AUTOMATIC PARTITIONING AND VIRTUAL SCHEDULING FOR EFFICIENT EXECUTION

Technical Report CSE-91-02

Carolyn L. McCreary, D. H. Gill

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

March 1991
Automatic Partitioning and Virtual Scheduling
for Efficient Parallel Execution

C.L. McCreary *
Auburn University

D.H.Gill**
The Mitre Corporation

Abstract:

Current parallel programming practices and tools are almost entirely architecture dependent. Efficient execution of a computation depends upon the extent to which program structure accommodates the performance characteristics of the target machine. Programs are developed and tuned for particular architectures and for particular configurations of processing and memory components. The software investments are largely wasted when highly optimized codes require reoptimization and reprogramming in order to transition to a new architecture or configuration. As an alternative to labor-intensive manual reprogramming, automated programming support is needed for efficient construction and execution of architecture-tailored programs.

This paper describes a method for partitioning and scheduling a program dependence graph (PDG) for efficient execution on a target multiprocessor architecture. The process decomposes the PDG into a hierarchy of subgraphs that are potential execution packages or grains. Through cost analysis of the target architecture, the appropriate scale and structure of the grains is determined. The grains are then assigned to virtual processors where consecutive grains are identified as program threads. A final schedule is determined by merging non-overlapping threads. Since the graph decomposition discovers the structure of the entire PDG, the technique is superior to the more standard approaches of focusing on the critical path or nearest neighbor nodes. The partitioning and scheduling technique described here will be part of a semi-automatic parallelizing tool to aid programmers in the development of parallel programs.

Key Words: parallelizing compiler, parallel, partitioning, multiprocessor scheduling, grain size

Presenter: Dr. Carolyn McCreary
Department of Computer Science and Engineering
Auburn University, AL 36849
e-mail: mcreary@eng.auburn.edu
Automatic Partitioning and Virtual Scheduling for Efficient Parallel Execution

C.L. McCreary *   D.H. Gill **
Auburn University   The Mitre Corporation

1. Introduction

Researchers approach the problem of programming multiprocessors from two differing points of view. One group is developing new languages and/or new language constructs and requiring the reprogramming of any application for a parallel system. The other group is developing parallelizing compilers or tools to enhance compilers to take an existing serial program, automatically detect tasks that can run concurrently, and schedule tasks for parallel execution. Both approaches have advantages over the other and investigation must continue along both lines. The language approach has the advantage of using human intelligence to develop entirely new algorithms that can uncover parallelism not available in the sequential algorithm. The compiler tool solution can eliminate the cost of reprogramming and use system information to efficiently utilize resources. Performance characteristics of the system that are rarely known by the programmer, but are essential to the creation of an efficient parallel program, can be incorporated into the compiler tool to enhance the system performance. The research of this report gives a method for automatic determination and scheduling of parallel modules from an existing sequential computation.

In compiling a sequential program for execution on a multiprocessor system, there are four major problems to be solved: [26]

1. Analyzing the data dependences and control dependences in the program.
2. Identifying parallelism in the program.
3. Partitioning the program into grains of sequential tasks.
4. Scheduling the tasks on processors.

The work of this paper addresses the last three problems. The automatic dependence analysis, the phase that detects where parallelism is constrained, has been studied extensively, and there exist a number of tools (e.g. PAT[3], PTOOL[2], Parafrase[21], Faust[19] and PTRAN[1]) that create data flow and control flow graphs from sequential code. A directed graph is typically used to model the dependence relation. Usually, a node of the graph G = (N,E) represents an operation such as a statement or block of statements, and an edge (u,v) represents the dependence of v on u, forcing the execution of u before v. For both data and control dependence, the key is to represent only essential dependence constraints as edges. This paper assumes that a dependence graph exists and discusses a method for performing the last three tasks on the dependence graph.

