Shortest Path Evaluation for Hierarchial Grain Aggregation

C.L. McCreary
Dept. of Comp. Sci. & Eng.
Auburn University

D. H. Gill
The MITRE Corp.
McLean VA

Y. Zhu
Dept. of Comp. Sci
North Dakota State Univ

Abstract

This paper presents techniques for analyzing the data dependence structure of a program to determine an efficient task “grain” structure for parallel execution. A graph parsing technique is used to detect potential parallelism in a directed acyclic graph (DAG). The parse identifies graph components which are linear, independent, and primitive. Linear components must be executed serially, while independent components may be executed in parallel. Primitive components have no specific directives for parallelization or linearization.

The primary contribution of this paper is a new technique for further decomposition of primitive structures. The new technique allows the parse to discover linear and independent components contained inside primitive components. The second contribution is an efficient scheduling algorithm which is systematic and flexible. The algorithm exploits the parallelism in a DAG based on its parse tree by modeling the parallelization/aggregation decisions with a multi-stage decision graph and then finding the shortest path of the decision graph. The enhanced technique is illustrated by its application to the DAG for the Cooley-Tukey Fast Fourier Transform [5].

Index Terms -automatic parallelization, fast fouier transform, grain size, graph decomposition, partitioning, scheduling, shortest path, .

1. Introduction

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 forces are at work. 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 solving the same computation, additional communication is required between the processes. Communication costs between processors play a large role in the execution time of parallel programs. Algorithm degradation can be severe due to communication costs, both in terms of resource utilization and waiting delay. There exists some optimal amount of work that must be done on a single processor for a parallel system to operate most efficiently.

This paper presents a graph decomposition method to support the serial to parallel conversion of algorithms in such a way as to create parallel algorithms that minimize completion time. Our method finds a balance between the conflicting goals of a high degree of concurrency and low communication cost. We define a *grain* to be a set of program steps that is to be executed sequentially on a single processor, and the goal of our work is to determine an automatic method for finding the optimal grain structure. Our objective is to automatically schedule programs for fast execution (i.e., earliest possible completion) rather than to maximize the utilization of processors.

A computation can be broken into components whose condensation graph is a *directed acyclic graph* (DAG). The nodes of the graph represent sequential processes, and a node weight gives process execution time. The edges represent data flows of values required by the processes, and an edge weight gives the cost of transferring data between the two nodes if they are assigned to different processors. Multiple edges emanating from a single node represent duplication of the node's output.

Analysis of the data dependence structure of a program permits the execution grain structure to be determined flexibly. First, the DAG is parsed uniquely, then the cost calculation is applied to the parse tree to determine the grain structure. This allows the program to be reconfigured and moved easily from one execution environment to another. Explicit programming of the concurrency structure of a program is less flexible. Explicit programming implies that a significant change in the architectural characteristics of the execution environment will cause substantial change in the source code defining the parallel task structure.

Determining a grain decomposition that will yield the fastest possible execution is an NP-hard problem and hence, the best we can do in a reasonable amount of time is to develop heuristic methods that will yield good results under most circumstances. Most researches on this topic have been concentrated on finding approximation algorithms based on critical path or list scheduling heuristics [1., 4, 8, 11,12]. For DAGs with edge weights and node weights, a path weight is defined to be the sum of the weights of both nodes and edges on the path. A *critical path* is a path of greatest weight from a source node to a sink node. Extending the critical path method due to Hu \cite{HU61} in classical scheduling, critical path algorithms try to shorten the longest execution path in the DAG by removing communication requirements (zeroing edges) and combining the adjacent tasks into a grain. This approach has received the most attention.

List scheduling heuristics assign priorities to the tasks and schedule them according to a list priority scheme. For example, a high priority might be given to a task with many heavily weighted incident edges or to a task whose neighbors have already been scheduled. Extending the list scheduling heuristic in classical scheduling, these algorithms use greedy heuristics and schedule tasks in a certain order. The graph decomposition method offers a totally different approach to this important problem. It first determines the hierarchical clan structure in the input DAG. Then schedules are derived by balancing the computation and communication costs of the clans using the parse trees. Critical path heuristics and list scheduling heuristics often produce parallel times that **exceed** serial time when communication cost is very high relative to execution costs[7]. The essence of the problem stems from the starting point of these algorithms. Initially, each node is placed on its own separate processor. The algorithms then use their individual heuristics to combine sets of nodes and place them together on processors. Graph decomposition, on the other hand, begins by identifying the parallel constructs available in the DAG and through a metric analysis determines when parallelization is beneficial. Compared with the traditional methods, the graph decomposition method has following desirable properties:

(i) The method offers a systematic and flexible way to exploit parallelism in a DAG.

(ii)The method develops a hierarchy of potential grains and computes execution and communication times on the hierarchy to find the proper balance of aggregation to parallelization.

(iii)The method has a sound theoretical basis. It has been proved to uniquely identify factors that preserve data dependence properties through the theory of boolean algebras as applied to graph grammar theory [2].

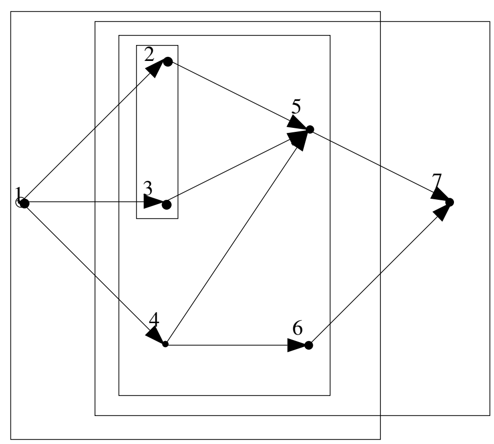
The rest of the paper is organized as follows. In the next section, we will introduce the concept of graph decomposition. In Section 3, we discuss the decomposition of primitive clans. Then in Section 4, we describe the scheduling algorithm based on the parse tree of a DAG. Examples are given in Section 5 to demonstrate the use of the graph decomposition method. Finally we give our conclusions in Section 6.

2. Decomposition of DAGs

The key concepts that distinguish our approach from the traditional ones are the definition of a subgraph known as a clan and a parse of the data flow graph into a hierarchy of clans.

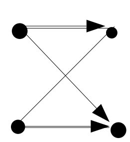
To define a clan, it is helpful to use some standard graph definitions. Let a DAG be denoted by G=(V,E), where V is the set of nodes and E the set of edges. If (u,v) is a graph edge, we say u is a *parent* of v or equivalently, v is a *child* of u. If there is no edge (x,u), then u is a *source* of the graph; and if there is no edge (u,x), then u is a *sink* of the graph. If there is a path from u to v in a DAG, we say u is an *ancestor* of v and v is a *descendant* of u. Associated with each DAG is its *transitive reduction* or *Hasse graph.* The Hasse graph H(G) of a directed graph G is a graph where (1) 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 (2) there is no graph with fewer edges than H(G) that satisfies condition (1). Intuitively, we can describe a Hasse graph as a directed graph with no short cuts. The Hasse graph retains the partial ordering of nodes in the graph while minimizing the number of edges.

