VISUALIZING SMALL-GRAIN PARALLEL PROGRAMS FOR PERFORMANCE EVALUATION

Dae-Yeon Park

CIS-TR-91-14
June 25, 1991

Abstract

The problems of visualizing small-grain parallel programs are studied in this thesis. A visualization tool, called VISTA, was implemented using X–Windows. The goal of VISTA is to give effective visual feedback to the user analyzing a program. To achieve this goal, the tool summarizes a whole reduction tree in one window space using color. The user is able to see the reduction tree from different perspectives, such as PE, time, PE–time, and procedure views. VISTA also provides the user with several functions such as zoom-up, subtree display, and node inquiry to help analyze programs in more detail. The features and algorithms of VISTA are described in this thesis. Examples are given of how the tool can be used to improve one's understanding of both small-grain parallel program characteristics.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

ACKNOWLEDGEMENTS

First and foremost I wish to express my sincere gratitude to my advisor, Evan M. Tick for his guidance and support throughout this research. His tutelage, insight and intuitions have been invaluable. I learned a lot about parallel processing, especially the visualization of parallel programs and logic programming through my work in this project. I also wish to thank Gary Meyer for the discussions on the color spectrum, and Renga Sundararajan for his valuable comments on earlier drafts of this thesis. Special thanks are given to A. D. Malony at University of Illinois at Urbana-Champaign for providing me with color pictures. Finally, I wish to thank the faculty, staff and colleagues in the CIS department who have made my stay here a pleasant experience.

DEDICATION

To my family

TABLE OF CONTENTS

Chapter — Page

I. INTRODUCTION 1

II. LITERATURE REVIEW 6

- WAMTRACE 6
- Gauge 8
- Atrace 9
- Other Systems 11
- Summary 12

III. VISTA 13

- Problems and Solutions 13
- Toolkit Organization 18
- What You See in VISTA 18
- How to operate VISTA 22
- Summary 23

IV. IMPLEMENTATION OF VISTA 24

- Overview 24
- First Pass: Calculating Tree Height 26
- Second Pass: Level Condensing 30
- Weight Calculation 37
- Node Allocation 42
- Display Graph : X–Window Interface 44
- Summary 45

V. ANALYSIS OF PARALLEL PROGRAMS USING VISTA 48

- Case Study 1: Pascal's Triangle Problem 50
- Case Study 2: Semigroup Problem 56
- Summary 65

VI. CONCLUSIONS AND FUTURE RESEARCH 68

BIBLIOGRAPHY 70

LIST OF FIGURES

Figure Page

1. Example of WAMTRACE Execution Graph 8
2. Example of Gauge Execution Profile 9
3. Example of Atrace Action Screen 10
4. Execution Graph with Menu Window from PE View 17
5. Overview of the Organization of the Toolkit of VISTA 19
6. Window System in VISTA 19
7. Example of Execution Graph from Time View 20
8. Overall Control Algorithm for VISTA 27
9. Trace Record Format 27
10. Sample Program : Quick Sort in FGHC 28
11. Trace for Quick Sort Query "?- go([2,1,5,4,3],X)." 28
12. Tree Definition (Height of Tree = 3) 29
13. Algorithm for Calculating Tree Height 31
14. Hash Table after Processing 15th Record in First Pass 32
15. Transformation from Tree to Binary Tree 34
16. Algorithm for Second Pass: Level Condensing 35
17. Hash Table after Processing 15th Record in Second Pass 36
18. Whole Reduction Tree for Sample Program 36
19. Reduction Tree After Level Condensing for Sample Program by 2:1 . . 37
20. Reduction Tree After Level Connection for Sample Program 39
21. Weight Calculation for Sample Program 40
22. Algorithm for Weight Calculation 41
23. Node Allocation for Sample Program 43
24. Algorithm for Node Allocation 44