*This research is sponsored by NSF grant no. CCR-8909253
**This research is sponsored by The MITRE Corporation
The technique decomposes the data flow graph into grains of the appropriate size for any underlying homogeneous multiprocessor architecture, determines which grains should be executed in parallel and which must be executed sequentially, and schedules those grains onto processors.

2. **The Conflict Between a High Degree of Parallelism and Communication Overhead**

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. 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.

The graph decomposition method explained in this paper divides a data flow graph of the computation into a hierarchy of potential grains, employs a general metric to represent communication and execution times, and using the metric, determines the grains that will most efficiently execute the computation. The grains are then scheduled by first defining threads of grains to be executed on the same processor and assigning the threads to processors.

Determining a grain decomposition that will yield the fastest possible execution is an NP-hard problem for which any tractable solution will yield only approximate results. The heuristics of the graph decomposition technique considers for grains subgraphs called clans. The nodes outside the clan view the sources of the clan as a single node and the sinks as another single node. Communication can be reduced by aggregating all clan nodes into a single grain to be executed on one processor where the communication to all clan sources is equivalent to communication to one graph node and where communication from clan sinks is equivalent to a multicast from the clan. See figure 1.

Graph decomposition has several properties that make it very useful in the parallelization of a computation:

1. A clan is a collection of computational elements with identical need to communicate with other clans. This property demonstrates that grouping by clans yields a substantial reduction in communication overhead.

2. Clans are classified as linear or independent. The linear clans of a data flow graph must execute sequentially, while the independent clans may execute in parallel.

3. The clan structure derived from a data flow graph is unique, and forms a hierarchy that can be regarded as a parse tree of the data flow graph, the df-parse. Traversal of the df-parse tree shows different degrees of parallelization that can be applied to a particular program.

4. If a cost metric that models a particular architecture is imposed on the df-parse tree, a good grain size can be determined for that architecture. Once the grains have been determined, a
natural traversal of the df-parse tree yields an efficient scheduling algorithm.

5. The graph decomposition technique can be applied to a variety of systems from those involving very high communication costs (for example, distributed systems) to tightly coupled multiprocessor systems. The cost metric reflects both the communication and processing characteristics of the target hardware and the hierarchy of grains allows a system a choice from which to find the proper balance of aggregation to parallelization.

6. The graph decomposition method is robust in that it can be applied to any directed acyclic data flow graph. Towsley [30] and Bokhari [6] develop scheduling algorithms for trees and series-parallel graphs. In addition to our effort, a few partitioning and scheduling schemes such as LAST by Baxter and Patel[4] and linear clustering by Kim and Browne [20] use as a basis a general acyclic data flow graph.

3. Parsing a Dataflow Graph

The grains derived for parallel computing will be subgraphs of the dataflow known as clans. The dataflow graph is a directed acyclic graph (DAG) whose edges denote the essential dependence of the input computation. A set of nodes X from DAG G is a clan iff for all x, y in X and all z in G - X, (a) z is an ancestor of x iff z is an ancestor of y and (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 any element outside the clan has the same ancestral relationship to every node in the clan. The importance of a clan as a grain is that all sources and all sinks of the clan can be seen as identical in their communication with the rest of the graph.

Clans can be classified as linear, independent, or primitive[14]. Linear clans must be executed serially whereas independent clans contain subclans (or singleton nodes) that may be executed in parallel. Primitives do not have such a clear execution order and are processed further to be decomposed into linear and independent components[25]. Resulting independent clans are referred to as pseudo-independent clans. After complete decomposition, all clans are labeled as linear, independent, or pseudo-independent.