A set of nodes X from graph G is a *clan* if and only if 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 if and only if z is a descendant of y. Informally we can describe a clan as a subset of nodes where every element outside the clan has the same ancestral relationship to each member in the clan. That is, a node outside the clan is either an ancestor of all clan nodes, a descendant of all clan nodes, or neither an ancestor nor descendant of any clan node. Trivial clans include G and the singleton sets. In Figure 1, the non-trivial clans include {2, 3}, {2, 3, 4, 5, 6}, {1, 2, 3, 4, 5, 6}, and {2, 3, 4, 5, 6, 7}. The importance of a clan for aggregation is that the sources and sinks of the clan can be seen as identical in their communication with the rest of the graph.



Clans
Figure 1

Independent
Figure 2.



Primitive
Figure 3.

Linear
Figure 4.

The method of determining appropriate grains revolves around the decomposition of a graph into a hierarchy of clans through the implementation of a graph grammar. Graph grammars are similar to string grammars in that nodes (characters) are replaced by graphs (strings of characters). However, the major complication of graph grammars is defining a rule for connecting replacement graphs to the host graph. Every Hasse graph can be generated from a single node by a series of graph grammar productions where the replacement graph is reconnected by the "hereditary" rule. Let us call the node we are replacing the *mother* node, m, and the replacement graph the *daughter* graph. A reconnection is called *hereditary* if the source nodes in the daughter graph are connected to the parents of m in the host graph and the sinks of the daughter graph are connected to the children of m in the host graph. More formally, if (u,m) is an edge in G, then (u,s) is an edge in the derived graph for every s that is a source of the daughter graph; and if (m,v) is an edge in G, then (t,v) is an edge for every t that is a sink of the daughter graph.

The graphs that are used as daughter graphs can be limited to simple graphs which we will classify as *primitive*, *linear*, or *independent.* A graph is primitive if its only clans are trivial; it is independent if it has no edges; and it is linear if for every pair of nodes x and y, either x is an ancestor of y or y is an ancestor of x. In independent graphs, every subset of nodes is a clan. Figure 2 illustrates an independent graph, Figure 3 is an example of a primitive graph, and Figure 4 is a linear graph with clans that can be described as sequences of one or more nodes v_i, v_{i+1} ,..., v_{i+k} where $k>0$.

A derivation is called *canonical* if every production with a linear (independent) daughter graph is never followed directly by another linear (independent) production. The heart of our method parses the graph into clans from which the grains are formed. Ehrenfeucht and Rosenberg [2] prove a theorem that shows the canonical parse of a Hasse graph of any DAG is a unique decomposition of the DAG into clans which can be identified as linear, independent, or primitive. In the parse tree each node represents a clan and the children of a tree node t represents t's subclans. If $t_1, t_2, \dots t_k$ are the children of t in the parse tree, the *quotient* graph of t, denoted by $t/t_1,t_2,\dots t_k$ is defined as the graph with nodes $t_1, t_2, \dots t_k$ and edges (t_i, t_j) whenever there exists $x \in t_i$ and $y \in t_j$ such that (x,y) is an edge of t. The quotient graph is then linear, independent, or primitive, and t is given the label of its quotient graph. If the node is identified as independent, we know the children can be executed concurrently; if it is linear, there is no hope for concurrent execution; and if it is primitive, we will attempt further to identify potential concurrency.

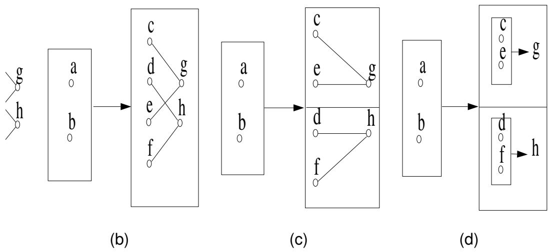
There are several properties of clans that make them attractive as grains for parallel processing. First, all sources of a clan have the same set of parents, so when all clan nodes are combined on a single processor, the communication cost is a fraction of the communication cost accrued when the sources are distributed among several processors. In a similar way all sinks of the clan have the same children so broadcast communications can be aggregated. In addition, the parse tree offers a systematic way to measure the costs of aggregation. The leaves of the tree represent all the original DAG nodes and the root corresponds to the entire DAG. As we go up the parse tree, prudent decisions determine the grouping of the clans into grains of the appropriate size.

Primitive clans represent a special challenge since they do not fall into a category of nodes that can be executed simultaneously nor into a group that requires sequential execution. We will provide a method to further decompose such clans next.

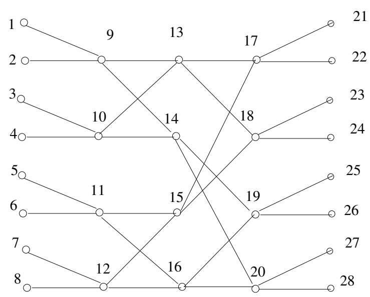
3. Decomposition of Primitive Clans

A primitive clan can be decomposed further by adding 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 and the entire primitive clan is now labeled as linear.. In the augmented clan, the source nodes then form what we call a *pseudo-independent* clan since these nodes are available for simultaneous execution. The remainder of the primitive, P' = P - S, may be factored further or 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.



Decomposition of Primitive
Figure 5



FFT Data Flow Graph
Figure 6

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.

Since augmented edges in pseudo-independent clans carry no weight and will be ignored, pseudo-independent nodes should be distinguished from independent nodes. However, since our treatment of independent clans discussed later also applies to pseudo-independent clans, unless we want to emphasize the difference, we will omit “pseudo-" to simplify our presentation.

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. The leaves of the parse tree correspond to the nodes of the original DAG. Then from the bottom up, an independent (a linear) node has a linear (an independent) node as its parent. In other words, independent and linear nodes appear alternately in a path on the parse tree. The entire decomposition process is illustrated by the following example.

Example: We use the DAG for the Cooley-Tukey formulation of the FFT in Figure 6 [5]. Nodes 1-8 are the initial data, while 9-20 represent partial transforms which are obtained by deriving a recurrence from previous partial transforms. Nodes 21-28 give the final transformation.