25. Algorithm for Display Graph (Main Routine) 46
26. Algorithm for Draw Graph (Sub Routine) 47
27. Example of Data Dependency Graph 49
28. FGHC Program for Pascal's Triangle 51
29. Execution Graph of Pascal's Triangle from Procedure View 52
30. Execution Graph of Pascal's Triangle from PE View (5 PEs) 54
31. Execution Graph of Pascal's Triangle from Time View (5 PEs) 55
32. Time Graph for Pascal's Triangle (1 PE) 57
33. Part A of Semigroup Program 59
34. Part B of Semigroup Program: Unbalanced Binary Hash Tree 60
35. Execution Graph of Semigroup from PE View (5 PEs) 61
36. Execution Graph of Semigroup from Time View (5 PEs) 62
37. Execution Graph of Semigroup from Time View (1 PE) 64
38. Subgraph of Semigroup from PE Perspective (5 PEs) 66
39. Subgraph of Semigroup from Time Perspective (5 PEs) 67

CHAPTER I

INTRODUCTION

A transition from uniprocessors to multiprocessors has been taking place, mainly due to the inherent performance limitations of uniprocessors. The transition to multiprocessors has also been driven by the economic reality that high performance is cheaper to attain with a collection of relatively slow processors than it is to attain using a single processor. Many kinds of parallel computers have been developed such as pipelined machines (e.g., CDC6600), vector machines e.g., CRAY-1), MIMD (multiple-instruction multiple-data stream) shared machines (e.g., Sequent, Encore, Firefly), and MIMD private memory computers (e.g., Hypercube and Transputer). With the availability of the hardware, one pressing question in parallel computing today is how to program parallel computers to solve problems efficiently, in a practical and economical way.

There is a consensus in the computer science community that making a parallel program more efficient is much harder than optimizing a sequential problem. One of the main factors contributing to this situation is the lack of robust symbolic debuggers, program profilers, trace generators, and similar tools which would help programmers analyze their programs and determine their execution bottlenecks.

The common method of evaluating the efficiency of parallel programs is by measuring the execution time and computing simple speedup. Collecting timing statistics and measuring speedups is often insufficient for understanding why the results are what they are. This is because in a sequential program, we know the

sequence of events, whereas in a parallel program, not only is the precise sequence of events unknown to us, but the sequence changes from one run to the next. Visualization can be used more effectively as an aid in understanding the behavior of parallel programs. Visualization tools for sequential programs have been available for a number of years. Similar aids for parallel programming have been developed [7, 6, 13], but, for the most part, they have been extensions of sequential tools that do not address the fundamental problems of complexity in parallel computations [9].

A goal reduction, which is equivalent to the procedure call and execution in imperative languages, is called reduction. A reduction tree consists of nodes and edges which represent the goals reduced during computation. The nodes of the tree are the goals which are reduced during the computation. Therefore, the reduction tree represents an execution graph. Granularity refers to the size of the tasks that are done in parallel. It ranges from a complete user program, which is called coarse-grain parallelism, to one instruction, which is called fine-grain parallelism.

In this project VISTA, which stands for VISualization Tool for Analysis of parallel programs, the problems of visualizing small-grain parallel programs are studied. I have focused on small-grain parallel programs, because such programs are inherently hard to analyze and as such seem well suited for visualization. Also, there is a greater need for tools to help deal with the added complexities of small granularity.

In this thesis, I survey the recent work that has been done in the context of visualizing parallel programs, and introduce a new visualization mechanism that facilitates performance tuning. In order for the visualization to be effective for programmers, multiple views must be displayed :

- reduction tree : without the reduction tree, it is difficult to know what the program execution looks like.
- processor activity : knowing which processor executed the reduction is important to understand parallel characteristics.
- timing : time dependency between reductions is another important factor for understanding parallel characteristics.

Since the reduction tree is usually large in small-grain parallel programs, and the shape of the tree can't be predicted, it is very difficult to represent all the above features in a reasonable window space (window space is defined as a rectangular area on the screen in which one views graphical output). Therefore, I approached this problem by a somewhat unusual method.

VISTA displays the execution graph in one window space, from four different perspectives (PE (Processing Element) Graph, Time Graph, PE–Time Graph, and Procedure Graph). Since the whole execution state is reduced to one window space, it is very important to utilize the window space efficiently. Therefore, instead of propagating the reduction tree from the top of the window space, a mechanism was adopted in which the reduction tree is propagated from the center of the window in a radial direction, since the reduction tree usually grows non-linearly towards the bottom of the tree. The window efficiency of this mechanism is better than that of conventional method (see Chapter III and V). If a reduction tree is too large to display in one window space, the tree is displayed with some condensations (see Chapter IV).