A parsing algorithm, O(n^3) in complexity and reported elsewhere [23], creates a hierarchy of clans in a parse tree, which we will refer to as the df-parse (data flow parse) tree. Ehrenfeucht and Rosenberg [14,15,16] prove that there exists a canonical parse leading to a unique parse tree. The labeling of the nodes of the df-parse tree as linear, identifies clans that must be executed serially and the labeling as independent or pseudo-independent identifies those that may be executed in parallel. The df-parse tree is bipartite with linear clans connected as parents or children to (pseudo-) independent clans. For a connected graph, the root of the df-parse tree is a linear node. Figures 2,3,4, and 5 show an example graph, clan decomposition, and df-parse trees.

4. Cost Metrics

Our treatment of the cost calculation is abstract to allow the cost function to be tailored to reflect differences in the target architecture. Our method applies cost metrics in a flexible way. Node and edge cost composition functions are evaluated on the parsed data flow graph to
determine the appropriate grains for execution on an architecture whose performance is modeled by the metric.

The following paragraphs describe a 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. Communication costs are incurred between execution of grains.

To determine which (pseudo-)independent clans should be executed in parallel, inspect the parent (linear) node in the df-parse tree and examine the adjacent children pairwise. 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 l: (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 6 let $n_i$ give the node cost of node i; $c_i$ represents a communication 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 $h, h', \text{ and } h''$ be edge combining functions. Then we can formulate the costs of the four cases as shown in Figure 7 where boxes represent clans whose nodes are executed concurrently and ovals represent aggregation.

A variety of communication models for multiprocessor systems have been proposed [18,22,26] and for purposes of illustration of our method, we will show how Sarkar's model of communication overhead can be represented as a metric for clan decomposition and scheduling. Communication has three components: a writing cost function $W_c$, a reading cost function $R_c$, and a delay component $D_c$. Following actual measurements from the iPSC/2, we model the delay cost function as a linear function of message size, $k$. Then $D_c = t_{\text{startup}} + k*t_{\text{send}}$ where $t_{\text{startup}}$ and $t_{\text{send}}$ are constants. Under the assumption that message broadcasting is available, there is no penalty for sending a message to multiple destinations and so $W_c$ is independent of the number of destinations. On the other hand, we assume reading must be done serially for each incoming message and the receiving node's reading cost, $R_c$, must be additive.

The composition function for parallel node execution time is the maximum function, $\max(n_1,n_2)$ for 2 nodes. The composition function for aggregated node execution cost is the sum of their individual costs, $(n_1+n_2)$ for two nodes. We adopt this model for our example formulas. Given a pair of independent nodes, the decomposition chooses the configuration that gives the minimum combined execution and communication cost.

The binary case represents the types of decisions that must be made for the generalized problem. In the general case, however, more than an all or nothing aggregation decision must be made. Arbitrary subsets of independent node may yield the best results. This question is the focus of some of our further research.

In Figure 7 all formulas give the process execution time of the left nodes, the communication cost, and then the process execution time of the right nodes. $W_i$ is the write cost of node i; $R_i$ is the cost of reading the message from edge i; and $k_i$ is the size of the message sent by node i. Since we assume broadcasting carries no penalty, cases 1 and 2 use maximum as the formula for the write
nd delay. The write cost of an aggregate is greater than the write cost of either component, and is modeled by the sum. The read component of the communication metric assumes the reading must be done serially for each incoming message as is represented by the sum.

\[
\begin{align*}
\text{Formula} & \quad \text{Case} \\
\text{general:} & \quad f(n_1, n_2) + h(e_1, e_2, e_3, e_4) + f(n_3, n_4) \\
\text{example:} & \quad \max(n_1 + W_i + t\text{start} + k_i * \text{tend}) + \max(R_1 + R_3 + n_3, R_2 + R_4 + n_4) \text{ for } i = 1, 2 \\
\text{general:} & \quad f(n_1, n_2) + h'(e_1, e_2, e_3, e_4) + g(n_3, n_4) \\
\text{example:} & \quad \max(n_1 + W_i + t\text{start} + k_i * \text{tend}) + R_1 + R_2 + (n_3 + n_4) \text{ for } i = 1, 2 \\
\text{general:} & \quad g(n_1, n_2) + h^{''}(e_1, e_2, e_3, e_4) + f(n_3, n_4) \\
\text{example:} & \quad (n_1 + n_2) + [W_1 + W_2 + t\text{start} + (k_1 + k_2) * \text{tend}] + \max(R_1 + n_3, R_2 + n_4) \\
\text{general:} & \quad g'(g(n_1, n_2), f(n_3, n_4)) \\
\text{example:} & \quad (n_1 + n_2) + (n_3 + n_4)
\end{align*}
\]