Let I, L, and P be the label for independent, linear, and primitive clan, respectively. The non-trivial clans and their labels (I for independent, L for linear and P for primitive) are:

C1: {1,2}I	C2: {3,4}I	C3: {5,6}I	C4: {7,8}I
C5: {1,2,9}L	C6:{3,4,10}L	C7:{5,6,11}L	C8:{7,8,12}L
C9:{1,2,3,4,9,10}I	C10:{5,6,7,8,11,12}I	C11:{21,22}I	C12:{23,24}I
C13:{25,26}I	C14:{27,28}I	C15:{17,21,22}L	C16:{18,23,24}L
C17:{19,25,26}L	C18:{20,27,28}L	C19:{17,18,21,22,23,24}I	
C20:{19,20,25,26,27,27}I	G:{C9,C10,C19,C20,13,14,15,16}P		

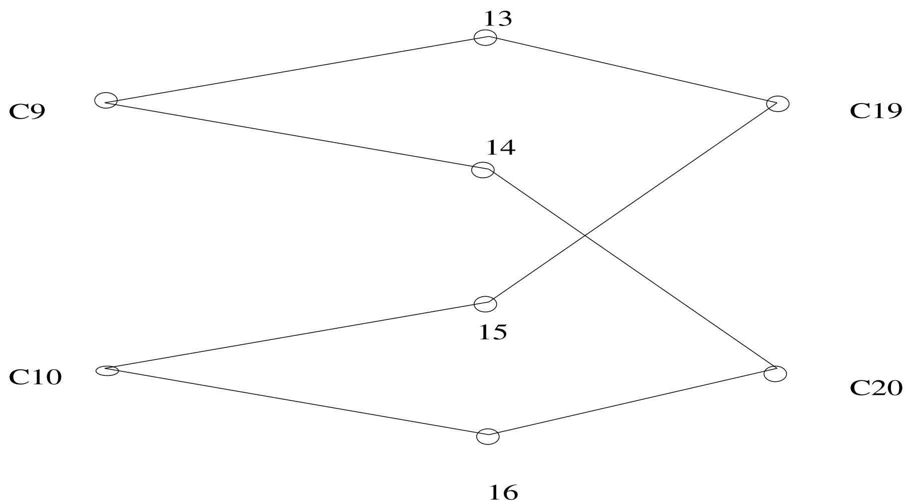
The entire graph is a primitive clan with nodes C9, C10, C19, C20, 13,14,15,16 as shown in Figure 7.

The further decomposition of this primitive clan is the one described by Figure 5. After augmenting with edges (C9,15) (C9,16) (C10,13) and (C10,14), we obtain 7 additional non-trivial clans.

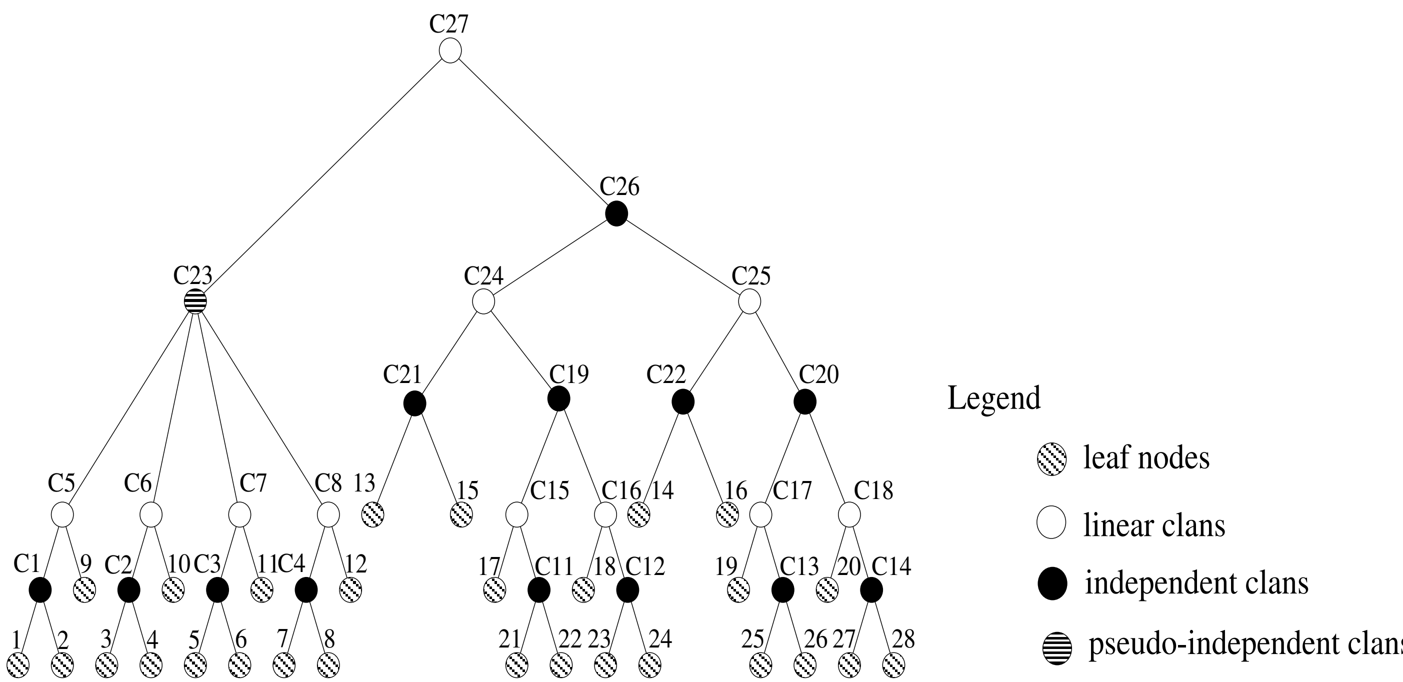
C21:{13,15}I	C22:{14,16}I	C23:{C9,C10}I	C24:{13,15,C19}L
C25:{14,16,C20}L	C26:{13,14,15,16,C19,C20}I		C27:{C23,C26}L

Since any two clans C and C' from a canonical derivation have the property that C∩C'=C or C∩C'=C' or C∩C'=∅, we can form a parse tree with clan C as parent of clan C' if C'⊆C. Figure

8 shows the parse tree for this example. See [] for an algorithm for finding clans.



Reduced FFT
Figure 7



FFT Parse Tree
Figure 8

In the decomposition of primitive clans, the augmentation imposes a stricter order for later execution by the scheduling algorihtm. Other methods for decomposition are being studied by this research group as well as intermediate dependence graph forms that may contain no primitive clans.

4. Scheduling of DAGs

The optimal size of a grain is dependent on the characteristics of the target architecture. Efficient programs on processors whose inter-processor communication costs are high relative to processing costs require larger grains than they would on processors with low inter-processor communication costs. *Scheduling* assigns grains to processors, insuring that all grain inputs are available at the scheduled start time. Associated with each scheduled grain is a processor number, a start time and a completion time (start time plus execution time). Our cost model for scheduling has standard features that have beed assumed by other researchers [8, 11, 12]. The communication cost model assumes:

1. In the DAG representing the PDG, the node weight represents tne computation cost and the edge weight denotes the communications cost. Two tasks scheduled on the same processor incur no communication costs, and any two tasks scheduled on two different processors incur the communication cost specified by the edge weight no matter what the processor assignments.

2 The architecture is a network of an arbitrary number of homogeneous processors.

3 Duplication of the execution of tasks in separate grains is not allowed.

4 Tasks communicate only before starting and after completing their execution. Communication occurs simultaneously with computation and tasks communicate via asynchronous mechanisms whether message passing or shared memory. Tasks may send and receive data in parallel and may multicast messages to more than one receiver.

