Automatic Partitioning and Virtual Scheduling for Efficient Parallel Execution

C.L. McCreary
Auburn University

D.H.Gill
The Mitre Corporation

1.Introduction

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. 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. 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.

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.

4. Cost Metrics and Grain Determination

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.

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 is described in [25]. 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 1 let n_i give the node cost of node i; e_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', and h" be edge combining functions. Then we can formulate the costs of the four cases as shown in Figure 2 where boxes represent clans whose nodes are executed concurrently and ovals represent aggregation.

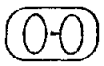
Formula	Case
$f(n_1,n_2) + h(e_1,e_2,e_3,e_4) + f(n_3,n_4)$	
$f(n_1,n_2) + h'(e_1,e_2,e_3,e_4) + g(n_3,n_4)$	
$g(n_1,n_2) + h''(e_1,e_2,e_3,e_4) + f(n_3,n_4)$	
$g'(g(n_1,n_2), f(n_3,n_4))$	

Figure 2

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 3. 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 4.

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 5. 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 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 distributed memory systems with hypercube topologies often display this characteristic [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 L and 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. In either a parallelized independent or a 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.

A functional description of the algorithm:

Simple Data Types:
Clan
Time
Metric

```
Types:
Ordered_Clan_List  =  Null + (Clan X Ordered_Clan_List)
Thread = Time X Time X Ordered_Clan_List
Thread_Set = Null + (Thread X Thread_Set)

Functions:
Schedule = Clan XTimeXThreadXMetric -> Time XThread_Set
Children = Clan -> Ordered_Clan_List
Append_Node = ClanXThreadXTimeXMetric-> TimeXThread
Insert_Communication_Delay  = Thread_Set X Thread_Set X
      Time X Metric -> Thread_ Set X  Time
Create_Thread = Time X Time X Ordered_Clan_List -> Thread
Threads = Clan X Thread_Set -> Thread_Set
Choose = Clan_Set X Thread -> ClanX
Next = Ordered_Clan List -> Clan
First = Ordered_Clan 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_Clan_List       children in df-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);
    elsif 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              -- (side effect)
            -- 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;

    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;

}.
```

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 set s 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 indcpendent 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.

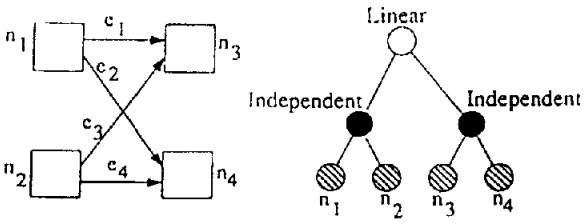


Figure 1. Adjacent Independent Clans

7. Summary

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 6. 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.

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.