Figure 7

The metric is applied to the df-parse tree from bottom up. The leaves receive their costs from the program DAG. Internal node calculations occur at linear nodes where the best configuration for adjacent pairs of the independent children is determined. 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 8. 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 9.

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 10. The conflict is resolved by choosing the alternative with minimum cost.

5. Scheduling

The general scheduling problem is to assign grains to processors in such a way as to minimize the parallel execution time which includes both processor execution time and communication delay. In the context of the graph decomposition method, the schedule associates grains developed by the partitioning scheme with processors utilizing the highest degree of parallelism suggested by the partition. Communication delays have already been considered in the partitioning scheme, so a simple mapping of the df-parse tree nodes onto processors is all that is required.

The df-parse tree of the DAG provides a structure from which a schedule can be developed. Efficient, optimal scheduling algorithms have been developed for algorithms represented by trees.
[5,6], but the problem of scheduling the df-parse tree is different in nature. In the previous scheduling algorithms, a tree represents the precedence ordering of the tasks represented by the nodes. The df-parse tree represents the hierarchy of clans found in the graph. In the df-parse tree, only the leaves represent the tasks to be scheduled and the precedence ordering is the left to right ordering of the subtrees. The tree's linear nodes show the dependency order of the data flow graph. For every linear node in the df-parse tree, the subtrees formed by each child must be executed in left to right order. The independent nodes which have been designated to execute in parallel indicate the grains that are to be executed concurrently.

Prior to scheduling, we assume the graph-decomposition technique has been applied and the following conditions are met:

A. The df-parse tree has been calculated and the appropriate grains have been defined in a previous grain determination phase.

B. In the df-parse tree, the dependency order of the children of linear nodes is preserved: i.e. if l is a left sibling of b, then l must be executed before b.

C. At each independent (pseudo-independent) node, the decision to parallelize or not has been made and parallel grains have been determined.

D. As many processors are available as needed by the algorithm.

E. The cost of communicating a message between one pair of processors is the same as between any other pair.

Assumption D will is an area for future work and plans are being made to incorporate the number of nodes in the physical system with the cost metric. Assumption E is quite realistic on many systems. Shared memory systems clearly have this property and even some distributed memory systems display this characteristic. In the work we are doing on communication performance on the iPSC/2, the number of hops between nodes does not contribute strongly to the overall cost of the communication. This conclusion is supported in the work of others [7].

The scheduling is done in two phases. In the first phase a virtual schedule is created where threads are formed. A thread is a sequence of graph nodes (or equivalently a sequence of df-parse-tree leaf nodes) to be executed on a single processor. A start time and completion time are associated with each thread. The second phase associates processors with threads in a greedy, load balancing fashion.

The grain determination phase classifies tree nodes as linear, leaf, aggregated (pseudo-)independent and parallel (pseudo-)independent, and the scheduler uses the classification to create the threads. The virtual scheduling algorithm recursively traverses the df-parse tree in pre-order. The parallelization decisions made at the linear nodes during the grain determination phase show the processor and communication requirements. When a decision to parallelize is made, threads are created.

The communication requirements of the linear nodes vary with the parallelization/aggregation decisions. At each linear node we consider the communication requirements of two adjacent children. If the left child, L, has been parallelized, but the right, R, has not, all processor results computing L must be passed to the processor computing R. Similarly if L is aggregated, but R is parallelized, L must be broadcast to all processors executing R. If both L and R are parallelized, all processors executing L must communicate results to all processors executing R.