5 The objective function is to minimize parallel time. Some scheduling heuristics attempt to

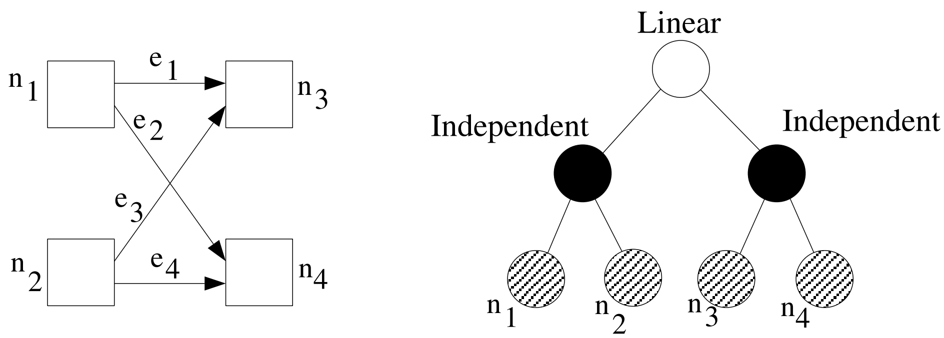
minimize communication time and others, such as load balancing, attempt to keep all processors busy. Since the goal of high performance computing environments is to execute programs quickly, parallel time is the objective that is most meaningful for the scheduling problem.

Our scheduling algorithm is as follows. Decompose a data flow graph into a parse tree. The costs of the leaves of the parse tree are inherited from the DAG, and the costs of the internal nodes are computed from bottom up. A linear clan's children must be executed sequentially. However, an independent clan's children can be either executed as a single grain (aggregated) or executed concurrently (parallelized). Each choice will result in a parallel processing cost that may differ from the other choice. Therefore, at each linear node, we execute its children in such a way that the cost of the linear node can be minimized. The process of determining how to schedule a linear node is repeated until the cost of the root is obtained.

Hence, the scheduling of a DAG is essentially reduced to a determination of a schedule for the children of a linear clan. Next we propose two heuristic algorithms for this purpose. The first one deals with simple linear clans, while the second provides a solution for the general case.

4.1 Simple Linear Clans

A *simple linear clan* is defined as a graph of four nodes as shown in Figure 9(a). This graph can be viewed as a linear graph containing two independent clans, each with two nodes. Figure 9(b) shows the parse tree corresponding to the simple linear clan. Each node in the graph may represent a clan. We consider the case of a simple linear clan first because 1) its simplicity offers an instructive solution, and 2) it appears frequently in parse trees. In Figure 9, n_i represents the cost of node i and e_i represents an edge cost. To simplify calculations, it is assumed that when 2 processes are executed in parallel, the execution time is the maximum of the individual execution times. When they are executed sequentially, the time is the sum of the two individual times. We also assume all outgoing communication from a node occurs simultaneously following process execution. The assumptions may be modified to capture the execution of a specific architecture or generalized by a more abstract formalization.We present the simple model here to illustrate the procedure.



(a) Simple Linear Clan (b) Parse Tree

Figure 9

A decision on the method of execution can be made by comparing the cost of aggregation with the cost of parallelization. Assuming a semi-synchronous model, the latency for each case consists of three parts: executing the nodes in the first stage, communicating the results and executing the nodes in the second stage. There are four possibilities with seven scheduling choices to consider, and their costs can be easily computed:

1. [PP] both the left and right children are executed in parallel, i.e.

    (a) vertices n_1 and n_3 are placed on one processor, n_2 and n_4 on another, or

    (b) n_1 and n_4 on the same processor, n_2 and n_3 on another

2. [PA] the left child is executed in parallel and the right child is aggregated, i.e.

    (a) vertex n_1 is placed on the same processor as n_3 and n_4, or

    (b) vertex n_2 on the same processor as n_3 and n_4

3. [AP] the left child is aggregated and the right child is executed in parallel, i.e.

    (a) vertex n_3 is placed on the same processor as n_1 and n_2, or

    (b) vertex n_4 on the same processor as n_1 and n_2.

4. [AA] both children are aggregated, i.e. all four vertices n_1,n_2,n_3, and n_4 are placed on the same processor.

The formulas for computing the costs are then:

1. (a): $\max(\max(n_1, n_2 + e_3) + n_3, \max(n_2, n_1 + e_2) + n_4)$

   (b): $\max(\max(n_1, n_2 + e_4) + n_4, \max(n_2, n_1 + e_1) + n_3)$

2. (a): $\max(n_1, n_2 + \max(e_3,e_4)) + n_3 + n_4$

   (b): $\max(n_2, n_1 + \max(e_1,e_2) + n_3 + n_4$

3. (a): $n_1 + n_2 + \max(n_3, n_4 + \max(e_2,e_4))$

   (b): $n_1 + n_2 + \max(n_4, n_3 + \max(e_1,e_3))$

4. $n_1 + n_2 + n_3 + n_4$

Using these formulas, the four decisions can be described as:

[PP]: min(1(a), 1(b)),

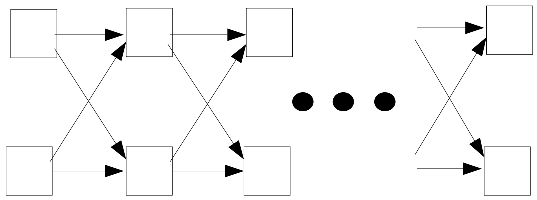
[PA]: min(2(a), 2(b)),

[AP]: min(3(a), 3(b), and

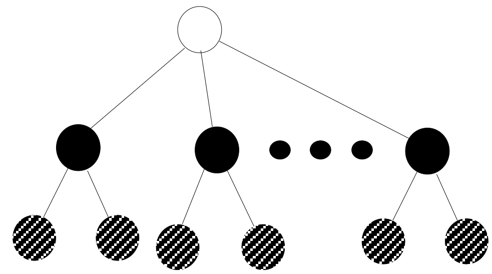
[AA]: 4.

4.2 General Linear Clans

A general linear clan contains m independent clans where each independent clan consists of an arbitraty number of nodes. Figure 10 shows such a linear clan and the corresponding parse tree. Unlike the simple case in previous section, here we have to be able to consider the decisions for all the independent clans together, since any local decision for one independent clan may affect the other decision(s). If decisions are made based only on adjacent pairs of stages, conflicts may arise. For example the decision at stages i - 1 and i may indicate that the nodes in stage i are to be parallelized while the decision at stages i and i + 1 may conclude that the nodes in stage i are to be aggregated. Our solution is to model the clan with a multi-stage decision graph (MSDG), assign weights to edges of the MSDG, and employ a shortest path algorithm on the constructed MSDG.



