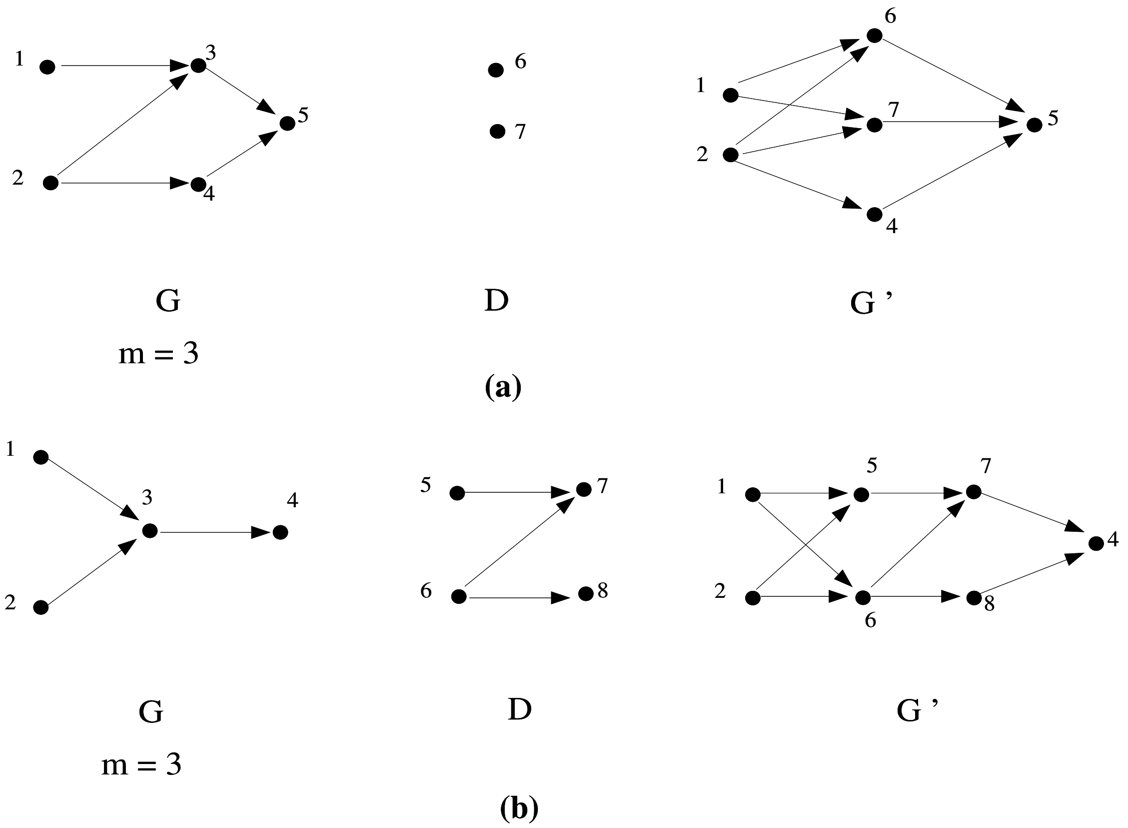
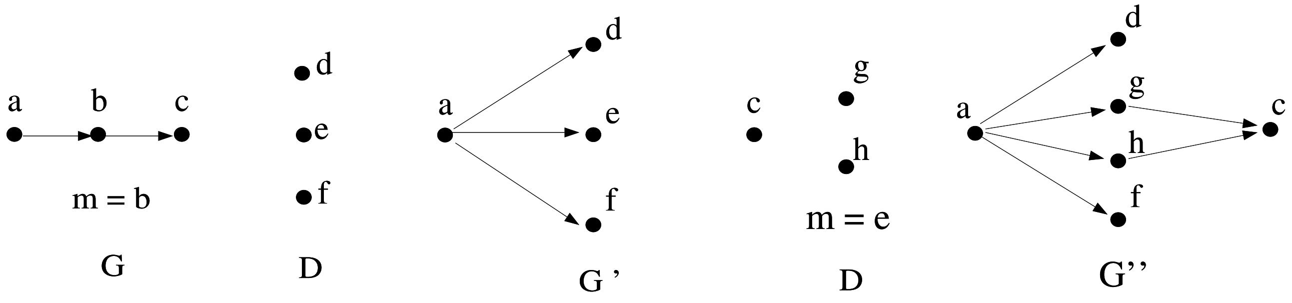
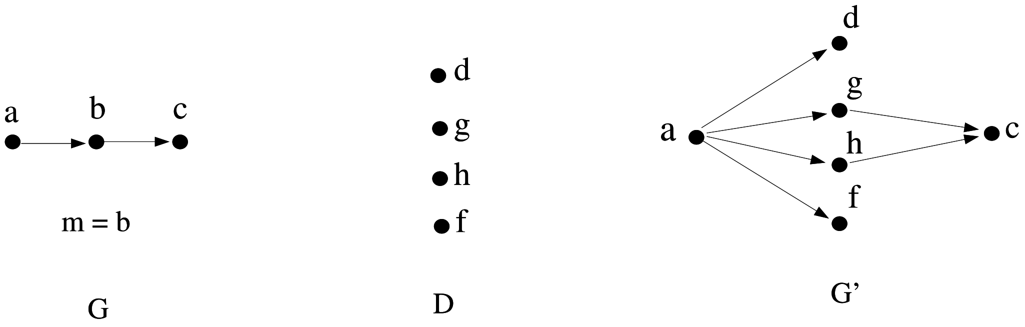