Given adjacent nodes Land R in a linear factor, if L is pseudo-independent, each grain of L
passes its results to a subset of grains of R. In contrast, if L is independent, there is a complete connection between L and R and each grain of L must pass results to all grains of R [25]. In either the case of a parallelized independent or pseudo-independent node, communication delay occurs. Both of these cases will be treated as the independent case. It is the easiest case to schedule, the most conservative, and may result in no communication cost addition for an architecture and language that readily supports broadcasting.

Leaves represent the actual computations and are the tree nodes that are scheduled for execution on the processors.

6. Scheduling Algorithm

The virtual scheduling algorithm is implemented by the recursive function Schedule which returns the updated set of threads. The schedule function is initially activated at the root of the df-parse tree. As each node of the df-parse tree is visited, threads are updated in a way dictated by the node labeling decisions made in the grain determination phase.

If the node is a leaf, it is appended to the sequence of nodes in the current thread and the thread end time is increased to reflect the execution requirements of the node.

If the node is labeled linear, the children from left to right are scheduled and threads are updated to reflect the communication requirements between two adjacent children. In the algorithm, the function Append_Broadcast determines the communication requirements between a child, CL and its right sibling, CR. Results must be sent from each thread in the subtree with root CL to each thread in the subtree rooted at CR and the end times of the threads must be updated to reflect the communication delay.

If the node is labeled parallel, a “best” node is selected for the current thread and new threads are created for each other child. The recursion schedules each child on its newly created thread.

If the node is labeled aggregate, its children are recursively scheduled on the current thread. Unless there is a child of the aggregate that is labeled parallel, the current set of threads, CPS, returned by the schedule function is not changed and no update of the set of threads actually takes place.

Figure 12 shows the threads derived by applying the scheduling algorithm to figure 11. Figure 13 shows the final schedule.

A functional description of the algorithm:

Simple Data Types:
Clan
Time
Metric

Types:
Ordered_Clan_List = Null + (Clan × Ordered_Clan_List)
Thread = Time × Time × Ordered_Clan_List
Thread_Sett = Null + (Thread × Thread_Sett)
Functions:
Schedule = Clan × Time × Thread × Metric -> Time × Thread_Set
Children = Clan -> Ordered_Clans_List
Append_Node = Clan × Thread × Time × Metric -> Time × Thread
Insert_Communication_Delay = Thread_Set × Thread_Set × Time × Metric -> Thread_Set × Time
Create_Thread = Time × Time × Ordered_Clans_List -> Thread
Threads = Clan × Thread_Set -> Thread_Set
Choose = Clan_Set × Thread -> Clan
Next = Ordered_Clans_List -> Clan
First = Ordered_Clans_List -> Clan

Variables:
N : Clan
TS : Time start time
TE : Time end time
TC : Time current time
P : Thread current thread
PS : Thread_Set subtree thread set
Ch : Ordered_Clans_List children in d-f-parse tree
CB, CL, CR, C : Clan (best, left, right, arbitrary) child
NP : Thread new process thread
CPS, LCPS, RCPS : Thread_Set thread sets from subtrees
M : Metric

Schedule (N, TS, P) = {
    TE := TS;
    PS := {};
    Ch := Children(N);
    
    if Leaf(N) then
        (TE, P) := Append_Node (N, P, TE, M);
    elseif Linear(N) then
        CR := First(Ch);
        repeat
            (TE, RCPS) := Schedule (CR, TE, P, M);
            (RCPS, TC) :=
                -- broadcast from LCPS is inserted at the head of each thread of RCPS
                -- Maximum end time is returned
                Insert_Communication_Delay (Threads (CL, LCPS),
                Threads (CR, RCPS),
                M);
            TE := max (TE, TC);
            
