Efficient Exploitation of Concurrency
Using Graph Decomposition

Technical Report CSE-90-03

C. McCreary
H. Gill

Department of Computer Science and Engineering
Auburn University
Auburn, AL 36849-5347

July 31, 1990
Efficient Exploitation of Concurrency  
Using Graph Decomposition

C. McCreary  *  
Auburn University  
Auburn, Al.

H. Gill  **  
The MITRE Corporation  
McLean, Va.

Abstract

We present techniques for analyzing the data dependency structure of a program to determine an efficient task "grain" structure for parallel execution. In a previous paper, [McCreary&Gill, 1989], we described the application of a generalized parsing technique that detects potential parallelism in a data flow graph. When combined with cost metrics tailored for particular execution and communication models, this technique permits different decompositions to be discovered for different parallel architectures. In the original decomposition method, the parse identifies components of the graph that are linearly data dependent and components that are independent. (Components that are linearly dependent must be executed serially, and components that are independent may be executed in parallel.) Certain graphs contain irregular data dependencies that are not further decomposed; such components are designated as primitive.

The primary contribution of this paper is a new technique for further decomposing primitive structures. In the earlier work these structures were to be executed serially on the same processor. The new technique allows the parse to discover linear and independent components contained inside primitive components that previously were left as monolithic structures. The enhanced technique is illustrated by its application to the data flow graph for the Cooley-Tukey Fast Fourier Transform, [Hockney&Jesshope, 1988]. A more detailed cost model is also employed.

Introduction

As a higher degreesof 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

* This research is sponsored by NSF grant no. CCR-8909253
** This research is sponsored by The MITRE Corporation
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 is 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 represented abstractly as a labeled (directed) data flow graph. The nodes of the graph represent sequential processes and the node labels give process execution time. The edges represent data flows of values required by the process. The edge labels give the cost of transferring data between nodes. The multiple edges emanating from a single node represent duplication of the node’s output.

Analysis of the data dependency structure of a program permits the execution grain structure to be determined flexibly using a cost model of the architecture on which the program is executed. First, the data flow graph 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 a NP-hard problem and hence, the best we can do in a reasonable amount of time is to develop methods that will yield good results under most circumstances. The graph decomposition method that we propose for partitioning a computation into grains has several desirable properties.

1. The method permits the application of cost metrics that model such characteristics of the target hardware as communication costs and execution costs in determining the grain structure.

2. Graph decomposition determines the appropriate grains for any acyclic data flow graph.

3. The method is flexible enough to be used with systems involving very high communication costs (for example, distributed systems), as well as with a tightly
coupled multiprocessor system.

4. 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.

5. The method has been proved to uniquely identify factors that preserve data dependency properties since it is based on the theory of boolean algebras as applied to graph theory, [Ehrenfeucht & Rosenberg, 1986].

The graph parsing step determines hierarchical clan structure in graphs such as trees and series-parallel graphs used by Towsley [Towsley, 1986] and Bokhari [Bokhari, 1987] as well as data flow graphs of more general structure. Our cost metric is similar to the one used by Sarkar [Sarkar, 1986] in partitioning a data flow graph generated by parallel programming. Sarkar described a comprehensive method for using program execution profiles to determine execution and communication cost statistics for use in a cost metric.

Background: Parsing a dataflow graph

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

To define a clan, it is helpful to use some standard graph definitions. If (u,v) is a graph arc, we say u is a parent of v or equivalently, v is a child of u. If there is no arc (x,u) then u is a source of the graph and if there is no arc (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 of directed graph G is a graph G' where (1) there is a directed path from u to v in G if, and only if, there is a directed path from u to v in G, and (2) there is no graph with fewer arcs than 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 iff for all x,y ∈ X and all z ∈ 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. 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 all clan nodes. 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.

The method of determining appropriate grains revolves around the decomposition of a graph into a hierarchy of clans through the implementation of a graph
Clans

Figure 1.

grammars. 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 a node is replaced by a Hasse graph and 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 arc in G, then (u,s) is an arc in the derived graph for every s that is a source of the daughter graph, and if (m,v) is an arc in G, then (t,v) 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; it is linear if for every pair of nodes x,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_1, v_2, ..., 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 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 \( t_1, t_2, \ldots, t_k \) are the children of \( t \) in the parse tree, define the quotient graph of \( t \), denoted by \( t/t_1t_2\ldots t_k \), as the graph with nodes \( t_1, t_2, \ldots, t_k \) 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, primitive or independent 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 primitive, we will attempt further to identify potential concurrency, and if it is linear, there is no hope for concurrent execution.

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 broadcasted communications can be aggregated. In addition, the parse tree offers a systematic way to measure the costs of aggregation. The root of the tree represents all operations being performed sequentially on a single processor. As we go down the tree opportunities for use of additional processors present themselves. We may accept or reject these opportunities according to the calculated costs of execution and communication.

**Decomposition of primitive clans**

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. If the node is labeled primitive, we augment the edges until we can decompose the primitive graph. 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 itself be primitive, in which case the edge augmentation process can be recursively applied.