Figure 3.4 Production examples



(a) Two adjacent independent productions



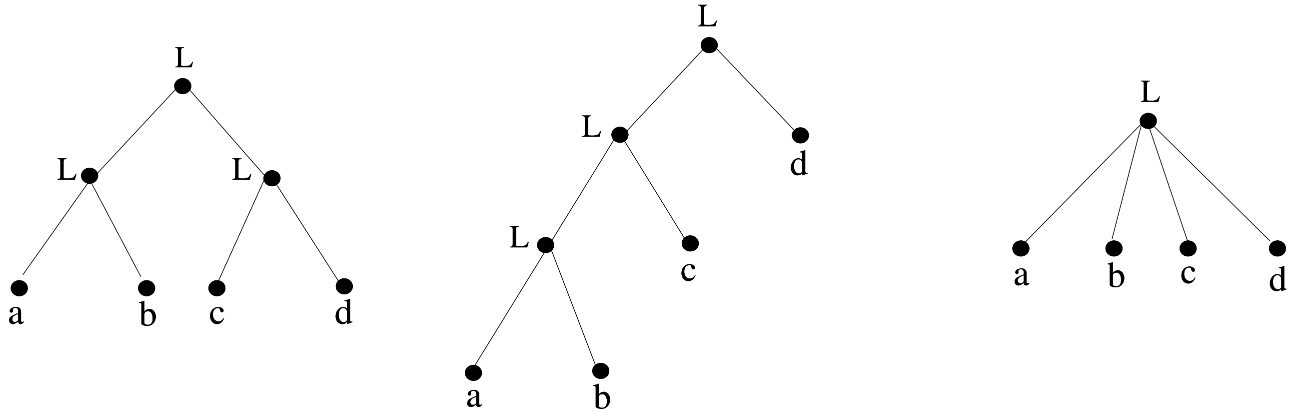
(b) Canonical production

Figure 3.5 Canonical productions

Quotient graph

The concept of a *quotient graph* is important for CGS because it permits the classification of complex clans into the categories of linear, primitive or independent. Let C be a clan and with $\{C_1,C_2,..C_k\}$ a partition of C where each C_i is a clan of C. The *quotient graph* of C denoted $C/C_1..C_k$, is the graph with vertices $C_1,C_2,.C_k$. Pair(C_i,C_j) is an edge of $C/C_1..C_k$ iff $x \in C_i$, $y \in C_j$, and (x,y) is an edge of C. In Figure 3.6, the clans that partition C are $C_1 = \{a\}$, $C_2 = \{b,c,d\}$, $C_3 = \{e\}$, and the quotient graph is linear. The clans that partition of Figure 3.7 are {a,b}, {c}, {d}, and {e} and the quotient graph is primitive. Every clan can be identified as linear, primitive or independent according to the classification of its quotient graph.

linear graph



(a) (b) (c)

Figure 3.6 Canonical parse

Theorem 3.1 When the original graph is a Hasse graph, the graph generated by CGS is also a Hasse graph.

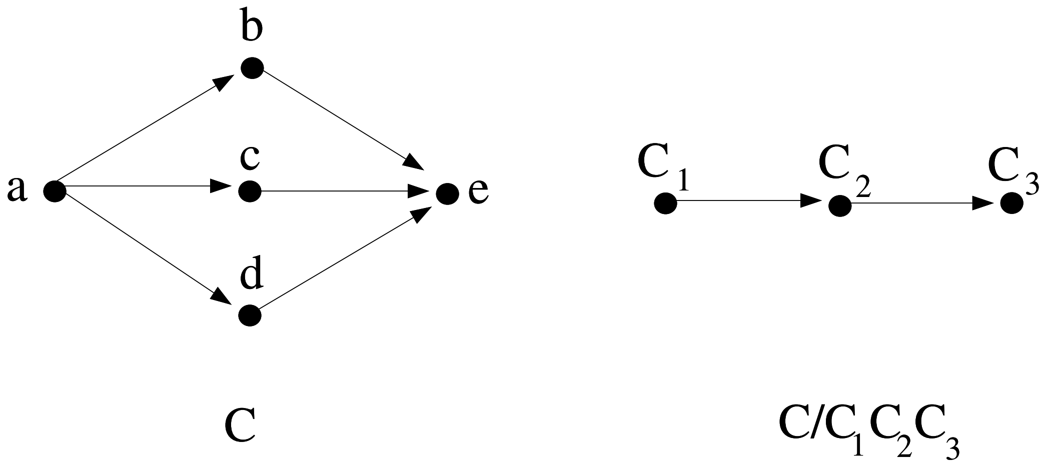
Proof: The proof follows directly as a consequence of the embedding scheme. Since new edges are connected only to sources and sinks of the daughter graph, no short cut edges are added and the graph remains a Hasse graph.

Theorem 3.2 Any Hasse graph can be generated by a unique canonical sequence of productions from CGS.

Proof: Graphs are a special case of 2-structures and this theorem is a special case of theorem 7.11 in [6].

Theorem 3.3 For every Hasse graph there is a unique decomposition into quotient graphs that are identified as linear, primitive or independent.

Proof: See pages 339-340 [6].



C

$C/C_1C_2C_3$

Figure 3.7 Quotient graph - linear

Parsing

Theorem 3.3 guarantees the existence of a unique parse tree from the clans of a graph. The clans that become components of the parse tree are directed subgraphs C with the following properties:

1. Any ancestor of one source element of C is an ancestor of all elements of C.

2 Any descendant of one sink of C is a descendant of all elements of C.

3 All children of non-sink elements of C must be contained in C.

4 All parents of non-source elements of C must be contained in C.

These four properties assure that all elements of C are considered as identical in their dependence relationship to all elements not in C and provides an alternate characterization of the definition of a clan. Properties 1 and 2 suggest that groups of vertices with the same parents and groups of vertices with the same children should be identified, respectively, as potential sources and sinks of clans. Properties 3 and 4 prohibit entry to and exit from the clan at points other than sources or sinks.

Theorem 3.4 Let $\mathcal{S}$ be the partition of the vertices of a graph G into sets S were x,y ∈ S iff the set of parents of x is the same as the set of parents of y and let $\mathcal{M}$ be the partition of vertices of G into sets M where x,y ∈ M iff the set of children of x is the same as the set of children of y. Let D(S) denote the union of S and the set of descendants of S, and let A(S) denote the union of S with the set of ancestors of S. A subgraph C of G is a clan iff

(i) C = D(S') ∩ A(M') for some S' ⊆ S ∈ $\mathcal{S}$ and M' ⊆ M ∈ $\mathcal{M}$.