            CL := CR;
            LCPS := RCPS;
        
            CR := Next(Ch);
        until End_Of_List(Ch);
    PS := PS + RCPS;
}

8
elsif Parallel(N) then
    CB := Choose (Ch, P, Metric); -- select best continuation
    -- for current thread;
    (TE,CPS) := Schedule (C, TE, P);
    PS := PS + CPS;
    forall C in Ch - (CB) loop
        NP := Create_Thread (TS, TS, []);
        PS := PS + [NP];
        (TC,CPS) := Schedule (C, TS, NP);
        TE := max (TC, TE);
        PS := PS + CPS;
    end loop;
elsif Aggregated(N) then
    forall C in Ch loop
        (TE,CPS) := Schedule (C, TE, P);
        PS := PS + CPS;
    end loop;
end if;
).

A generalized form of virtual scheduling from clan df-parse trees is presented in the algorithm VSchedule. In this algorithm thread selection and concatenation strategies are isolated in the calls to Sequence and Aggregate. As before, time is propagated in the ordered traversal. Threads from subtrees are selected for concatenation in postorder fashion, as opposed to extending a pre-allocated thread.

Functions:
VSchedule = Clan x Time x Metric -> Time x Thread_SetSequence = Thread_Set x Time x Thread_Set -> Time x Metric
Aggregate = Thread_Set x Thread_Set x Time x Time x Metric -> Thread_Set

Variables:
N : Clan                clan at root of current df-parse subtree
M : Metric              metric for architecture
TS : Time               start time for subtree at N
TC : Time               end time for subtree at current child
TE : Time               end time for subtree at N
Ch : Ordered_Clans      children in df-parse tree
C : Clan                child
PS : Thread_Set         completed thread set
CPS : Thread_Set        thread set from current child
LCPS, RCPS : Thread_Set thread sets from left and right subtrees

VSchedule (N, TS, M) = {
    TE < TS;
}
PS <- \{\};
Ch <- Children(N);

if Leaf(N) then
    P <- Create_Thread (N, M);
    TE <- M(N);
    PS <- \{P\};

elsif Linear(N) then
    LCPS <- \{\};
    TE <- TS;
    forall C in Ch loop
        (TE, RCPS) <- VSchedule (C, TE, M);
        (TE, LCPS) <- Sequence (LCPS, RCPS, TE, M);
    end loop;
    PS <- LCPS;

elsif Parallel(N) then
    forall C in Ch loop
        (TC, CPS) <- VSchedule (C, TS, M);
        TE <- max (TC, TE);
        PS <- PS + CPS;
    end loop;

elsif Aggregated(N) then
    CPS <- \{\};
    forall C in Ch loop
        (TC, CPS) <- VSchedule (C, TE, M);
        (TE, PS) <- Aggregate (PS, CPS, TE, TC, M);
    end loop;

end if;
return PS;
}

In both algorithms attention must be paid to threads from parallel nodes with aggregated ancestors. Determining the communication requirements of the parallel node with other nodes in the aggregate requires additional accounting. The communication information needs to be propagated upward for potential connection. One implementation approach would be to provide functions which distinguish, after any call to Schedule, which threads of the set returned still have unresolved input or output, and which were resolved by internal communication at a lower level of the hierarchy.

The call to Sequence from a linear node causes threads from the adjacent siblings, L and R, to be combined where that is advantageous. If both are singleton sets, the threads are always concatenated, and the cost of the resulting thread is the sum of the costs of the individual threads. The cost metric can be exploited to permit additional concatenations of communicating threads (providing suspend and resume are permitted in the computational model for the target
architecture).

For example, total elapsed time can be reduced if time to execute the longest thread is reduced. A strategy (similar to the critical path heuristic) would be to examine the longest threads, l and r, in each of two adjacent thread sets L and R and combine them into a single thread.

