Partitioning and Scheduling Using Graph Decomposition

C. L. McCreary*  J. J. Thompson
Department of Computer Science and Engineering
Auburn University, Alabama 36849
(205) 844-6300
mccreary@eng.auburn.edu

D. H. Gill  T. J. Smith
The MITRE Corporation
McLean, Virginia

Y. Zhu
North Dakota State University
Fargo, North Dakota

Abstract

Automated parallelization of source code is a goal on which many researchers in parallel computing have focused. The increasing availability of parallel computers, the difficulty of creating good parallel programs, and the vast amount of existing serial source code all contribute to the need for automated means of parallelization.

This paper centers on the issues of partitioning and scheduling within automatic parallelization, or the creation of appropriately-sized tasks and their assignments to processors. An algorithm is introduced which uses the program dependence graph (PDG) representation of serial programs, and relies on a prior graph decomposition, or parse, for identification of parallelism. The algorithm uses local heuristics to determine the cost effectiveness of each opportunity for parallelization, and creates and schedules tasks accordingly.

Keywords

partitioning, scheduling, automatic parallelization, graph decomposition, program dependence graph

*This research is sponsored by NSF grant number CCR-9203319
I. Introduction and Background

The goal of our research is to extract all possible parallelism in source programs. Code sections that could execute concurrently can be found in loops (both between iterations and within iterations), in consecutive non-loop statements, and among subprograms. One of our tools, based on PAT, the Parallelizing Assistant Tool developed at the Georgia Institute of Technology [14], extracts loop-level parallelism. The partitioning/scheduling tool analyzes the resulting program dependence graph for non-loop level parallelism and parallelism within loop statements.

A theoretical model is required to provide common ground for the discussion of the algorithm. We adopt Sarkar’s model of execution, where a program dependence graph (PDG) represents the flow of data and control in a program [13]. Nodes in the graph represent sections of executable code; their weights are estimates of execution time. Edges between nodes represent communication that will be required if the nodes are assigned to different processors. Edge weights are estimates of communication times.

The graph decomposition required by the scheduling algorithm is based on the concept of a clan, which originated in the work of Ehrenfeucht and Rozenberg [1]. A clan is defined informally as a subset of the vertices of a graph such that "every element not in the subset is related in the same way (i.e. ancestor, descendant, or neither) to each member in the subset." [6] Clans are identified as trivial, independent, linear, or primitive, where each single node is a trivial clan, and primitive clans may be decomposed into independent and linear clans. Nodes within an independent clan could be executed in parallel, whereas those within a linear clan have dependences which order their execution. These properties make clans well-suited for identifying parallelism. Figure 1(a) illustrates an example of a program dependence graph with all clans identified.

A good algorithm exists for locating all clans in a program dependence graph, decomposing all
the primitive clans, and building a bipartite *parse tree* from the clans, with linear and independent clans at alternating levels [5]. We refer to this step as the *parse* of the PDG. The scheduling algorithm in this paper assumes that the PDG has been parsed, and the resulting parse tree along with the original program dependence graph are available as input. An example parse tree is shown in Figure 1(b).

![Diagram of Program Dependence Graph and Parse Tree]

**Figure 1. Program Dependence Graph and Parse Tree**

### II. An Overview of the Algorithm

The responsibility of the algorithm is to examine every possibility for parallelism, ignore those possibilities which are too costly, and intelligently exploit those which would result in a decrease in execution time. The possibilities for parallelism are cleanly identified by the parse tree as independent clans. The children of an independent clan exhibit no sequentializing dependences; simply, they may be executed in parallel. The traversal of the parse tree is depth-first so that all lower-level clans will be processed before their antecedents. The bottom of the parse tree represents the finest grain size available. Processing the parse tree bottom-up allows adjustment of the grain size as opportunities for...
parallelism are accepted (parallelization is cost-effective for grain size) or rejected (aggregation chosen, grain size is increased). Aggregation is defined as the serialized execution of all nodes in the clan on a single processor.