(a) Multi-Stage Linear Clan

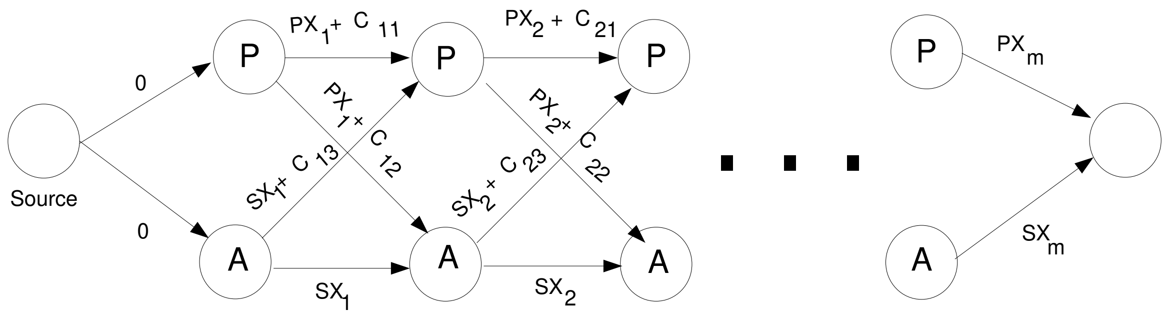


(b) Parse Tree

Figure 10

A Simple Metric

Given a linear clan with m independent clans as shown in Figure 10, we construct an (m+2)-stage decision graph as shown in Figure 11. Let the stages be numbered from 0 to m+1. The MSDG nodes in stage 0 and (m+1) form the source and the sink, respectively. Each stage i between 1 and m has two nodes, with the top one representing the decision to parallelize the execution of the ith independent clan and the bottom node representing the decision to aggregate the ith clan. The four edges between stage i and i+1 correspond to the four decision choices for the simple linear clan in previous section. For edges from stage i to stage i+1, the edge weights represent the cost of execution at stage i plus the cost of communicating the results to stage i+1. In particular, let PX_i represent the parallel execution at stage i, SX_i, the serial execution at stage i and C_{ij} the communication cost from stage i to i+1 along edge j. The corresponding costs are shown in Figure 11.



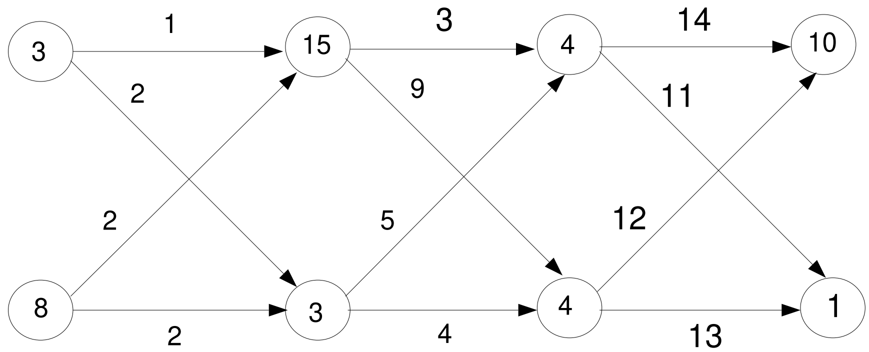
Multi-stage Decision Graph
Figure 11

By using a multi-stage optimization algorithm [6], we can find the shortest path from the source to the sink in the MSDG. This path determines how to schedule the independent clans, i.e., we parallelize (aggregate) an independent clan if the corresponding top (bottom) node is on the path. Also the length of the shortest path represents the total cost of executing the entire linear clan.

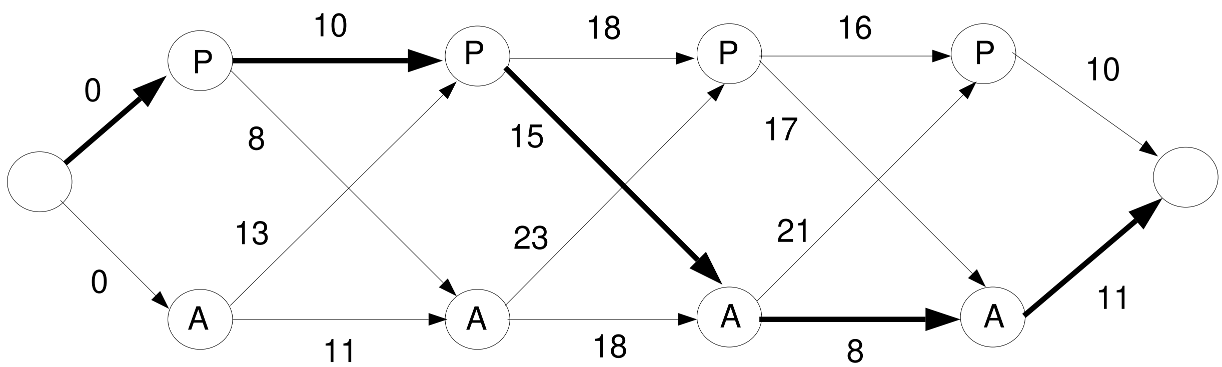
Figure 12 illustrates the use of a MSDG in linear clan scheduling. For the example DAG in Figure 12(a), we build the multi-stage graph of figure 12 (b). The edge costs are similar to the formulas for [PP], [PA], [AP], and [AA] given in section 4.1 except the execution costs, n_3 and n_4, are not included. For the example, the edge [PP] will have cost:

$PX_i+C_i=\min\{\max[\max(n_{i1}, n_{i2}+e_{i3}),\max(n_{i2},n_{i1}+e_{i2})],\max[\max(n_{i1},n_{i2}+e_{i4}), \max(n_{i2}, n_{i1} + e_{i1})]\}$.

According to the shortest path shown in (b), we should parallelize the nodes in stages 1 and 2, and aggregate the nodes in stages 3 and 4. The total execution time for the DAG is 44, as indicated by the length of the bold faced path in (b). If the stages are computed pairwise, the decision at stage 2 is [PP] and the decision at stage 3 is [AA]. The [PP] decision indicates that the stage 3 nodes are to be executed in parallel while the [AA] decision chooses to aggregate the nodes at stage 3. The shortest path in figure 12 (b) resolves the conflict and shows that the stage 3 nodes should be aggregated.