(ii) D(S') - (D(M') ∪ A(M)) = ∅, and

(iii) A(m') - (D(S') ∪ A(S')) = ∅.

Proof: See [15]

Property (i) defines a subgraph whose source elements, S', have the same set of parents and whose sink elements, M', have the same set of children. If this were not the case, the clan property would be violated. Properties (ii) and (iii) guarantee that there is no illegal entry or exit from the clan. (ii) prohibits cases where there is an edge from a clan element that is not a sink to a non-clan element, and (iii) guarantees that no edge enters the interior of the clan. In Figure 3.8 consider S = {c,d,e} ∈ $\mathcal{S}$, and M= M' = {i,j} ∈ $\mathcal{M}$. S' = {c,d}. The ancestor and descendant sets are:

A(S') = {c,d,a,b}; D(S') = {c,d,f,g,h,i,j,k,l}

A(M') = {a,b,c,d,f,g,h,i,j} D(M') = {i,j,k,l}

The potential clan is C = D(S') ∩ A(M') = {c,d,f,g,h,i,j}

Since properties (ii) and (iii) hold, C is a clan. If the edges (e,g) and (h,m) are added to the graph, properties (ii) and (iii) are violated: m is then in D(S') but not in (D(M') ∪ A(M)) and e is then in A(M') but not (D(S') ∪ A(S')). All clans can be found by computing C from theorem 3.4(i) using all pairs (S, M) of sets from $\mathcal{S}$ and $\mathcal{M}$. This would pair as many as n^2 sets, many of which could not form a clan. To reduce the options, a partial ordering is defined on the elements from $\mathcal{M}$ and $\mathcal{S}$, and only those pairs where elements of S are ancestors of M are considered.

Linear clans can be detected as the concatenation of two or more clans. Two clans, C_1 and C_2, combine to form a linear clan C_3 iff the set of children of the sink of C_1 is the same set as the source of C_2 and the set of parents of the source of C_2 is exactly the sink of C_1. Independent clans have more than one connected component, and each component forms its own subclan. An example graph and parse is shown in Figure 3.9. Internal vertices on the tree are labeled, I for independent, P for primitive, and L for linear.

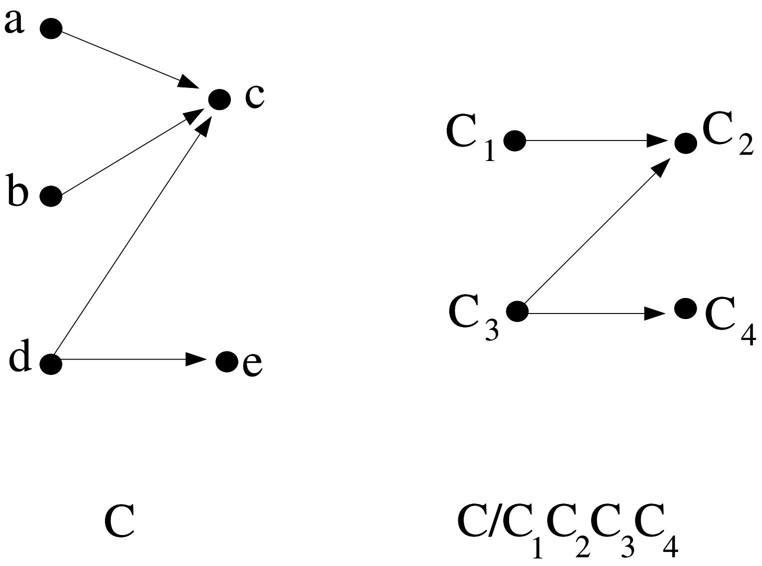


Figure 3.8 Quotient graph - primitive

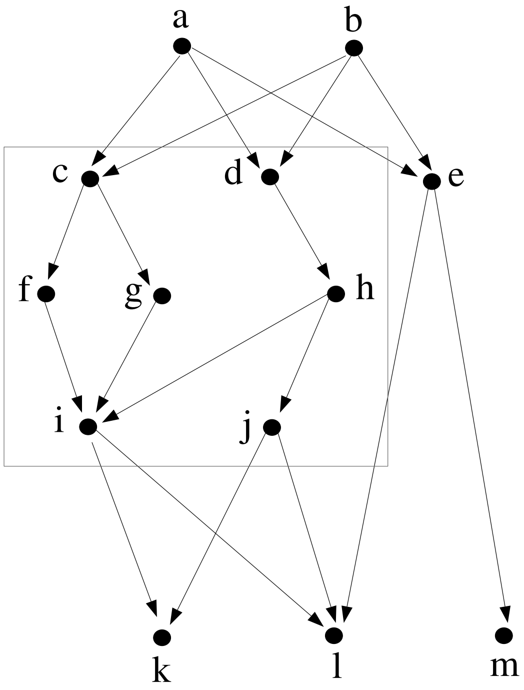


Figure 3.9 Clan characterizations

Decomposing Primitives

When partitioning for parallel execution, primitive clans represent a special challenge. They do not fall into the clear-cut categories of vertices that can be executed simultaneously nor vertices that must execute sequentially. In addition, primitive clans can be arbitrarily large as shown in Figure 3.10. One procedure for further reduction of primitive clans is the addition of edges of weight zero[17]. For a primitive clan P, add edges from the source vertices to the union of their children or from sink vertices to the union of their parents. The addition of edges guarantees that the previous sources (sinks) form an independent clan which is linearly connected to the rest of the clan. This process can be applied recursively. Often, a single edge augmentation can lead to a full decomposition (see Figure 3.11)

The parse tree of any completely decomposed graph is a bipartite tree where the internal vertices are classified as either linear or independent. For a connected graph, the root vertex is always linear. See Figure 3.12.