Parallelization or aggregation decisions are made at linear clans, as the linear clan’s independent children are examined pair-wise. The decision for the pair is then one of several choices, among which are: aggregate both the left and right children, aggregate the left child and parallelize the right, parallelize the left child and aggregate the right, or parallelize both the left and right children. Within these decisions, multiple choices exist for the assignment of nodes to processors. Other decisions require closer scrutiny of the parse tree. This method provides the framework for the scheduling algorithm introduced in this paper. Additional concepts which improve its performance are discussed in the following sections.

![Multi-stage Decision Graph](image_url)

Figure 2. Multi-Stage Decision Graph
III. A Multi-Stage Decision Graph

As previously mentioned, the algorithm proceeds with a depth-first traversal of the parse tree. It makes decisions by examining the independent children pair-wise at each linear clan, but the decision encompasses all the children. A multi-stage decision graph (MSDG) is created from all possible decision sequences for the independent children of the linear clan [12].

Given a linear clan with n independent sub-clans as shown in Figure 2, we construct an (n+2)-stage decision graph, and number the stages from 0 to n+1. The MSDG nodes in stage 0 and n+1 are the source and sink, respectively. Each stage i between 1 and n has one node for each decision option, where a decision option may be full aggregation, full parallelization, or one of the more precise options introduced later in this paper. Each edge from stage i to stage i+1 is weighted with the cost of communication requirements and execution costs for stage i+1, which are calculated based on the decision options the incident nodes represent. In the MSDG shown in Figure 2, A_i represents full aggregation of independent clan i, P_i represents its full parallelization, and X_i represents one of the more precise options.

Once the graph is complete, the shortest path from source to sink is found using a multi-stage optimization algorithm [4]. The shortest path represents the sequence of decisions which will result in the lowest total cost for executing the linear clan.

IV. Processor Assignment

When the decision for an independent child is to parallelize, processor assignment must be considered [3, 6]. For example, if the independent child consists of two nodes, each of which communicates to a third node, the third node should be placed on the same processor as the previous with the more expensive communication. Similarly, if communication times are constant, and two nodes
communicate to a third node and are to be parallelized, the more expensive of the two should be assigned to the same processor as the third. In this way, some of the communication will overlap with the greater computation time of the more expensive node. These methods extend for multiple communicating nodes and varying node and edge weights.

![Diagram](image)

**e1, e2 are edges in the PDG**

**Example 1**

<table>
<thead>
<tr>
<th>Node</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>50 units</td>
</tr>
<tr>
<td>n2</td>
<td>50 units</td>
</tr>
<tr>
<td>n3</td>
<td>50 units</td>
</tr>
<tr>
<td>e1</td>
<td>25 units</td>
</tr>
<tr>
<td>e2</td>
<td>10 units</td>
</tr>
</tbody>
</table>

**Example 2**

<table>
<thead>
<tr>
<th>Node</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>n1</td>
<td>40 units</td>
</tr>
<tr>
<td>n2</td>
<td>50 units</td>
</tr>
<tr>
<td>n3</td>
<td>40 units</td>
</tr>
<tr>
<td>e1</td>
<td>25 units</td>
</tr>
<tr>
<td>e2</td>
<td>25 units</td>
</tr>
</tbody>
</table>

**Figure 3. Processor Assignment**

Figure 3 shows two examples in which processor assignment is useful. For both examples, we calculate the cost of parallelization from these two formulas, and choose the less expensive option:

1. $\max(n_1, n_2 + e_2) + n_3$

2. $\max(n_1 + e_1, n_2) + n_3$
The first formula represents the assignment of nodes 1 and 3 to the same processor, and the second represents the assignment of nodes 2 and 3 to the same processor. The formulas yield execution times of 110 and 125 units respectively, for the first set of example weights. If parallelization is chosen as the option, then nodes 1 and 3 should be assigned to the same processor, while node 2 executes separately and communicates its results.

The same two formulas are applicable in the second example of Figure 3. The costs are 115 and 105 respectively, indicating that we should assign nodes 2 and 3 to the same processor, while node 1 executes separately and communicates its results. Similar formulas can be used to represent varying numbers of communicating nodes and different decision options [7, 12].