(a) A Linear Clan With Four Independent Clans



(b) Multi-Stage Decision Graph

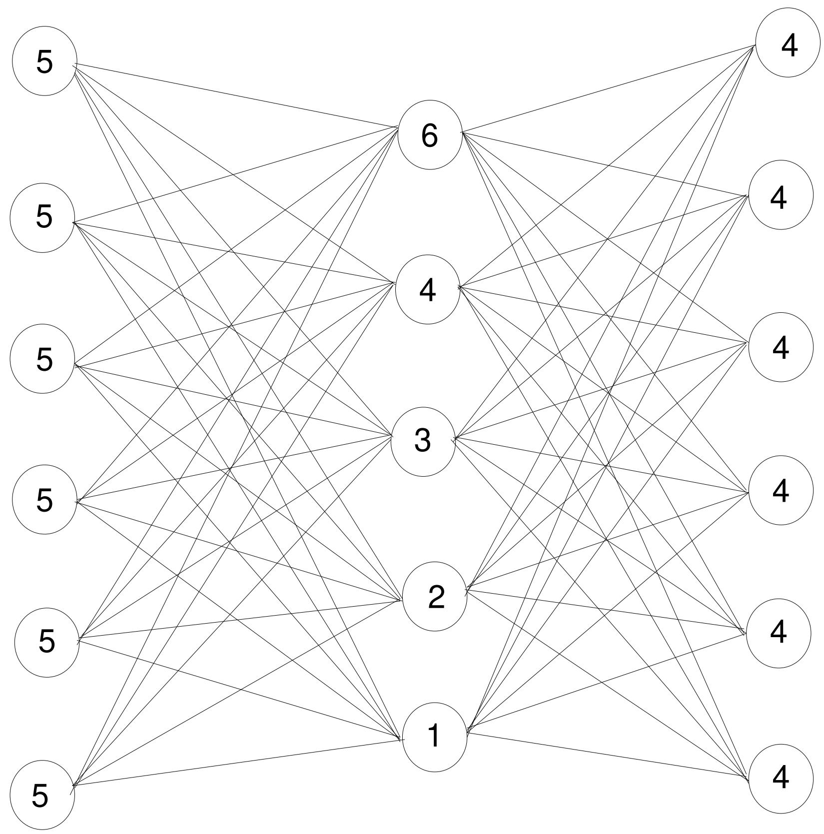
Figure 12

A Complex Metric

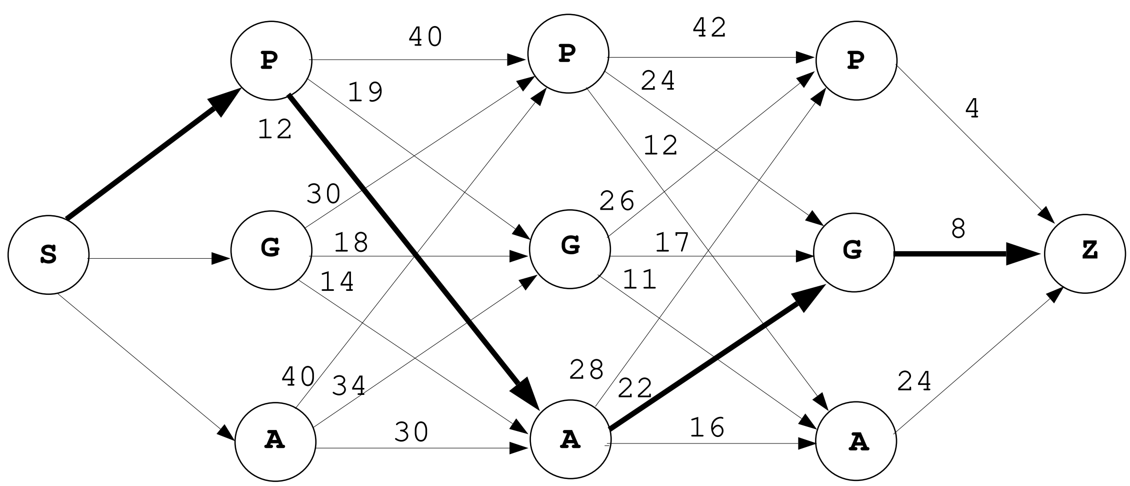
We extend the multi-stage linear clans into general linear clans by allowing each independent node to contain an arbitrary number of nodes. That is, at stage i there are k_i nodes where $k_i > 0$. Now many more choices than two (aggregate or parallelize) are possible. In fact the number of choices at each stage is given by the total number of partitions of the nodes into subsets. A large variety of heuristics could be used to determine how to partition the nodes within any stage. However, no matter what heuristic is used to select the partition, the creation of the corresponding multi-stage graph is identical. Methods for determining edge and node costs must accompany the partition creation. Again the shortest path in the multi-stage graph will give the most efficient solution for the partitions chosen as candidates. Different partitions may give different performance results.

One example heuristic is to choose among three possible partitions: “maximum parallelization”, “total aggregation”, “group to pre-determined grain size”. In the maximum parallelization option, each node within an independent clan will be assigned to a separate processor. Total aggregation means all nodes in the independent clan will be placed on a single processor. To “group to pre-determined grain size”, nodes would be combined until the sum of the node weights is greater than some threshold. This grouping is an instance of the Kth largest subset problem [3]. To model the three partitions and choose the least cost option, create a multi-stage graph with three nodes per stage. Each node represents one of the three possible partitions and the outgoing edge represents the cost of executing the partition and communicating the information with the nodes in the next stage. The shortest path through this graph gives the best schedule.

As an example of this method, consider the graph of Figure13(a) where the node labels represent execution costs. This graph represents a linear clan with three independent clans or three stages. If the grain size is 10, stage I is grouped into three grains, each of size 10; stage II is grouped into two grains of size 8 and stage III produces three grains of size 8.