VISTA is an effective visualization tool, in that it:

- Summarizes the whole reduction tree in a single window space.

- Propagates the tree radially from the center of the window space, for high space utilization.
- Enables a user to view the tree from the various perspectives (PE graph, Time graph, PE–Time graph, and Procedure graph).
- Enables a user to get a close-up view of any portion of the reduction tree.
- Enables a user to view any subtree without condensation.
- Enables a user to make an inquiry (with the mouse) for information attributed to a node.
- Uses color to represent both PEs and procedures, and the RGB color spectrum to represent time.
- Uses efficient internal data structures and algorithms, enabling fast display and recomputations.

Thus VISTA provides a parallel programmer with a tool which can be used effectively to understand the program and evaluate its performance.

This project was originally designed for parallel logic programs, but other families of parallel or concurrent languages such as Pascal Plus, Ada, and Parallel-Lisp, can also use the tool effectively, provided a trace of the same format is generated (see Chapter IV).

This thesis is organized as follows: Chapter II surveys graphics tools described in the literature, used for visualizing parallel programs. Chapter III describes VISTA from users' viewpoint. The first section of Chapter III introduces the problems for visualizing small-grain parallel programs and then gives the solutions for the problems. The rest of Chapter III describes VISTA from the viewpoint of an operational

perspective. In Chapter IV, the algorithms for VISTA are explained. In Chapter V, I discuss how VISTA can be used effectively for analyzing parallel programs, giving two examples of Pascal's Triangle and Semigroup problems. In Chapter VI, conclusions are drawn about VISTA and the future work is presented. Appendix A gives the glossary used throughout this paper.

CHAPTER II

LITERATURE REVIEW

This chapter surveys various graphics tools, described in the literature, used for visualizing parallel programs. These tools are picked as examples because the goals of the tools are different. WAMTRACE and Atrace are tools for observing the behavior of parallel programs, and can be used for performance analysis. Gauge is an execution profiler for parallel logic programs. PPUTT is a tool for debugging a parallel program and analyzing program performance. Voyeur is a prototype tool that provides user-specific, visual views of parallel programs.

WAMTRACE

There are two types of parallelism that arise most naturally in logic programming computation models: AND-parallelism and OR-parallelism. AND-parallelism is the ability to execute the conjunctive tasks in parallel, and OR-parallelism is the ability to execute the disjunctive tasks in parallel. That is, in AND-parallelism, processes co-operate in generating one particular solution to a query or a clause body. On the other hand, OR-parallelism refers to the case when independent processes are assigned to the various clauses of a procedure.

WAMTRACE [2] is a visualization tool developed at Argonne National Laboratory. The tool is loosely coupled to Aurora [3], a parallel version of the Warren Abstract Machine (WAM) [16] that runs on a variety of shared-memory multiprocessors. The WAM is a stack-based machine for traversing the search tree repre-

sented by a Prolog program. The nodes of this tree consist of AND-nodes, representing multiple subgoals in a single clause, and OR-nodes, representing multiple clauses for a single predicate. Because Aurora is OR-parallel logic language system, WAMTRACE focuses on the display of the OR-tree. During execution, the OR-nodes of the search tree appear as branch points that provide alternatives that are to be explored in parallel. WAMTRACE displays the OR-tree by a conventional method (i.e., propagates the OR-tree from the top of window).

Figure 1 shows an example of the WAMTRACE execution graph. The nodes of the tree represent branch points. The level of parallelism has temporarily dropped off to 11, as can be seen both by counting the branches and by reading the display in control panel. The solid black circle indicates that a new branch is just about to be created out of that node. In each circle is a number, indicating alternatives at each branch point that have been explored. The panel in the upper third of the display is used for controlling the run.

WAMTRACE is different from VISTA, in that WAMTRACE :

- focuses on OR-parallelism, whereas VISTA puts emphasis on AND-parallelism.
- displays an execution graph dynamically (i.e., by animation), whereas VISTA does so statically.
- displays the graph step by step in detail, whereas VISTA summarizes a whole tree in one window.
- shows the OR-tree without reference to time, PE (the information displayed in control panel is the information for one node), whereas VISTA shows the AND-tree with reference to time, PE, and procedure.