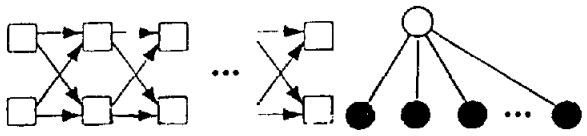


Figure 3. Multiple Adjacent Independent Clans

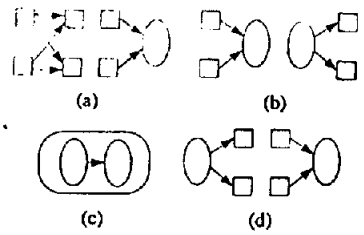


Figure 4. Stable Adjacent Decisions

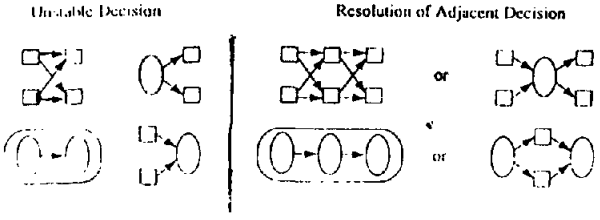


Figure 5. Resolution of Unstable Adjacencies

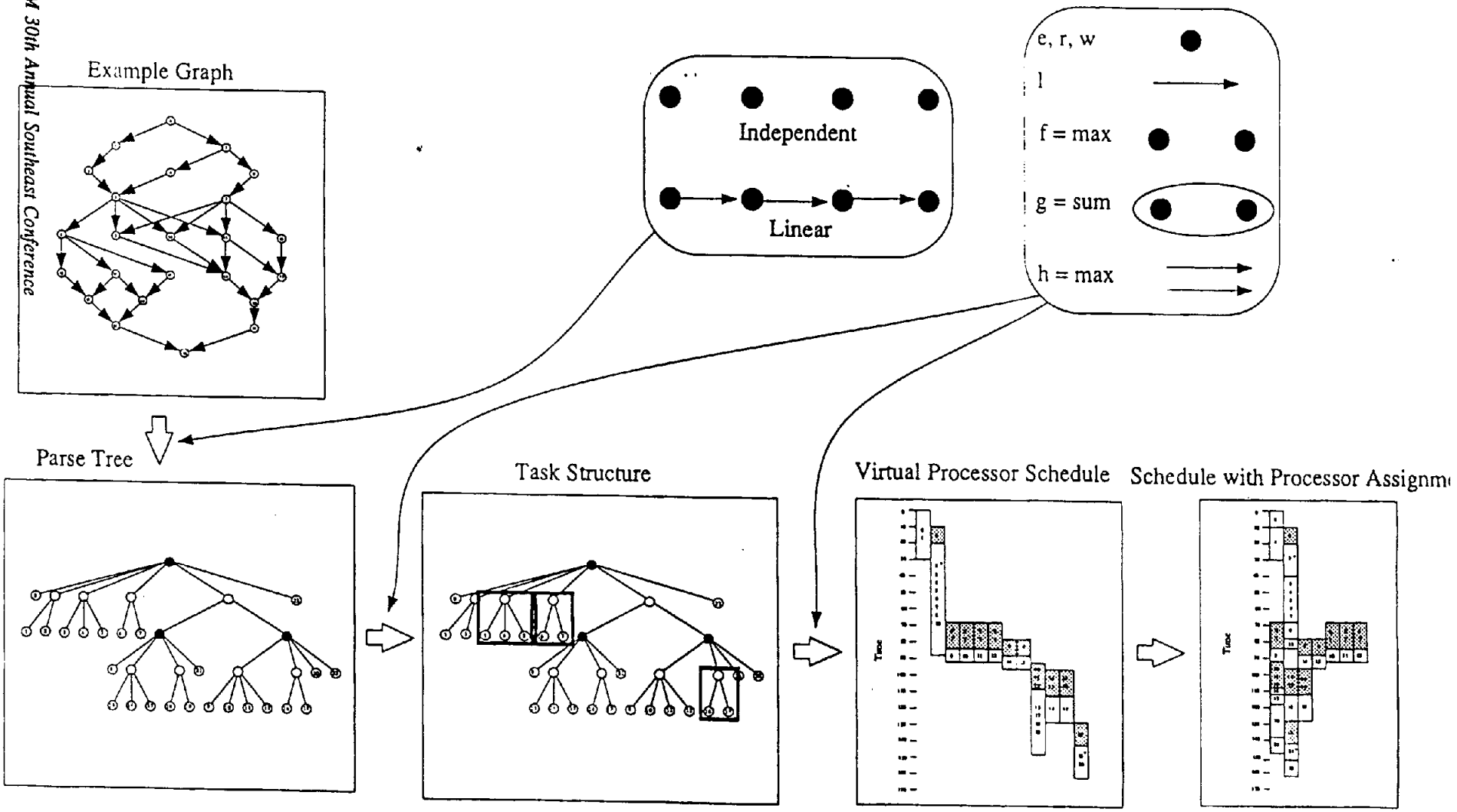


Figure 6. Partitioning and Scheduling Method

Bibliography

1. Allen, F., Burke, E., Charles, M., Cytron, P., and Feranti, R."An Overview of the PTRAN Analysis System for Multiprocessing", *Proceedings of the 1987 International Conference on Supercomputing*, Athens, Greece, 1987.

2. Allen, R., Baumgartner, D., Kennedy, K., and Porterfield, A."PTOOL: A Semi-Automatic Parallel Programming Assistant," *Proceedings of the 1986 International Conference on Parallel Processing*, pp. 721-727.