Aggregation occurs at an independent clan. The call to Aggregate means that the set of unresolved independent threads from the independent descendant clans will be executed sequentially. If there are threads from clans lower in the hierarchy that have been determined to execute in parallel, their internal sequential communications are already resolved (as described above). Remaining are threads that receive input data from the common parents and provide their output data to the common children of the current independent clan. A single thread is constructed by receiving the input values (once), concatenating the set of independent threads, and combining and broadcasting the resulting output values.

7. A Complete Example

For an example we selected a subgraph of a molecular physics application [11]. (The entire graph is too complicated to be instructive.) The objective was to choose a real application whose data flow graph was not completely regular so most of the features of the method would be employed. Regular graphs are much easier to decompose and analyze. The example graph is shown in Figure 14, and its df-parse tree in Figure 15. For simplicity, we assume uniform communication costs. For any node, \( W_c = 5 \), \( R_c = 3 \), \( t_{start} = 2 \), \( t_{send} = 0.1 \), and \( k_i = 5 \). Figure 15 shows the aggregate/parallelize decisions.

As an illustration of the parallelize/aggregate decision, consider the computations made at node g. Making a joint decision about nodes j and k where the only decision at j is to aggregate all or do all in parallel, leads to the 4 cases of table 7.

\[
\begin{align*}
\text{Case I:} & \quad [5+5+2+0.5] + [6+5] = 23.5 \\
\text{Case II:} & \quad [5+5+2+0.5] + [6+10] = 28.5 \\
\text{Case III:} & \quad 20 + [20+2+2] +3 + 5 = 52 \quad \text{(assuming total aggregation of node j)} \\
\text{Case IV:} & \quad 20 + 10 = 30 
\end{align*}
\]

since case I gives the lowest cost, our temporary decision is to parallelize the leaves of both j and k.

When considering a decision at g for nodes k and 20, there are only two cases:

\[
\begin{align*}
\text{Case I:} & \quad \text{parallelize k:} \quad [5+5+2+0.5] + [6+20] = 38.5 \\
\text{Case IV:} & \quad \text{aggregate k:} \quad 10+20 = 30 
\end{align*}
\]

This analysis indicates that we should aggregate at k! To reconcile the problem, compare the costs of the two decisions. If we parallelize k, we’ll have Case I from both analyses for a total cost of 62. If we aggregate, Case II from the first decision combines with case IV from the second, and will combine to give a total cost of 58.5 and the most cost effective decision is to aggregate the leaves of k.

Figure 16 depicts the virtual processor schedule and figure 17 shows the final schedule.
8. Summary and Comparison with Other Work

The goal of the work presented in this paper is to automate the structuring of parallel computations to the extent possible. The described techniques can be applied to the problem of automatically determining the parallel grain size and thread structure of an algorithm. The graph decomposition method uses the techniques in a parallel programming scenario in which an algorithm is developed by following the sequence of steps illustrated in Figure 18. The algorithm is coded, then transformed in step 1 into a graph that captures data and essential control dependencies. In step 2, the hierarchical structure in this graph is discovered by parsing the graph using the clan parsing algorithm. The graph construction and parsing steps are static and independent of a target architecture. In step 3 the algorithm is partitioned by identifying which potentially parallel regions of the computation should be executed in parallel and which should be aggregated (i.e., executed serially). The metric which models the target architecture is instrumental in determining the size of the grains defined by this partition. In step 4, the task or thread structure of the graph is extracted by a process called virtual scheduling. The fifth and final step is scheduling. This step allocates processing and communication resources to execute the algorithm.

Graph decomposition is a robust and comprehensive approach to the partitioning/scheduling problem. The technique most frequently referenced, load balancing, attempts to keep as many processors busy as possible. When a processor becomes available, it is assigned a task awaiting action. This strategy has been shown to yield results inferior to those of graph decomposition in [24][25]. Its objective, to keep as many processors as possible busy, is not the appropriate goal for parallel processors. The point of parallel processing for practical applications is to produce a system that executes programs in the fastest time possible. Load balancing suffers from ignoring communication costs.