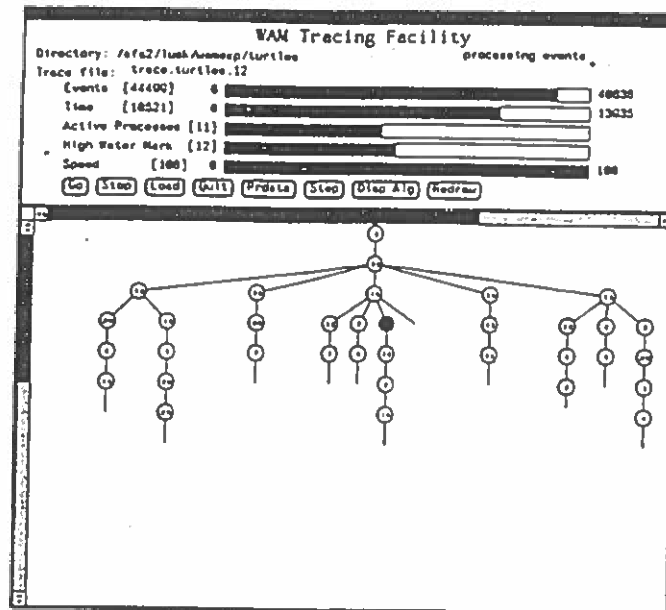


Figure 1: Example of WAMTRACE Execution Graph

Gauge

Gauge [8] was developed by Aerospace Corporation, originally to aid in profiling programs written in Prolog. The tool has also been incorporated into the Strand System [5]. Gauge uses internals built into Strand to analyze the runtime performance of a program. A static graphical display is generated, using color to show the amount of time and processing spent in specified Strand functions. Gauge displays the execution profile of a parallel program while the program is executed. Because the run-time overhead for gathering the data must be as small as possible to reflect the execution precisely, Gauge employs the combination of static and dynamic analysis. Run-time overhead is restricted to updating counters. Figure 2 shows an example of the Gauge execution profile. The goal names are sorted by the descending value of execution time (represented by horizontal bars). The panel in

the lower third of the display is used for controlling the run.

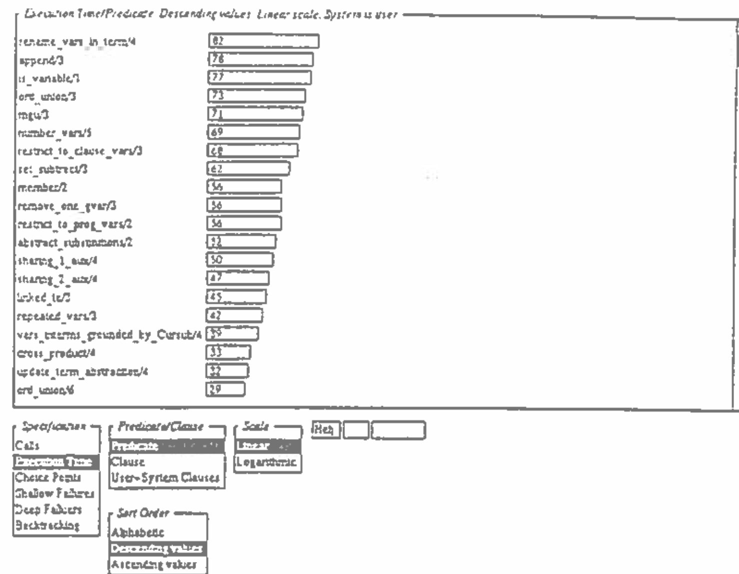


Figure 2: Example of Gauge Execution Profile

Atrace

Atrace [4] comprises a set of tools currently under development at Argonne National Laboratory. The tool is based around a general event logging scheme, in which programs of any type can log significant events in a specified format. These events are then displayed with any of several tools. Upshot, one of the Atrace tools, displays the events on a time line, showing their exact sequence. Unravel, another tool of Atrace, animates the event log, animating communication patterns, using color to display busy and idle states. Once Unravel has started to run, the action screen and the control panel appear on the screen, and thereafter, the user interacts with Unravel using a mouse. On the action screen, each task is represented by a

circle. The borders of the circles can be different colors, indicating a certain type of task. Tasks can be in one of various states: busy, idle, or in any user-defined state that is specific to an application. To indicate that a task is in a certain state, the circle is filled with a color. Messages are indicated by colored arrows between tasks. The arrows can also be different colors, indicating different types of messages.