3. Appelbe, W. F. and Smith, K. "PAT: A Retargetable Parallelizing Tool for Fortran" *Proceedings of the IEEE 1990 Conference on Software Maintenance*, 1990.

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

5. Berman, F., "Experience with and Automatic Solution to the Mapping Problem", in *The Characteristics of Parallel Algorithms*, edited by Leah H. Jamieson, Dennis B. Gannon, and Robert J. Doublas, MIT Press, Cambridge, 1987.

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

7. Bomans, L., and Roose, D., *Hypercube and Distributed Computers*, Elsevier Science Publishers, New York, 1989.

8. Carroll, J. M., Thomas J. C. and Malhotra, A., 1980 "Presentation and Representation in Design Problem-Solving", *British Journal of Psychology*, 71 (1980), pp. 143-153.

9. Chang, S., editor, *Principles of Visual Programming Systems*, Prentice-Hall, Englewood Cliffs, 1990.

10. Chang, S., Ichikawa, T., and Ligomenides, P. A., editors, *Visual Languages*, Plenum Press, New York, 1986.

11. Coffman, E. "Computer and Job-Shop Scheduling theory", New York: John Wiley and Sons, 1976.

12. Dongarra, J. J. and Sorensen, D. C.,"SCHEDULE: Tools for Developing and Analyzing Parallel Fortran Programs," in *The Characteristics of Parallel Algorithms*, edited by Leah H. Jamieson, Dennis B. Gannon, and Robert J. Doublas, MIT Press, Cambridge, 1987.

13. Dongarra, J. J., Brewer, O., Kohl, J. A. and Fineburg, S., "A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel Processors," *Journal of Parallel and Distributed Computing*, 9 (June 1990), pp. 185-202.

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

15. Ehrenfeucht, A. and Rozenberg, G., "Pimitivity is Hereditary for 2-Structures", *Theoretical Computer Science* (1990), pp. 343-358.

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

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

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

19. Guarna, V. A., Gannon, D., Gaur, Y., and Jablonowksi, D. "FAUST: An Environment for Programming Parallel Scientific Applications," *Proceedings of Supercomputing '88.*

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

21. Kuck, D. H., Kuhn, R. H., Leasure, B. R., and Wolfe M. J., "The Structure of an Advanced Retargetable Vectorizer." In *Supercomputers: Design and Applications Tutorial* (Hwang K., ed.), IEEE Society Press, Silver Spring, MD, pp. 967-74.

22. Leirson, C., and Maggs, B., "Communication Efficient Parallel Graph Algorithms", *International Conference on Parallel Procesing*, 1986, pp. 861-868.

23. McCreary, C.L., " An Algorithm for Parsing a Graph Grammar", *Proceedings of the Computer Science Conference*, 1988.

24. McCreary, C. L. and Gill, D. H., "Automatic Determination of Grain Size for Efficient Parallel Processing," *Communications of the ACM* (1989), 1073-1078.

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

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

27. Stramm, B., and Berman, F., "Communication-Sensitive Heuristics and Algorithms for Mapping Compilers", Proceedings of ACM/ SIGPLAN PPEALS, 1988, pp.222-232.

28. Stramm, B., and Berman, F., "How Good Is Good?, Department of Computer Science and Engineering, University of California, San Diego, Technical Report CS90-169, 1990.

29. Stone, D. C., "Using Cumulative Graphic Traces in the Visualization of Sorting Algorithms", *SIGCSE Bulletin*, 21: 4 (Dec 1989), pp. 37-42.

30. Towsley, D., "Allocating Programs Containing Branches and Loops Within a Multiple Processor System," *IEEE Transactions on Software Engineering*, SE-12 (Oct 1986), pp. 1013-1024.

31. Wolfe, M. J., *Supercompilers for Supercomputers*, MIT Press, Cambridge, 1989.

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

33. Zima, H. P., Bast, H. J., and Gerndt, H. M. "SUPERB - A Tool for Semi-Automatic MIMD/SIMD Parallelization," *Parallel Computing*, 6 (1988), pp. 1-18.