![Figure 4. Grain Structure Analysis](image-url)
V. Grain Structure Analysis

In some cases, a more precise examination of the independent clans is required at the time of the decision [2]. When both of the following are true: some of the independent clans contain sub-clans for which parallelize decisions were made, and the communicating nodes between the clans are not part of the previous parallelize decision, a reduction in communication can sometimes be found. Figure 4 illustrates a situation of this type.

If one or more independent clans are to be parallelized, the communicating nodes could still be aggregated, thus saving communication in some cases. When calculating the cost of parallelizing both independent clans I9 and I10, we also examine the option of executing nodes 5, 6, 7, and 8 on one processor. If communication is costly relative to the execution costs of these nodes, this option with its reduced communication requirements may less expensive. This more precise decision, or grain structure analysis, is one of the more precise options referred to when the multi-stage graph was introduced.

VI. A Relative Schedule

All the above concepts and techniques are taken into consideration at a linear clan. In order to maintain the lower-level decisions as the traversal moves upward in the parse tree, a relative schedule is created at each linear clan. It records a suggested processor assignment for each node, and the node’s earliest possible start and end times. When the decision at a higher level linear clan incorporates this schedule, all the information is offset appropriately based on the relative schedules of other sub-clans within the linear clan.

Figure 5 illustrates the offsets in relative schedules occurring at linear clans. The lower level linear clans have no independent children, so their schedules (drawn in Gantt chart form) represent the
sequential execution of their children. The root linear clan has one independent child. Assuming parallelization of this child, the relative schedule for this clan shows the offsets from execution and communication that would represent the decision.

Because the parse tree is processed bottom-up, leaf clans are traversed prior to the clans which contain them. Each leaf clan is initialized with a relative schedule on processor 0, starting at 0.0, and ending at the time equal to its weight.

If the original program dependence graph is connected, the root of its corresponding parse tree will be a linear clan. When the root of the tree is reached and the decisions are made, the last relative schedule contains suggested processor assignments and execution time intervals for every node.

A graph which is disconnected decomposes into a parse tree with an independent clan as the
root. In this case, there are multiple relative schedules remaining when the root is reached. However, the schedules are independent of one another, and only the processor assignments must be offset in order to maintain the integrity of the existing schedules within the one final schedule.

VII. Special Case Aggregation of Communicating Nodes

Aggregation has been defined previously as the serialized execution of processes on a single processor. When considering aggregation of a clan which has sub-clans which have been previously parallelized, this definition loses some of its meaning. If one or more sub-clans have been previously parallelized, multiple processors are already required.

Aggregation will be chosen for a clan when communication is too expensive for a parallelization decision to be cost effective. If communication costs are not consistent, the situation can occur where aggregation is chosen for a clan with internal parallelization decisions. Figure 6 shows an example where this is the case. Linear clan L1 is considered first. Opportunities for parallelization are exploited as shown in L1’s relative schedule of Figure 6(c). But when the next linear clan (L2) is reached, the communication costs are far too costly to allow its parallelization. In this special case, the definition of aggregation must be reconsidered.

The algorithm deals with this anomaly by choosing the less expensive of the following two options: aggregate all communicating nodes that are a part of the decision leaving internal parallelize decisions as they are; or serialize the entire clan, in effect discarding all internal potential for parallelization. While the second option would appear to be consistently more expensive, its likelihood of being chosen increases as the variation of communication costs within the clan increases.

In the example of Figure 6, the aggregation of only the communicating nodes is the less expensive option. One internal parallelize decision is maintained, while the other is discarded due to the
expense of its outgoing communication. The relative schedule of linear clan L2 is shown in Figure 6(c).

![Program Dependence Graph](a) Program Dependence Graph

![Parse Tree](b) Parse Tree

![Schedules](c) Schedules

Figure 6. Special-Case Aggregation of Communicating Nodes

VIII. The Algorithm

Input: A program dependence graph labelled with node and edge costs, and a parse tree.