Other partitioning techniques either fail to account for communication overhead[27], target only one class of problem[17], or minimize communication without regard to instruction execution time[6]. Other strategies attempt to map the graph of a particular algorithm or set of algorithms onto the physical layout of some architecture with primitive or no communication measurements[5]. The graph decomposition approach maps a general PDG onto an arbitrary architecture; it considers both communication overhead and execution time in its partitioning criteria; and it is based on sound heuristics for the distribution required.

Two partitioning techniques, other than graph decomposition, that offer suitable heuristics for meeting the goal of minimal total processing time have emerged. One is the LAST algorithm of Baxter and Patel [4], and the other is based on linear clustering [20]. Like graph decomposition, both techniques use a dag representation of the program dependence graph as input, consider communication edges in the partitioning, and apply a cost metric.

The LAST algorithm chooses nodes to be allocated based on their connectivity to previously allocated nodes, and then assigns them to processors for which elapsed execution time (including communication delays) is minimized. The criteria which measures the connectivity of a node with a group of nodes is an arbitrary heuristic and the cost assumptions in the model are not realistic. For example they assume that nodes k hops apart require a communication time of k times that of a neighbor communication. Our experiments and research results of others[7] have shown that the delay to multiple hops on the iPSC/2 is negligible. In such a setting, an architecture behaves as if
it were nearly completely connected and the LAST algorithm is nearly the same a load balancing.

Linear clustering first recognizes groups of nodes that must execute serially, and determines a "critical" cluster, i.e. most costly cluster. Clusters can be refined by cutting the critical cluster and redistributing its nodes. This process results in a Virtual Architecture Graph (VAG) which represents an optimal architecture for the given computation graph. The complexity of this step is $O(ne^3)$, where $n$ is the number of DAG nodes and $e$ the number of DAG edges. Once the VAG is found, it is mapped onto a physical architecture in a complicated process with complexity $O(l^2 p^2)$ where $l$ is the number of linear clusters in the VAG and $p$ is the number of processors in the physical architecture. The mapping technique can be applied to heterogeneous systems as well as homogeneous ones. The approach has two major drawbacks. It doesn’t provide the opportunity for aggregating closely communicating linear clusters, and the mapping of the VAG onto a processor architecture is complex and costly.

Currently comparison testing is being done on these three algorithms. The preliminary results of the study show that graph decomposition is superior to the LAST algorithm in the two cases that have been run to date. The improvement in total processing time for one case was about 18% and the other was 45%. The authors of the linear clustering algorithm have not yet defined the parameters in the criteria for finding the critical path and they have not yet been able to implement the VAG mapping. The six properties of Graph Decomposition listed above make it a theoretically best candidate for incorporation into a parallelizing compiler.

Our ongoing work addresses additional elements of the support required for architecture-independent programming. Metrics that accurately model parallel processing environments are a central component of this work and empirical validation of these metrics is one focus of our ongoing work. Developing grains and threads that will map onto a fixed number of processors is another open study topic, as is the application of the metric to linear nodes with more than 2 children.
Figure 1. Reduction by a Clan

Figure 2. An Example Graph

Figure 3. Clan Decomposition of Graph of Figure 2.

Figure 4. Parse Tree

Figure 5. Parse Tree
Figure 6. Adjacent Independent Clans

Figure 8. Linear Clan with Multiple Adjacent Independent Clans

Figure 9. Stable Adjacent Decisions

Figure 10. Resolution of Unstable Adjacencies

Figure 11. Parse Tree
Figure 15. Aggregate/Paralleline Decisions of Complete Example With Cost Assignments

Figure 14. Virtual Processor Schedule

Figure 17. Processor Schedule
Figure 18. Partitioning and Scheduling Method: Summary
Bibliography


277-301.