This description of the 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. Augmented edges carry no weight and will be ignored. Since their incident edges differ, independent and pseudo-independent nodes must be distinguished.

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.

Example: Graph decomposition

We inspect the use of graph decomposition on the data flow graph for the Cooley-Tukey formulation of the FFT in figure 6 (see Hockney and Jesshop pg. 499.) 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.
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. Decompose this primitive clan further by augmenting with edges (C9,15) (C9,16) (C10,13) and (C10,14).
The result adds 7 additional non-trivial clans.

\[ C_{21} = \{13, 15\} \]

\[ C_{22} = \{14, 16\} \]

\[ C_{23} = \{C_9, C_{10}\} \]

\[ C_{24} = \{13, 15, C_{19}\} \]

\[ C_{25} = \{14, 16, C_{20}\} \]

\[ C_{26} = \{13, 14, 15, 16, C_{19}, C_{20}\} \]

\[ C_{27} = \{C_{23}, C_{26}\} \]

Since any two clans C and C' 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. To achieve maximal parallelization, a parse tree node labeled pseudo-independent will never be allowed to have independent children. The independent children will be combined to form a larger independent node. Figure 8 shows the parse tree for this example.

![Parse Tree Diagram](image_url)

**Legend**
- ◯ leaf nodes
- ○ linear clans
- ● independent clans
- ☐ pseudo-independent clans

**FFT Parse Tree**

Figure 8

**Cost metrics**

Our treatment of the cost calculation is abstract. We expect the cost function to be tailored to reflect differences in the target architecture. For this example we focus on a refinement of the metric used in our previous work. Sarkar [Sarkar, 1986] determines the overhead cost of node execution to be a function of the sum of the data sizes of inputs, latency, and the sum of data sizes of outputs. We adopt this decomposition of the cost of data transfer. However, in the example the overhead of reading inputs \( r \) and writing outputs \( w \) is associated as overhead with the cost of execution giving total node cost.
It is typical of distributed memory systems that transfer of data between processors be pipelined or concurrent. Thus the latency cost is incurred only once. Edge cost in the cost metric equals latency cost, 1. Our method applies cost metrics with similar node and edge cost composition functions in a flexible way for decomposing the data flow graph.

The following paragraphs describe a new cost model for partitioning that allows the assignment of a grain to a processor to be independent of the processor on which other grains have executed. This allows scheduling to select from among grains that have their data dependencies satisfied, rather than requiring a particular assignment. Edge costs are incurred between execution of grains.

![Diagram](image)

**Adjacent Independent Clans**

Figure 9

To determine which (pseudo-)independent clans should be executed in parallel, inspect the parent (linear) node in the parse tree and examine the adjacent children pairwise. Imagine the subgraph of Figure 9. 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 (pseudo-) independent clans contain 2 nodes, there are four possibilities for adjacent children of a linear node 1: (i) both the left and right children can be executed in parallel; (ii) the left child can be executed in parallel and the right child aggregated; (iii) the left child can be aggregated and the right child executed in parallel; or (iv) both children can be aggregated. In Figure 9 let n_i give the node cost of node i; e represents an edge cost. Let f, g, and g' be node cost combining functions for parallel execution of the child nodes, aggregated execution of independent children, and linearly aggregated execution (of aggregated children), respectively. And let h, h', and h'' be edge combining functions. Then we can formulate the costs of the four cases as shown in Figure 10 where boxes represent clans whose nodes are executed concurrently and ovals represent aggregation. For some parallel architectures such as a hypercube with no shared memory, an appropriate function for the communication delay between several parallel processors is the maximum delay between any communicating pair. The execution time for parallel nodes is also the maximum function and the execution cost of aggregated nodes is the sum of their individual costs. We adopt this model for our example formulas. (For a shared memory
model, in which all data produced are stored and each input is a separate data access, summation may be appropriate for combining edge latencies.)

Given a pair of independent nodes the decomposition chooses the configuration that gives the minimum combined execution and communication cost. (Although there are many problems for which binary operations suffice, the binary case can be generalized for nodes of bounded degree.)

<table>
<thead>
<tr>
<th>Formula</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>general: ( f(n_1, n_2) + h(e_1, e_2, e_3, e_4) + f(n_3, n_4) )</td>
<td><img src="image1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>example: ( \max(n_1, n_2) + \max(e_1, e_2, e_3, e_4) + \max(n_3, n_4) )</td>
<td><img src="image2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>general: ( f(n_1, n_2) + h'(e_1, e_2, e_3, e_4) + g(n_3, n_4) )</td>
<td><img src="image3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>example: ( \max(n_1, n_2) + \max(e_1, e_2, e_3, e_4) + (n_3 + n_4) )</td>
<td><img src="image4.png" alt="Diagram" /></td>
</tr>
<tr>
<td>general: ( g(n_1, n_2) + h''(e_1, e_2, e_3, e_4) + f(n_3, n_4) )</td>
<td><img src="image5.png" alt="Diagram" /></td>
</tr>
<tr>
<td>example: ( (n_1 + n_2) + \max(e_1, e_2, e_3, e_4) + \max(n_3, n_4) )</td>
<td><img src="image6.png" alt="Diagram" /></td>
</tr>
<tr>
<td>general: ( g'(g(n_1, n_2), g(n_3, n_4)) )</td>
<td><img src="image7.png" alt="Diagram" /></td>
</tr>
<tr>
<td>example: ( (n_1 + n_2) + (n_3 + n_4) )</td>
<td><img src="image8.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>