(a) General Linear Clan



(b) Multi-Stage Decision Graph

Figure 13

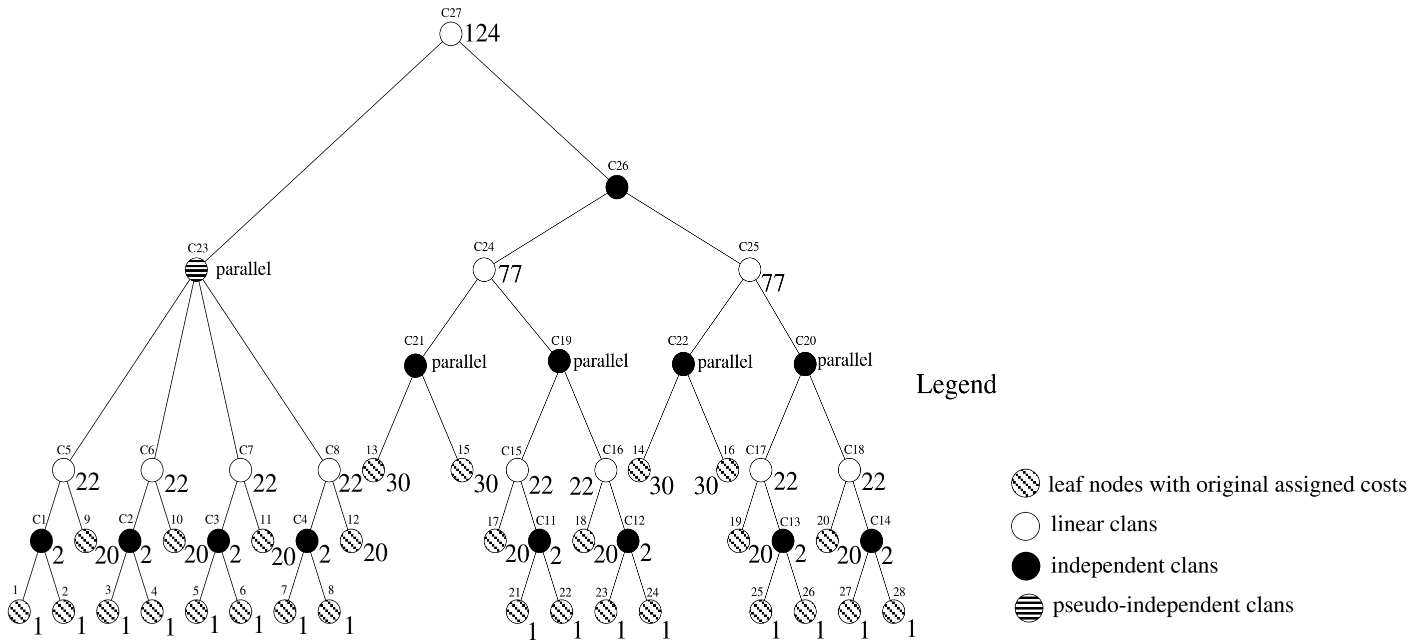
The edge costs are the combined time of executing the nodes in the partition and communicating the information. A specific metric modeling the target architecture is required to give these costs. To illustrate the process here, we choose a model from our work with the Sequent. Communication from a set of p_i nodes to p_{i+1} nodes requires a setup time for each of the p_{i+1} nodes and a message passing cost for each of the p_i*p_{i+1} messages received. This is represented by the formula $c_1*p_{i+1} + c_2*p_i*p_{i+1}$, where c_1 and c_2 are constants. To simplify the calculations let $c_1 = c_2 = 1$. (The exact metric from which the cost is derived is not pertinent to our method, but this metric is used to illustrate the approach.) The total cost of a stage is the execution cost plus the communication cost. We again assume that the cost of parallelization is the maximum of the individual execution costs and the cost of aggregation is the sum of the execution costs. The execution cost of a grouping will be the maximum of the grain sizes where a grain size is the sum of the component nodes costs. The edges of the MSDG are labeled with the sum of the communication and execution cost for each option. For example the [PP] edge from stage 1 to stage 2 is computed as max(stage 1 exection times) + p_2 + p_1*p_2 = 5 + 5 + 6*5 = 40 and the [PG] edge is 5 + 2 + 6*2 = 19. In Figure 13(b), the minimum cost path shows that the processes in stage 1 should be executed in parallel, stage 2 processes should be aggregated and stage 3 processes should be grouped.

5. Examples: Partition and Schedule

To show the flexibility of the scheduling algorithm, we give two examples by using the FFT DAG shown in Figure 6 on two sets of arbitrary values. For this illustration, the metric of section 4.1 is used.

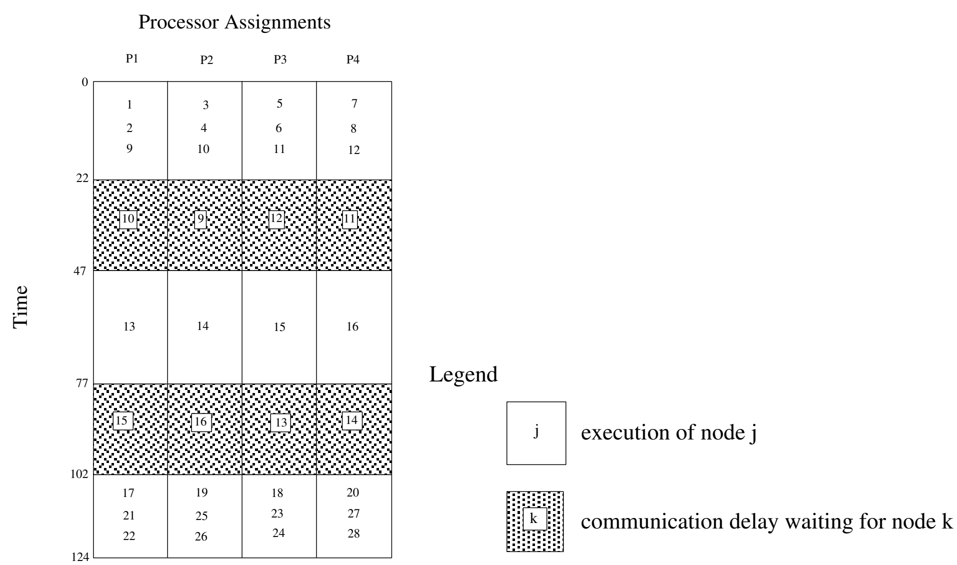
Example I. We assign a cost of 1 to the initial and final nodes; nodes 9--12 are given an execution time of 20; nodes 13--16 are assigned a time 30; and finally, nodes 17--20 are given a time 20. All the edges are assigned a cost of 25. At each linear clan, the relative merits of parallel execution of each (independent) child are weighed against those of the aggregation of the children. At node C24 in Figure 14, for example, we compare the costs of the two possible decisions (since this is a simple linear clan): 1) to parallelize both children and communicate the results of C21 to

C19 is 77 (30+25+22); or 2) to aggregate both children (i.e., execute the entire clan on a single processor) costs 104 (30+30+22+22). Clearly, the better decision is to execute both children in parallel and pay the communication cost. At node C27, the algorithm again determines to parallelize both children and communicate the results of C5--C8 to C24 and C25, resulting a total time of 124. The linear ordering specified by linear nodes C27, C24, and C25 force the execution of C5--C8 to complete before starting any other processes. Then C21 precedes C19, and C22 precedes C20. Four processors (P1--P4) are all that are required for the task. Simultaneously, the four processors will compute C5, C6, C7, and C8. When that work is complete, they will compute nodes 13, 15, 14, and 16, then pass the results to processes C15, C16, C17, and C18. Figure 15 illustrates the complete schedule.