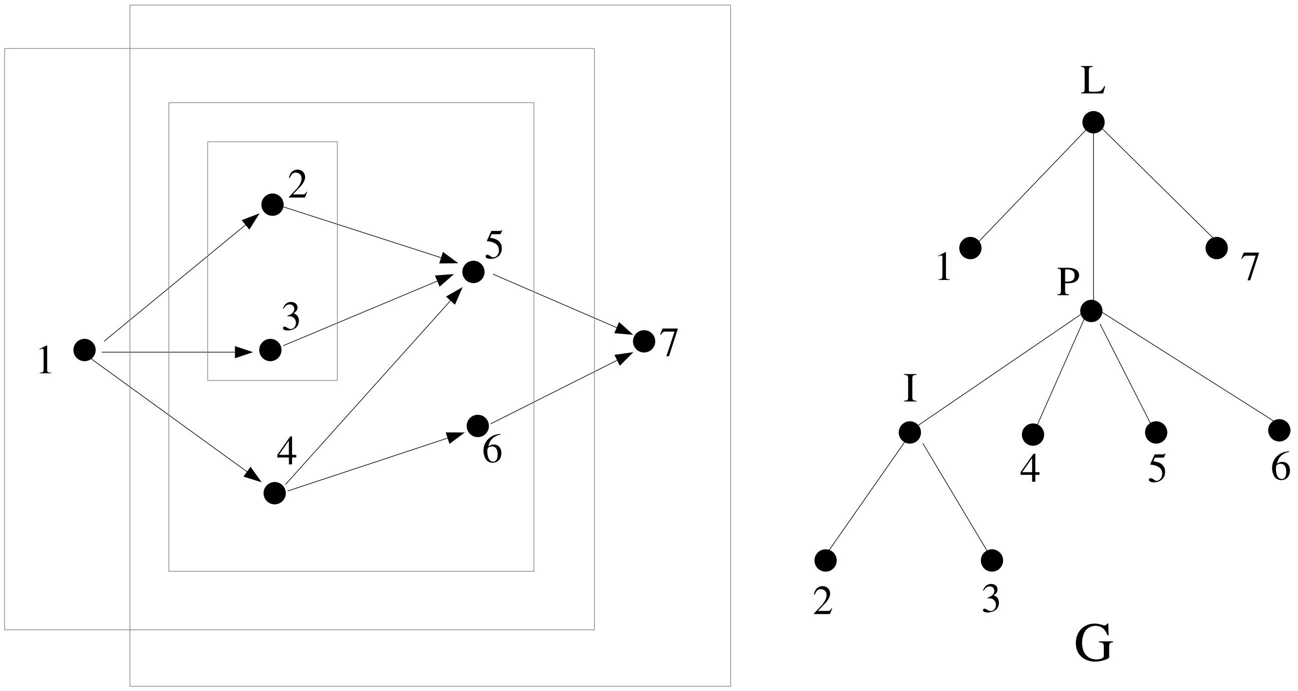


Fig. 3.10 Graph and parsetree

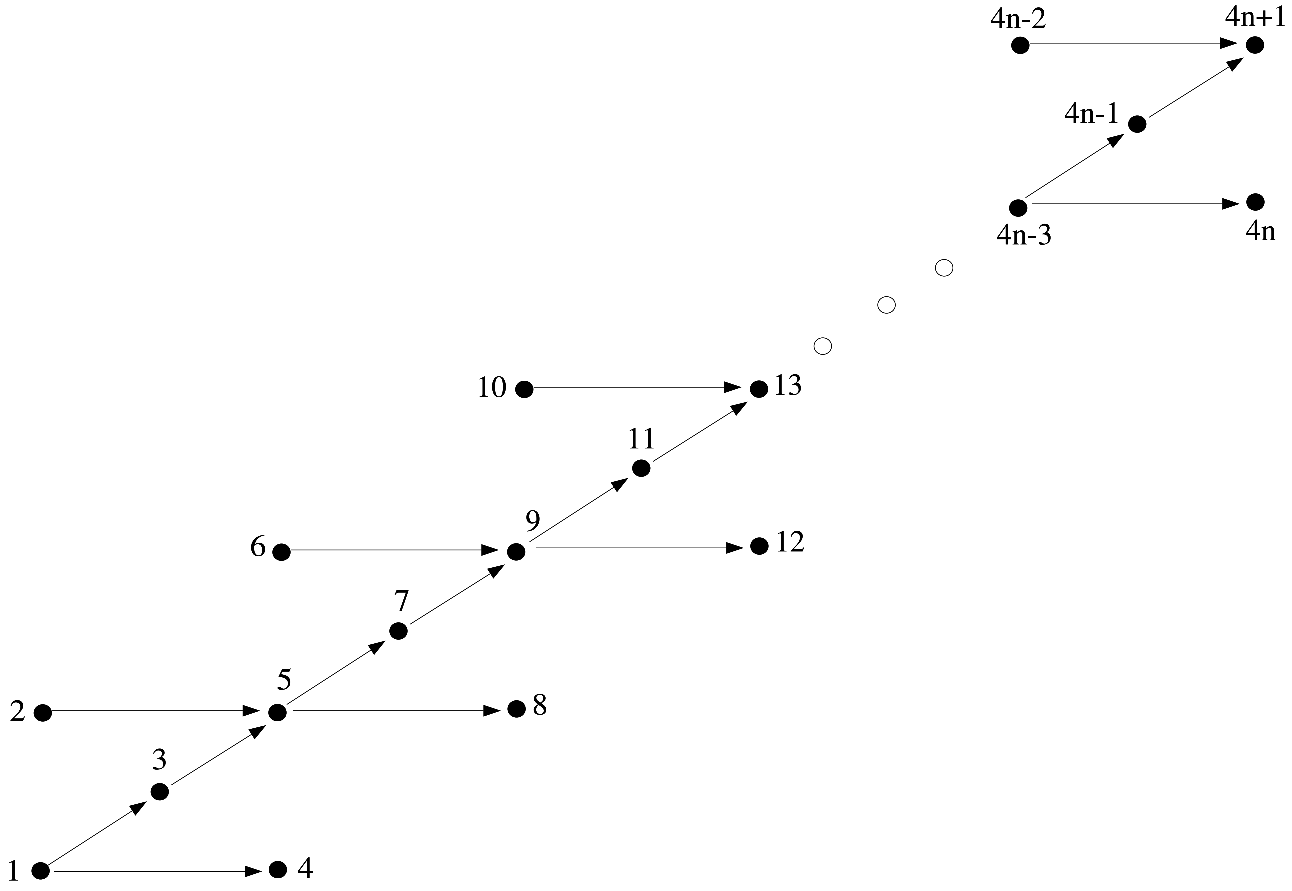
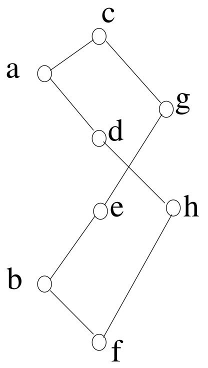
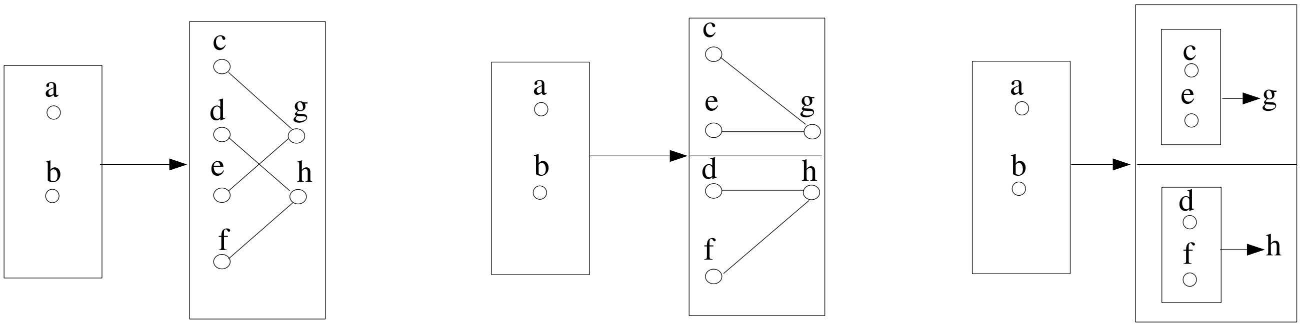