Although these tools work with any programming system, their usefulness is limited to problems with a small number of processes because a circle represents a process and one action screen can contain only a very limited number of circles. Figure 2 shows an example of the action screen in Unravel. The number inside the circle represents the task number. Tasks 2, 7, and 8 are filled with the same color, indicating the same state. An arrow pointed from task 0 to task 10, indicating that task 10 just received a message from task 0.

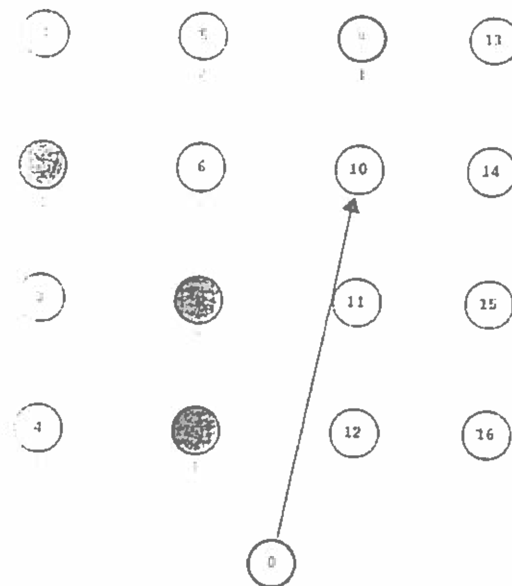


Figure 3: Example of Atrace Action Screen

Other Systems

PPUTT (Parallel Program Understand Tools and Techniques) [6], developed at University of Rochester, ia a visualization tool for the analysis and debugging of parallel programs. The goals of the tool are to help users understand the behavior of incorrect programs so that these programs can be debugged, understand a program's performance so that the program can be tuned, and improve both intuitive and formal understanding of parallel computations. The approach to the tools is based on a fine-grained characterization of parallel program executions, and a toolkit used to analyze executions graphically and interactively. The core of the toolkit consists of facilities for recording execution histories, a common user interface for the interactive, graphical manipulation of those histories, and tools for examining and manipulating program state during replay of a previously recorded execution.

Moviola, one tool of PPUTT, is the execution history browser and runs under the X–Windows system. Moviola implements a graphical view of an execution based on a DAG (Direct Acyclic Graph) representation of processes and communication. Moviola gathers process-local histories and combines them into a single, global execution history in which each edge represents a temporal relation between two events.

Voyeur [13], developed at University of Washington, is a prototype system that facilitates the construction of *user-specific*, visual views of parallel programs. These views range from textual views showing the contents of variables to graphical maps of the state of the computational domain of the program. The original motivation for Voyeur was to improve the trace facility of Poker [12]. Because there is often an increased gap between a programmer's mental conception of a problem and the realization of a parallel program that solves that problem, Voyeur allowed the par-

allel programmer to easily to construct application-specific, visual views of parallel programs. To achieve this goal, Voyeur prototype simplifies the task of learning how to build views.

Although both tools show great flexibility because of user specific view in Voyeur, and a high level view of an execution in PPUTT, the tools are limited in the number of processes the tools can show.

Summary

As discussed, some tools (e.g., WAMTRACE, Atrace) visualize the program execution dynamically (i.e., by animation), whereas others (e.g., Gauge, VISTA) show the graph statically. The animation mechanism has pros and cons: it gives the user a more "live" execution graph, but if a program is executed by a large number of processors or has a large number of processes, it is not easy to understand the animation graph.

Some tools (e.g., Gauge, PPUTT) display the execution graph *while* executing a program. With this mechanism, it is impossible to display a reduction tree in a fixed-size of window statically, because the size of the tree cannot be known before completing the execution of the program. The entire reduction tree is important for understanding a program (discussed in the next chapter), especially for the small-grain parallel programs.