Output: A processor assignment, and start and end times for each node of the program dependence graph.

for each clan processed in depth-first traversal of parse tree {

    switch (type of clan) {

        case Linear :

            create empty multi-stage graph;
            R = first child of clan;

}
compute cost of aggregating R;
compute cost of parallelizing R;
add costs to multi-stage graph;
L = R;
R = next sibling to the right;

while (R is defined) {
    compute cost of aggregating R if L was aggregated;
    compute cost of aggregating R if L was parallelized, using best processor assignment;
    compute cost of parallelizing R if L was aggregated, using best processor assignment;
    compute cost of parallelizing R if L was parallelized, using best processor assignment;
    compute costs of options based on grain structure if applicable;
    add costs to multi-stage graph;
    L = R;
    R = next sibling to the right;
}

find shortest path through multi-stage graph;
create relative schedule based on decisions, checking for special-case conditions on communicating nodes;
break;

case Independent :
    if node is the root of the parse tree
        combine relative schedules into one;
    break;

case Leaf :
    create initial relative schedule;
    break;
}
IX. An Example

We now apply the algorithm to an example program dependence graph. Figure 7 illustrates the PDG and its corresponding parse tree. This example was chosen because it illustrates most of the features of the algorithm.

The traversal locates clan 17 as the first linear clan to be processed. Leaf and independent clans have been traversed prior to L-17; however, only initialization takes place at these type clans and is omitted from the discussion for brevity. The multi-stage decision graph required for L-17 is shown.
in Figure 8(a). It requires two stages, one for each child, and an initial and final node. The upper node in each stage corresponds to an aggregation decision, while the lower node represents parallelization. Because no lower level parallelize decisions exist at this clan, grain structure analysis is not applicable, and the upper and lower nodes are all that are required in the multi-stage decision graph.

The edge labels were calculated as follows. Aggregating I-15 costs $100 + 100 = 200$ units; parallelizing I-15 costs 100 units. At the next stage, the corresponding child is a leaf; consequently, a parallelize decision is undefined, and the edges leading to a lower node are not viable. Aggregation of this child when the previous child was aggregated costs 5 units; aggregation after parallelization costs $25 + 5 = 30$ units, since communication from the previous stage is required. The shortest path through the graph is indicated by boldface. The decision sequence suggested by the shortest path is to parallelize the first child and aggregate the second.

A relative schedule is now created for L-17. It combines the relative schedules of leaf clans 4, 5, and 9. The relative processor of 5 is offset by 1 to represent parallelization. The relative start and end times of leaf clan 9 are offset to establish an allowance for execution and communication of the schedules on which it depends. Node 9 is assigned to the same processor as node 4 by default; in this case, either would have resulted in the same cost.

**Figure 8. Linear Clan 17**
In Figure 8(b), a Gantt chart for the execution of L-17 shows the information recorded by the relative schedule. Identical analysis of linear clan 18 results in a similar relative schedule.

The traversal moves through more independent and leaf clans, where more initialization takes place, and then finds linear clan 21 as the next clan for consideration. A decision graph with three stages is created for the clan, and is shown in Figure 9(a). Since lower-level parallelization decisions are present between two of the children, I-19 and node 11, and the communicating nodes (9, 10, and 11), are not part of the decision, grain structure analysis is appropriate. This increases the width of the decision graph by 2 at the second stage.

The nodes in each stage of the MSDG are interpreted as follows: if there are only two nodes, the upper node represents an aggregation decision and the lower node corresponds to parallelization; if there are four nodes, in order of uppermost to lowest they represent aggregation, parallelization, aggregation with grain structure analysis, and parallelization with grain structure analysis. The "aggregation with grain structure analysis" decision at stage i+1 means that the nodes in stage i+1 are
aggregated with each other, as well as with the communicating nodes from stage i. Similarly, the "parallelization with grain structure analysis" decision at stage i+1 represents a decision to parallelize at stage i+1, but to aggregate all of the communicating nodes between stages i and i+1. As with the previous linear clan, some edges may represent an undefined decision.

The edge weights were calculated as follows. I-19 consists of the two linear clans previously scheduled. Recall that they each cost 130 units. The cost of aggregating I-19 is then 130 + 130 = 260 units. Parallelization costs 130. At the next stage of the graph, we consider the second child of I-19. First we calculate the cost of aggregation after aggregation to be 5 units. Aggregation after parallelization costs 25 + 5 = 30 since the communication cost is required. Any edges to the next node of this stage would be undefined, since parallelization of a single node has no meaning.