Figure 3.11 An arbitrarily large primitive



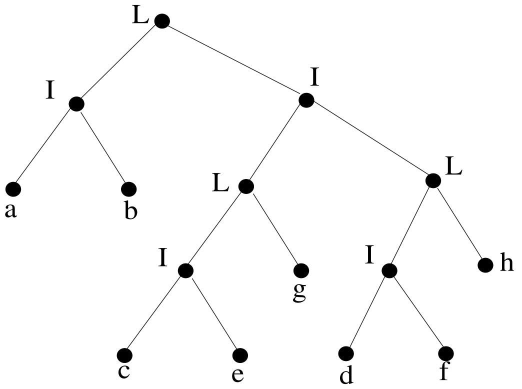
(a) Primitive: augment with zero-weight edges (a,e) (a,f) (b,c) (b,d)



(b) Linear connection

(b) independent

(c) linear



(e) Parse tree of decomposed primitive

Figure 3.12 Decomposing a primitive

4. Graph Decomposition as Means of Partitioning for Parallel Execution

Clan-based graph decomposition has several properties that make it very useful in the parallelization of a computation.

- A clan is a collection of computational elements with identical need to communicate with other clans. Grouping by clans therefore yields a substantial reduction in communication overhead.
- Clans are classified as *linear*, *primitive* or *independent*. The linear clans of a PDG correspond to program segments that must execute sequentially, while the code segments represented by independent clans may execute in parallel. Primitive clans are decomposed further into linear or independent clans.
- The clan structure derived from a PDG is unique, and forms a hierarchy that can be regarded as a parse tree of the PDG.
- The levels of the PDG-parse tree correspond to different degrees of parallelization that can be applied to a particular program. When a cost model for a particular architecture is applied to the PDG-parse tree, the optimal grains can be determined for that architecture.

At each independent clan in the parse tree, a decision must be made as to whether the children will be executed in parallel or aggregated to execute sequentially on a single node.

Cost metrics

No single parallel architecture has yet emerged as the standard. Multiprocessor systems may have a few fast processors such as the Cray vector processors. The system might be an array multiprocessor (SIMD), or a shared memory asynchronous model (MIMD-SM) or a distributed memory system (MIMD-DM). Each of these models has different performance characteristics, different types of inter-processor communication overhead, different scheduling requirements, and different requirements for synchronization. In short, each system requires its own unique model of performance characterization or cost metric.

CGD analyzes the structure of the PDG with no regard for the target architecture and its cost metric. When it is necessary to make partitioning decisions, the cost metric must become a part of the process. The next section will demonstrate the application for cost models in general and will give as an example the decision formulas that correspond to the iPSC system of parallel computers. The iPSC multiprocessor computing system is a distributed memory asynchronous multi-processor system with a hypercube interconnection topology. Associated with each node is a direct connect module (DCM) that handles the inter-processor communication. Although the nodes are configured as a hypercube topology, the DCMs set up virtual links between any pair of nodes and the cost of passing messages between any pair of nodes is essentially the same as passing messages between directly connected neighbors [19]. For this reason, the iPSC hypercube can be viewed as completely connected.

To apply a cost metric to the parse tree, it is necessary to make some simplifying assumptions about the communication and execution requirements of the system: (1) Communication and execution happen concurrently. (2) The total time to execute more than one process on a processor is the sum of the process execution times. (3) Communication times are independent of the communicating node locations. (4) The parallel execution time of processes starting at the same time is the maximum of the individual executions. (5) Increasing the amount of information to be sent from a node does not change the cost of the communication. These simplifying assumptions about the underlying communication metric are not unrealistic for the iPSC systems. The performance measurements in [19] prove that (1) and (3) are true for these systems. (2) and (4) are assumptions generally agreed upon by researchers of parallel systems[3, 11,13]. The last assumption should be modified for more detailed applications of the cost metric. However, it is a very reasonable assumption for the iPSC system, especially when messages are small. The performance measurement work [19] shows that the greatest communication cost is in the start-up overhead. All messages under 100 bytes have the same transmission costs. For messages up to 1000 bytes, fully 50% of the communication cost is overhead. Unless messages are extremely large, the fifth assumption is reasonable.