CHAPTER III

VISTA

Parallel computations, with hundreds or perhaps thousands of asynchronous processes, are extremely complex. To be effective, visualization tools must assist the user in managing this complexity. That is, a visualization tool for parallel programs must first address all of the basic problems of sequential programs, such as display of the reduction tree. Furthermore, the tool must also address the fundamental characteristics of parallel computations, attributes such as processor activity, timing, and synchronization, which are not problems in sequential programs. However, it is difficult to visualize so many different attributes simultaneously and effectively, especially for fine-grain parallel programs. This chapter describes how VISTA resolves these problems, and explains VISTA from the user's viewpoint.

Problems and Solutions

In order for a visualization tool to help the user understand and analyze a small-grain parallel program, several things must be displayed. I believe that the most important view is the reduction tree (actually, this is not a problem of parallel computations). Without viewing the reduction tree, it is difficult to understand a program.

Since the reduction tree is large in small-grain parallel programs, a single window space is not adequate to represent it. Tools such as WAMTRACE [2] display the whole tree by conventional window mechanisms (i.e., by vertical and

horizontal slide bars). However, with WAMTRACE it is difficult to know how the entire execution appears. For example, it is difficult to know whether the reduction tree is symmetrical or not, and if the graph grows exponentially, linearly, etc. In VISTA, a reduction tree is displayed in a single window, regardless of the size of the tree. In order to achieve this goal, window space must be utilized efficiently. If the tree is represented by the conventional methods (from top of window), window space will be wasted since a tree usually grows non-linearly towards the bottom of the tree. For the window space efficiency, VISTA propagates the tree from the center of the window, in the radial direction. If a reduction tree is represented by the conventional method (i.e., a level is represented by a horizontal line), the maximum nodes n one level can represent is calculated as $n = \lfloor \frac{d}{w} \rfloor$, where d is the distance between two adjacent nodes and w is the window width, and n is the same for all levels. In the VISTA's method, n ranges from 1 (level 0) to the maximum $n = \lfloor \frac{d}{w\pi} \rfloor$. If the tree grows non-linearly towards the bottom of the tree (this is true for most reduction trees), in the conventional method, the window space is wasted in the lower levels (i.e., the levels closer to the root) and the window space is overcrowded (i.e., some nodes can not be displayed) in the higher levels. In VISTA, the higher the level, the larger space VISTA allocates to the level. Hence, the above problems of wasting window space at some levels and overcrowding at others are finessed to some extent. Therefore, the window space efficiency of VISTA is better than that of the conventional method.

Although window space utilization is improved to some extent by the above mechanism, the window space is still limited. Therefore, the problem of mapping an arbitrary large tree to limited window space must be resolved. VISTA solves this problem with two kinds of condensations: level condensing and node condens-

ing. They are defined and explained in Chapter IV. That is, VISTA summarizes a reduction tree in one window space with some condensations regardless of the size of the tree, and thus enables the user to view the entire reduction tree abstractly. After viewing the summarized tree (if the tree is within the window space limit, the whole tree is displayed without any condensation), the user can select any subtree in order to view without condensation. In order to represent the level of the reduction tree clearly, VISTA draws concentric circles of gray color with the same distance between any two adjacent circles. That is, each circle represents a level of the tree. The level distance is defined as the shortest distance between two adjacent circles and is the same throughout the tree. To represent parent-child and sibling relationships clearly, a node and its children form a colored polygon if the number of children is greater than one, or a line if it is one. If the node has no children, only the node is displayed.

The actual complexity of parallel programs arises from the fact that programs are executed by multiprocessors and the processors communicate with each other. To help understand the execution of parallel programs, a visualization tool must represent processor activity (which processor executed the reduction and how the processors communicate). This is not an issue in a sequential visualization tool. VISTA represents only the execution locality (which processor executed the reduction). Interprocess communication is not represented in VISTA, but can be inferred to some extent from the various execution graphs of VISTA (see Chapter V: Analysis of Parallel Programs using VISTA). Some visualization tools [4, 6] display the processor activity with a separate graph. However, with a separate graph, it is difficult to understand a program because the processor graph represents only processor activity without reference to the logical reduction tree. Colors play an important

role in VISTA, in solving this problem. Colors in the reduction tree (PE graph) represent a PE on which the node is executed. That is, VISTA displays the reduction tree and processor status simultaneously (in the PE graph). Figure 4 shows an example of the PE Graph. This graph is the reduction tree executed by four PEs from the PE perspective, for N–Queens problem. The N–Queens problem is to place N queens on $N \times N$ chessboard so that no queen can attack another queen.