Solution for Data Set I

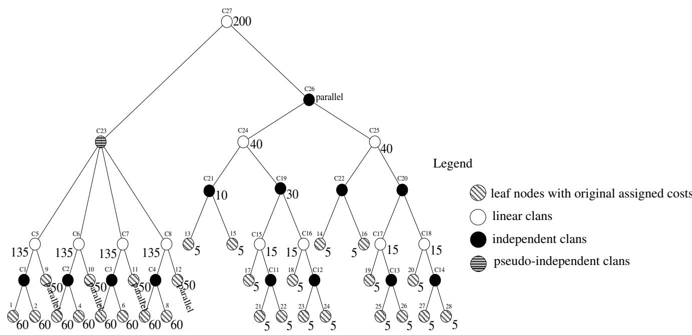
Figure 14



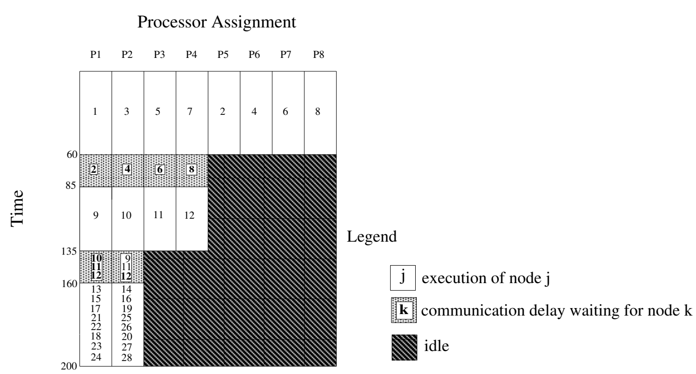
Schedule for Data Set I

Figure 15

Example II. We again assume the communication cost of all edges is 25. Nodes 1--8 have costs of 60, nodes 9--12 costs of 50, and nodes 17--28 costs of 5. Here the decision at node C24 in Figure 16 is to execute all children sequentially on the same processor, because the aggregation cost 40 (5+5+15+15) is less than the parallel cost 45 (5+25+15). The best decision for this data set is to use a single processor to perform the computation of nodes 13, 15, 17, 18, 21, 22, 23, and 24. The schedule is shown in Figure 17.



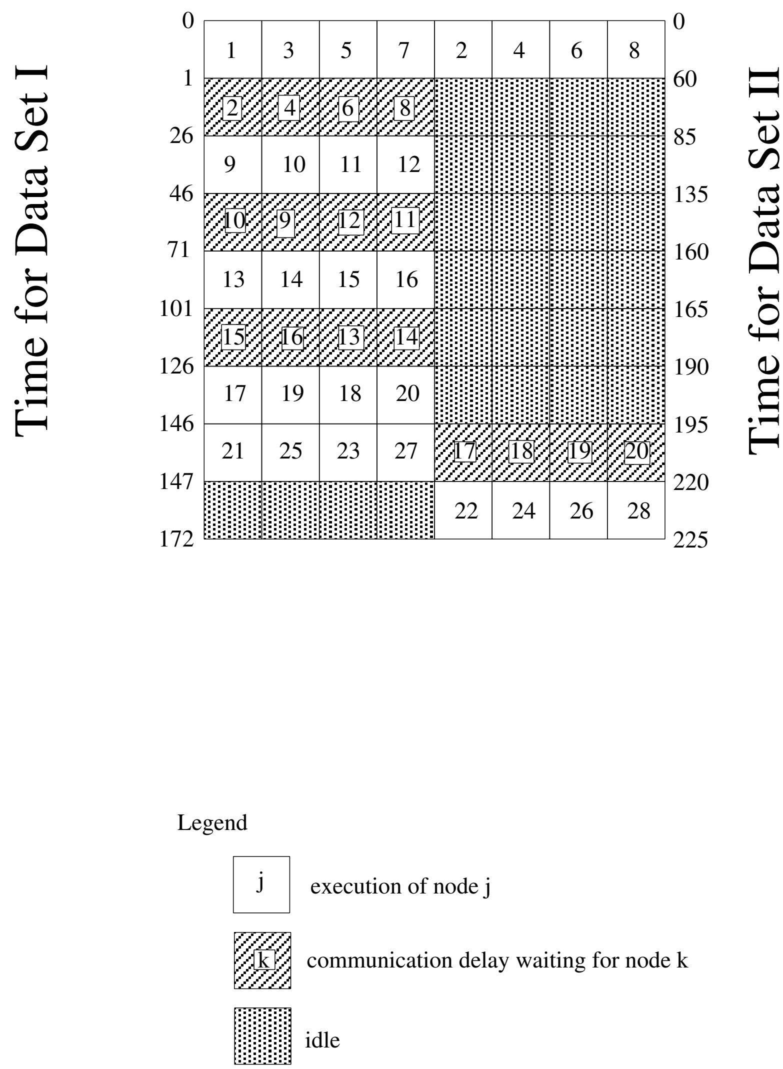
Solution for Data Set II
Figure 16



Schedule for Data Set II
Figure 17

Load balancing is a scheduling method that is often considered for parallelization of tasks. In load balancing, the primary objective is to supply work to any idle processors. Figure 18 shows the schedule generated by a simple load balancing paradigm for our two examples. PDG nodes are scheduled as soon as they are ready for execution. Initially the first 8 processes are started on the 8 available processors. The second stage nodes (9-12) are ready when data is passed from PDG nodes 2, 4, 6, and 8 to nodes 1, 3, 5, and 7 resp. The load balancing method produces total times of 172 and 225 for data sets I and II, respectively, as compared to 124 and 200 by the graph decomposition method.

Load Balancing Schedule



Schedule for Load Balancing
Figure 18

6. Conclusions

We have presented a systematic method for parsing an arbitrary directed acyclic graph representing the data flow of a program to determine potential parallelism in its computation.

Based on sound graph grammar theory, the decomposing method can generate a parse tree which describe the complete hierarchical structure of a graph. The method is more powerful than its predecessor in decomposing primitive factors that were previously not analyzed. Furthermore, we propose an algorithm which can produce an efficient schedule for a DAG by using its parse tree. The FFT example demonstrates that the method is very flexible and can be used to partition non-trivial and practical problems.

We believe that the graph decomposition technique is a powerful tool and has potential applications in many areas which involve analysis of graphs. In fact, the scheduling algorithm developed in this paper can be viewed as one successful application of the graph parsing technique.

For future work, we would like to extend the graph decomposition method to other scheduling models and investigate scheduling algorithms tailored to different parallel architectures.

REFERENCES

1. E.G. Coffman, Jr. (Ed.). *Computer and Job Shop Scheduling Theory*. New York: John Wiley & Sons, 1979.

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

3. M. Garey and D. Johnson, *Computers and Intractability*, New York: W.H. Freeman, 1979.

4. R.L. Graham, Bounds on multiprocessing timing anomalies. *SIAM J.Appl.Math.*, **17, 3** (Mar. 1969), 416--429.

5. R. Hockney and C. Jesshope, *Parallel Computers 2,* Philadelphia: Adam Hilger, 1988.

6. E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms,* Computer Science Press, Rockville, MD, 1984.

7. A. Khan, C. McCreary, and M. Jones, A Comparitons of Multiprocessor Scheduling Heuristics, *1994 International Conference on Parallel Processings.*

8. B. Kruatrachue and T. Lewis, Grain Size Ddetermination for Parallel Processing. *IEEE Software,* **5** (Jan. 1988), 23--33.

9. C. McCreary, An Algorithm for Parsing a Graph-Grammar, Ph.D. dissertation, Department of Computer Science, University of Colorado, Boulder, April, 1987.

10. C. McCreary and H. Gill, Automatic Determination of Grain Sizefor Efficient Parallel Processing, *Communications of the ACM,* **32**, pp. 1073-1078, Sept. 1989.

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

12. T. Yang and A. Gerasoulis, A Fast Static Scheduling Algorithm for DAGs on an Unbounded Number of Processors, *The Proceedings of IEEE Supercomputing 91*, pp.633-642.