Application of Cost metric to the Parse Tree

The parse tree shows processes that may execute in parallel as well as those that must execute sequentially. It is the job of partitioning to determine when to use the opportunities to execute in parallel and when to group possible parallel tasks into a single grain. The performance characteristics as described by the cost metric are required to make the partitioning decisions.

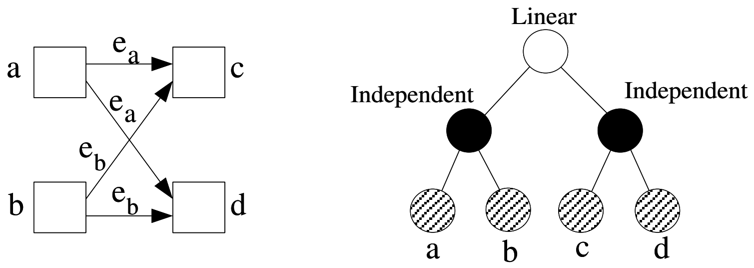


Figure 4.1 Adjacent independent clans

To determine which independent clans should be executed in parallel, inspect the parent (linear) vertex in the parse tree and examine the adjacent children pairwise. Imagine the subgraph of Figure 4.1. By comparing the cost of aggregation with the cost of parallelization, a decision on the method of execution can be made. In the case where the independent clans contain 2 vertices, there are four possibilities for the adjacent children of a linear vertex with seven scheduling choices:

1. both the left and right children can be executed in parallel with

    (a) vertices a and c placed on one processor, b and d on another

    or (b) a and d on the same processor, b and c on another

2 the left child can be executed in parallel and the right child aggregated with

    (a) vertex a placed on the same processor as c and d

    or (b) vertex b on the same processor as c and d

3 the left child can be aggregated and the right child executed in parallel with

    (a) vertex c placed on the same processor as a and b or (b) vertex d on the same processor as c and d.

4 both children can be aggregated and all four vertices a,b,c, and d placed on the same processor.

The seven cases are represented in Figure 4.2. The small squares represent the decision to parallelize and the solid ovals the decision to aggregate. The dashed ovals represent the scheduling choices where all vertices within the oval are to be executed on the same processor. The appropriate metric can be applied to the graph at this point. The example equations in Figure 4.2 are based on the iPSC cost model.

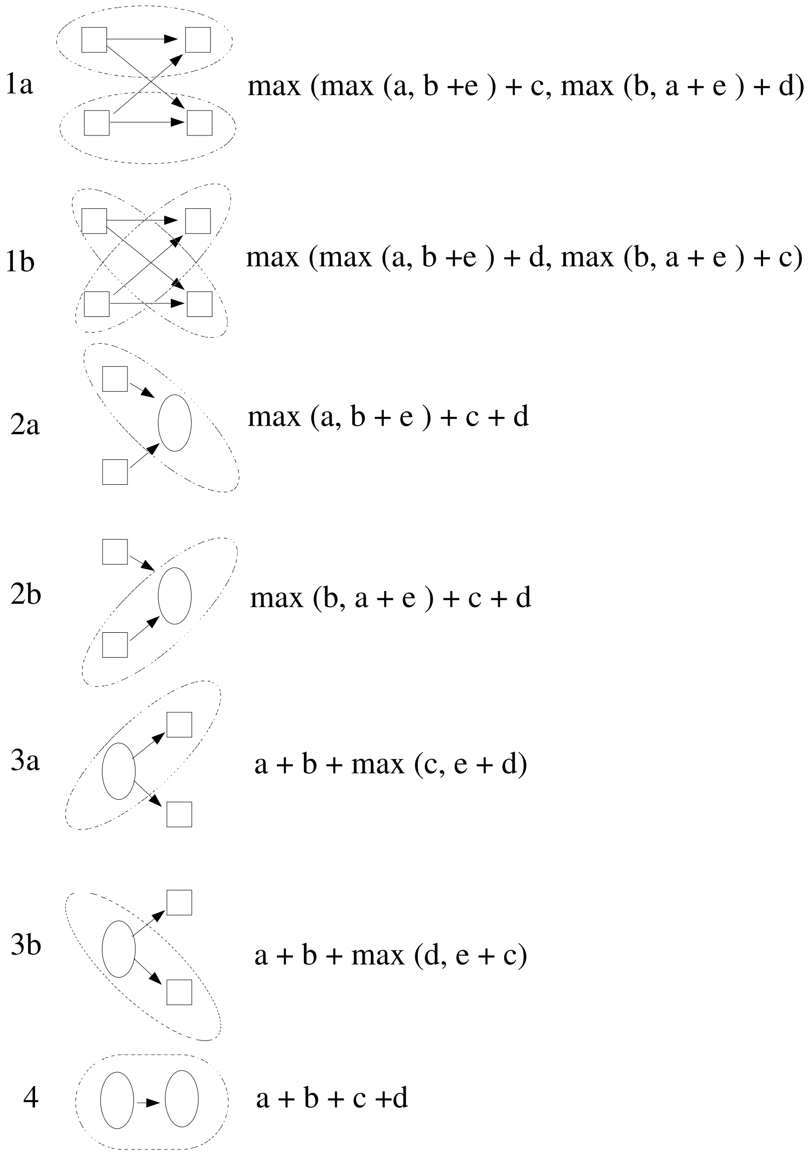


Figure 4.2 Parallel vs. aggregate cost decisions

Let x give the cost of executing PDG vertex x, and let e_x represent cost of the communicating the results of process x to any other process. Under the assumptions, the formulations of the combined execution times of the adjacent independent vertices are given next to the diagrams. In 1a, for example, a and b execute concurrently and pass information before c and d can execute. The processor containing a and c executes a, waits for the results from b, if necessary, and then executes c for a total time of $\max(a, b + e_b) + c$. Similarly the processor containing b and d executes for a total time of $\max(b, a + e_a) + d$. By comparing these seven values the best local decision can be made.