Another important factor for understanding parallel programs is timing of reductions. Timing would be easy to understand if it were represented by the distance from the root. If the reduction tree and timing graph (represented by the distance) could be combined without critical loss, this would be the ideal way to understand parallel programs. Unfortunately, however, this combination of the reduction tree and timing graph seems almost impossible since the representation of the reduction tree and a timing graph with the above features would result in losing the shape of the reduction tree. To solve this problem (the loss of the shape of the reduction tree), VISTA uses colors again. Time is represented by the RGB color spectrum (Time Graph). That is, time is represented not by distance, but by the color spectrum in keeping with the reduction tree.

In summary, VISTA displays the reduction tree along with the PE, time, or procedure (explained in the Chapter III), which are represented by color. Although this scheme works well in most cases, these graphs still don't show PE and time simultaneously. That is, it is difficult to know *when* a node is executed (in the PE Graph), and by *which PE* the node is executed (in the Time Graph). In order to solve this problem, another graph (PE–Time graph) is introduced. In this graph, the reduction tree, PE, and timing are displayed simultaneously. The PE and time are represented by the colors of vertices and arcs (or polygon), respectively, in the

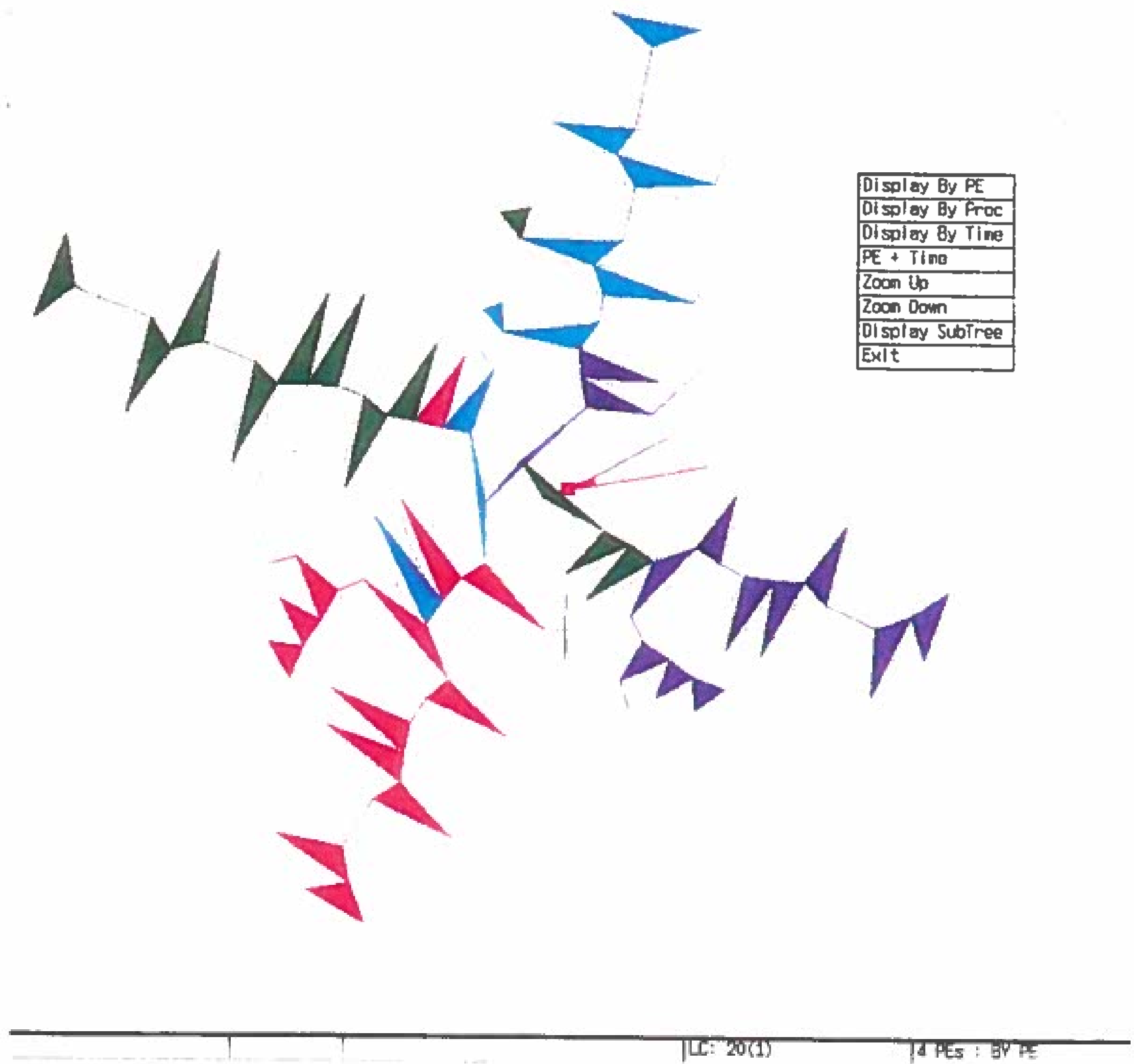


Figure 4: Execution Graph with Menu Window from PE View

reduction tree. The color of a polygon (or a line if the node has one child) represents the time the node is executed and the color of a vertex represents PE on which the node was executed. The shortcomings of VISTA are discussed in Chapter VI.