Parallel vs. Aggregate Cost Decision Formulas

Figure 10

Since the leaves inherit their costs from the program DAG, we start the internal node calculations from bottom up and determine the best configuration for adjacent pairs of independent nodes. Each decision corresponds to an edge in the parent (linear) clan. 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 11. In this case, the aggregation 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. They take the forms illustrated in Figure 12.

![Diagram](image9.png)

Linear Clan with Multiple Adjacent Independent Clans

Figure 11
Stable Adjacent Decisions
Figure 12

The unstable configurations disagree on the aggregation of the shared node and must be resolved. The heuristic chosen is a local smoothing technique. Specifically, the aggregation decision for the shared node is redetermined using its context on both sides. Unstable pairs are resolved by selecting the best aggregation alternative for the combined configuration. Examples of unstable decisions and their corresponding resolutions are illustrated in Figure 13. The conflict is resolved by choosing the alternative with minimum cost.

Resolution of Unstable Adjacencies
Figure 13

Example: Partition and schedule
In the model, execution time and communication delay are estimated. For the FFT example we assign two sets of arbitrary values to show the flexibility of the algorithm. In the first example we assign the cost of 1 to the initial and final nodes; nodes 9 through 12 are given an arbitrary execution time of 20; nodes 13 through 16 are assigned time 30 and 17 through 20, 20. For all edges, we assume the edge cost is 25. At each linear clan, the relative merits of parallel execution of each (independent) child is weighed against the aggregation of the children. At node C24 in Figure 14, for example, we compare the costs of the 4 possible decisions: the cost of executing both children in parallel and communicating the results of C21 to C19 is 77 (30+25+22). To aggregate the nodes of C21, communicate the results to one of the nodes of C19 and concurrently execute C15 and C16. This yields a cost of 107 (60+22+25). To concurrently execute 13 and 15, pass results to the aggregated C19 costs 99 (30+25+44); and to aggregate both children (i.e. execute the entire clan on a single processor) costs 104 (30+30+22+22). Clearly, the best decision is to execute both children in parallel and absorb the communication cost. At node C27, the total time, including communi-
cation delays is calculated to be 124. The linear ordering at nodes C27, C24, and C25 force the execution of C5, C6, C7 and C9 to complete before starting any other processes. C21 precedes C19; C22 precedes C20. Four processors (P1-P4) are all that are required for the task. Simultaneously, the 4 processors will compute C5, C6, C7, and C8. When that work is complete, they will compute nodes 13, 15, 14 and 16 and pass the results to processes C15, C16, C17 and C18. Figure 15 illustrates the complete schedule.

Solution for Data Set I
Figure 14

Processor Assignments

Schedule for Data Set I
Figure 15
As a second example to demonstrate the flexibility of the method, we again assume the communication cost of all edges is 25. Nodes 1-8 have costs of 60; 9-12 costs of 50; 17-28 costs of 5. Here the decision at node C24 in figure 16 is to execute all children sequentially on the same processor. The 4 decisions as listed in the paragraph above now give comparative results of 45 (5+25+15); 50 (10+25+15); 60 (5+25+30); and 40 (5+5+15+15). The best decision for this data is to use a single processor to perform the computation of nodes 13,15,17,18,21,22,23, and 24.

Solution for Data Set II
Figure 16

Processor Assignment

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 the load balancing paradigm for our two examples. The load balancing total times of 172 and 225 for data sets I and II respectively, compares to 124 and 200 for graph decomposition.

Schedules for Load Balancing

Figure 18

Conclusions and future work

We have presented a practical method for parsing an arbitrary directed acyclic graph representing the data flow graph of a program to determine potential parallelism in its computation. The method is more powerful than its predecessor in decomposing primitive factors that previously were not analyzed. It was shown how the completely analyzed hierarchical structure of the graph can be interpreted using a cost metric, to find an efficient partition for parallel execution. The cost metric applied in this paper illustrates some of the general issues for designing cost metrics for decomposition, and shows a simple technique for resolving aggregation decisions. The FFT example illustrated that the method can be used to partition a non-trivial and useful problem.

The parsing step is general and powerful. Additional work is required to systematize the definition and evaluation of cost metrics and to extend the graph decomposition method for additional execution models. Realistic metrics are needed for computing approximations of total execution cost including communication, latency, task initiation and termination, and task suspension and resumption overhead for parallel architectures.
BIBLIOGRAPHY