The calculations for the cases can be revised to fit the target architecture. Our formulas are merely **examples** of how to arrive at the proper decisions.

When one of the children of a linear parse tree vertex is a singleton and the other indicates the independence of two vertices, only three choices hold: (i) parallelize the two and place the first on the same processor as the single vertex; (ii) parallelize the two and place the second on the same processor as the single vertex; or (iii) place all three vertices on the same processor. The formulas are similar to the ones for the seven cases.

When either child has more than three vertices, heuristic procedures must be used to determine the proper aggregation/parallelization decision, for considering all cases is unreasonable. Let $c(i)$, be the cost of transmitting the information produced by i, $x(i)$ be the execution time of vertex i, and the $S(i)$ be the sum of i's execution time plus communication time. The heuristic algorithm finds a lower bound on the cost and aggregates processes when those whose aggregations do not increase the overall processing time above the lower bound. First consider the case of the linear vertex with two independent children and the situation where there are many processes in the left child of the linear vertex. Sort the left-child's processes in non-decreasing order by $S(i)$ and say $S(i_1) \geq S(i_2) \geq \ldots \geq S(i_k)$. If there is exactly one right child r, vertex i_1 should be scheduled on the same processor as r. If the sum of all the execution times is no greater than $S(i_1)$, all processes should be aggregated. Otherwise the lower bound for processing the remaining vertices is $L = \max(x(i_1), S(i_2))$ Aggregate the processes in order $i_3 \ldots i_{3+q}$ until the sum of the execution times plus the maximum communication time becomes greater than L. Assign an additional processor for the next vertex group and reapply the aggregation criteria until all vertices are scheduled. Figure 4.3 illustrates this situation. When there is more than one right child and the decision is made to parallelize the right child into more than one group, the lower bound becomes $S(i_1)$. The process can be reformulated for alternate cost models.

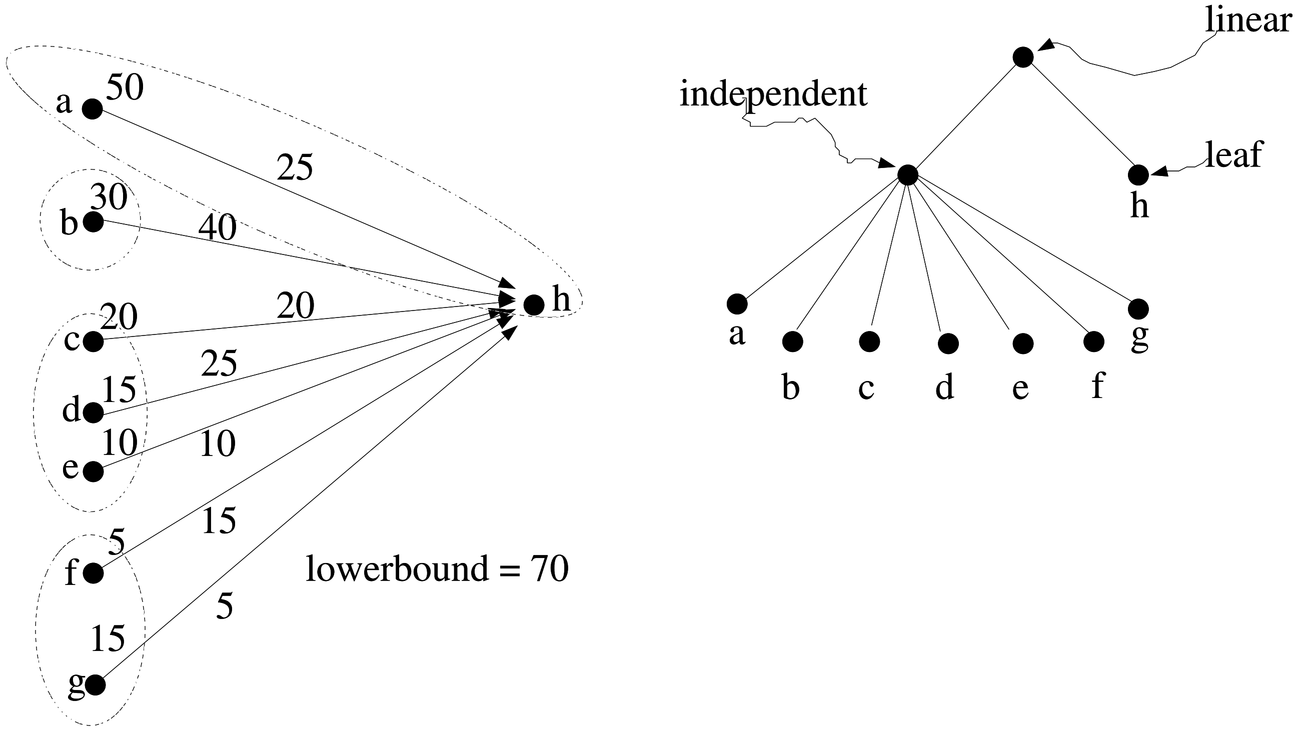
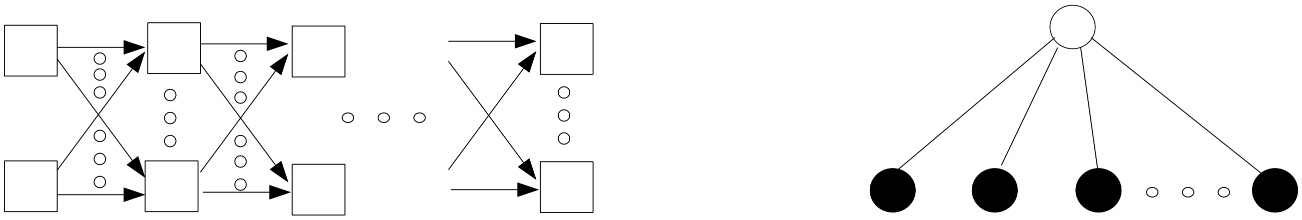


Figure 4.3 Partitioning decision

Linear clans are not limited to the connection of only two sets of independent clans, but can represent the sequential dependencies of any number of clans as illustrated by Figure 4.4. In this case, the aggregation/parallelize decisions on pairwise adjacent clans may lead to conflict. If adjacent decisions agree on the common (middle) clan, the adjacent configurations are referred to as stable. The sequence of decisions is modelled by a multistage graph where the nodes in stage i represent the decision space for stage i and the edge weights from stage i to stage i+1 give the cost of the corresponding decisions. The shortest path through the graph gives the correct decision at each stage [18].