![Figure 10. Effect of Grain Structure Analysis](image)

Recall that the third node of the stage corresponds to aggregation with grain structure analysis. The cost of this option after aggregation is 5 units, since it represents the execution of node 11 on the same processor as last scheduled node from the previous stage with which it communicates. This option after parallelization represents the aggregation of all the communicating nodes between stages...
on the same processor. This would cause one of nodes 9 and 10 to be unscheduled, placed on the same processor as the other, and followed by node 11 of this stage. Figure 10 illustrates the change in relative schedules that will take place if this option is chosen. The cost is $5 + 5 = 10$ units, which was calculated by the additional cost of the schedule for stage $i$, and the cost of stage $i+1$. The fourth node of the second stage is unreachable, since it also represents parallelization of a single node.

We now move to the third stage of the decision graph. It has links from only two of the nodes in the previous stage. Since both of the previous nodes represent aggregate decisions, edges extending from them are the same. Aggregation after aggregation costs $25 + 25 = 50$ units. Parallelization after aggregation also costs $25 + 25 = 50$ units.

The shortest path is shown in boldface. The best decision sequence is given by parallelize, aggregate with grain structure analysis, and aggregate. Figure 9(b) shows the chart representing the relative schedule for this linear clan. L-17 and L-18 are parallelized, which causes an offset in processor assignments. Nodes 9 - 13 are aggregated with each other by the decision sequence, and onto the same processor as node 4 by default.

The remaining leaf and independent nodes are processed, and the traversal ends with the root linear clan, L-24. The four children of L-24 require the decision graph to have four stages. Grain structure analysis is not applicable between any pair of children, so the width of each stage is 2. Unlabelled edges in Figure 11(a) represent undefined options.

The labels were calculated as follows. The cost of aggregating node 0 is 10. At the next stage, the cost of aggregation after aggregation is $300 + 50 + 50 = 400$; parallelization after aggregation will cost 300 units, if nodes 1 and 0 are destined for the same processor. This will allow the communication and execution of nodes 2 and 3 to take place during the time node 1 executes. At the third stage, aggregation after aggregation costs $190 + 150 = 340$; aggregation after parallelization costs $190 + 150$
+ 25 = 365; parallelization after aggregation costs 25 + 190 = 215; and parallelization after parallelization costs 25 + 190 = 215. The costs at the final stage are 5 + 5 = 10 for aggregation after aggregation, and 5 + 5 = 10 for aggregation after parallelization. Note that the communication to node 14 will be avoided if we assign node 14 to the same processor as nodes 12 and 13.

![Multi-Stage Decision Graph](image)

**Figure 11. Final Schedule**

The shortest path is shown in boldface, and represents the decision sequence aggregate, parallelize, parallelize, aggregate. The Gantt chart of Figure 11(b) represents the relative schedule of the root of the tree, and hence the schedule of the original program dependence graph. The parallelized version completes in 535 units on 5 processors, for a speedup of approximately 2 over the serial version.

**X. Analysis and Complexity**

The scheduling problem has been proven to be NP-complete [15]. Because the optimal solu-
tion is not known in the general case, the performance of scheduling algorithms must be measured relative to other algorithms. In terms of speedup and efficiency, early comparisons of the performance of our algorithm with other techniques have been favorable [9].

The execution costs of our algorithm are highly dependent on the characteristics of both the program dependence graph and the parse tree. Many variables, including the size of the PDG, the shape of the parse tree, and variations in the numbers of children at different levels of the parse tree affect the execution time, and therefore preclude the calculation of a tight theoretical bound. Empirical analysis of execution times of randomly generated PDGs have indicated a loose approximation of the complexity as $O(n^2)$ in the size of the input graph. We continue to study the issue using empirical data and amortized analysis.

XI. Ongoing Research

Additional related research at Auburn University continues. An extensive comparison of the algorithm with other promising static scheduling research is taking place. Further work in making more precise decisions which closely represent the characteristics of specific parallel architectures will improve the application of the scheduling algorithm [3, 11]. This will effectively add another option to each stage of the multi-stage graph created for each linear node. The option will represent parallelizing for a pre-determined grain size, calculated from characteristics of the target architecture.

XII. References