Since the leaves inherit their costs from the program DAG, the internal vertex calculations begin from bottom up by determining the least cost configuration for adjacent pairs of independent vertices. Each decision corresponds to an edge in the parent (linear) clan.



(a) Sequence of independent clans

(b) Parsetree

Figure 4.4 Linear sequence of independent clans

5. Comparison of Partitioning Methods

Many partitioning techniques have been proposed. Some techniques fail to account for communication overhead[3], target only one class of problem[9][25], or minimize communication without regard to execution time[24]. Additional strategies attempt to map the graph of a particular algorithm or set of algorithms onto the physical layout of some architecture with primitive or no communication measurements[2]. Techniques that analyze the PDG usually examine some pertinent subgraph such as the critical path[22,13,21] or neighborhoods of nodes[1.]. None of the schemes of which we are aware analyze the entire graph except clan decomposition. Creating the hierarchy of clans describes the structure of the graph that is necessary for parallelization and provides adequate information for making wise decisions.

A study is currently underway to compare the results of clan-based partitioning with some of the other more promising techniques. The first phase of the study will schedule PDGs from a variety of real programs. The PDGs that have been gathered represent both very regular computations such as FFT and some irregular computations. Preliminary results show clans in a positive light.

Bibliography

1. Baxter, J., and Patel, J., "The LAST Algorithm: A Heuristic-Based Static Task Allocation Algorithm," P*roceedings of the 1989 International Conference on Parallel Processing*, pp. 217-222.

2. Berman, F., "Experience with an Automatic Solution to the Mapping Problem", in *The Characteristics of Parallel Algorithms*, by L. H. Jamieson, D. B. Gannon, and R. J. Douglass, eds.,MIT Press, Cambridge, 1987.

3. Bokari, S., "On the Mapping Problem", *IEEE Transactions on Computers*, March, 1981.

4. Buer, H., and Mohring, R.H., "A Fast Algorithm for hte Decomposition of Graphs and Posets", *Mathematics of Operations Research,* Vol. *, No. 2, May 1983, pp. 170-184.

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

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

7. Ehrenfeucht, A. and Rozenberg, G., "Primiitive is Hereditary for 2-structures," *Theoretical Computer Science* (1990), pp. 343-358.

8. Ferrante, J., Ottenstein, K. J., and Warren, J. D., "The Program Dependence Graph and Its Use in Optimization.", *ACM Transactions on Programming Languages and Systems,* 9: 3 (July 1987), pp. 319-349.

9. Fortes, J. A. B. and Moldovan, D. I., "Parallelism Detection and Transformation Techniques Useful for VLSI Algorithms", *Journal of Parallel and Distributed Computing,* 2 (1985), pp. 277-301.

10. Gerasoulis, A. and Yang, T. On the Granularity and Clustering of Directed Acyclic Task Graphs. *LCSR-TR-153, Dept. of Computer Science, Rutgers University*. Sept. 1990, pp. 1-28.

11. Gibbons, P., "A More Practical PRAM Model", International Computer Science Institute, Berkeley, CA. Technical Report TR-89-019, 1989.

12. Jamieson, L. H., Gannon, D. B. and Douglass, R. J., editors, *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, 1987.

13. Kim, S.J. and Browne, J.C., "A General Approach to Mapping of Parallel Computations Upon Multiprocessor Architectures", *Proceedings of the 1988 International Conference on Parallel Processing*, Vol. III, pp 1-8.

14.. Leiserson, C., and Maggs, B., "Communication Efficient Parallel Graph Algorithms", *Proceedings of the International Conference on Parallel Procesing,* 1986, pp. 861-868.

15. McCreary, C.L., "An Algorithm for Parsing a Graph Grammar", Ph.D. Thesis, Department of Computer Science, Univeristy of Colorado, 1987.

16. McCreary, C. L. and Gill, D. H., "Automatic Determination of Grain Size for Efficient Parallel Processing," *Communications of the ACM* Vol. 32, No. 8, Sept. 1989, 1073-1078. .

17. McCreary, C. L. and Gill, D. H., "Efficient Exploitation of Concurrency Using Graph Decomposition," *Proceedings of the 1990 International Conference on Parallel Processing*

18. McCreary, C. L., Gill, D. H. and Zhu, Y., "Shortest Path Evaluation for Hierarchial Grain

Aggregation", Auburn Univ. Tech Rpt. CSE-93-03.

19. McCreary, C., McArdle, M., and McCreary, J., "Hypercube Communication Performance", Auburn University Dept. of Computer Science and Engineering Technical Report CSE-91-13.

20. Muller, J.H., and Spinrad, J., "Incremental Modular Decomposition", *Journal of the ACM,* Vol. 36, No. 1, Jan. 1989. pp. 1-19.

21. Papadimitriou, C. H. ,and Yannakakis, M., "Towards an Architecture-Independent Analysis of Parallel Algorithms", *Proceedings of the 1988 Symposium on Theory of Computing* , pp. 510-513.

22. Sarkar, V., *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, 1989.

23. Stramm, B. and F. Berman, (October 1990), "Performance Prediction--How Good is Good?," *Proceedings of the Third Symposium on the Frontiers of Massively Parallel Computation*, pp. 43-46.

24. Towsley, D., "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," *IEEE Transactions on Software Engineering*, October 1986,SE-12: 1013-1024.

25. Zhu, Y. and McCreary, C., “Optimal and Near Optimal Tree Scheduling for Parallel Systems”, *Fourth IEEE symposium on Parallel and Distributed Processing.,* Dec. 1992, pp. 112-119.

26. Zima, H., *Supercompilers for Parallel and Vector Computers*, ACM Press, New York, 1990.