Toolkit Organization

Figure 5 is an overview of the organization of the toolkit of VISTA. VISTA is written in C, interfacing the X–Window system. VISTA runs on SUN SPARC stations. To use VISTA, a program is executed with a trace generator on a multiprocessor. The trace file and the source program are the input to VISTA. With these inputs, VISTA displays the execution graph in a window. The window system is composed of one main window and three sub-windows, as shown in Figure 6. The main window is used to display an execution graph. The time window is used to draw the time spectrum, and is displayed only in a Time Graph and PE–Time Graph. The color of the left side represents earlier time. A menu window is used to select a menu item. There are eight menu items, which are explained in the following section. The node window is used to display the information of the node when the user wants to know about the node in detail. The goal name (procedure name), PE number, time (represented graphically by a horizontal bar), the number of the nodes condensed to the node, level condensing information, and the current tree type, are displayed in the node window.

What You See in VISTA

In addition to the graphs described above, VISTA provides the user with several functions to help understand a program. Figure 7 shows an example of the execution graph from the time perspective for Pascal's Triangle problem (this problem is analyzed in Chapter V). The following are the functions VISTA provides.

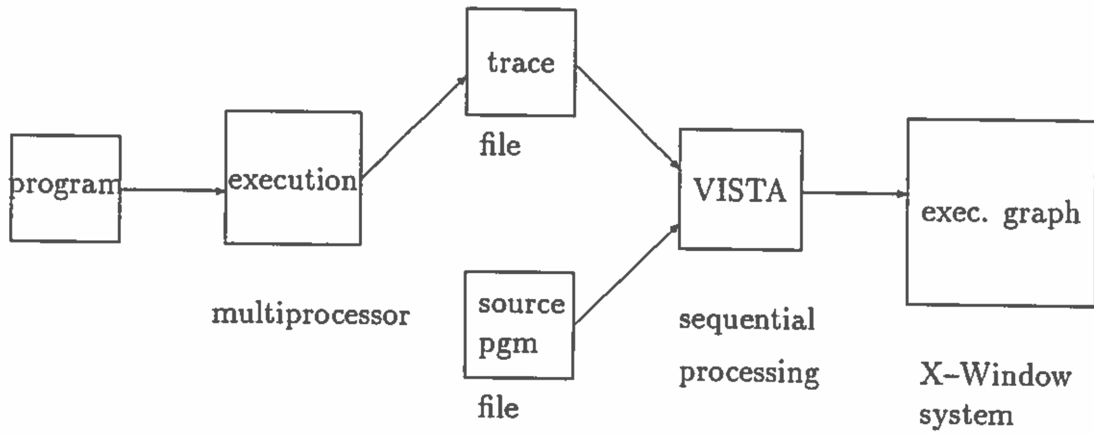


Figure 5: Overview of the Organization of the Toolkit of VISTA

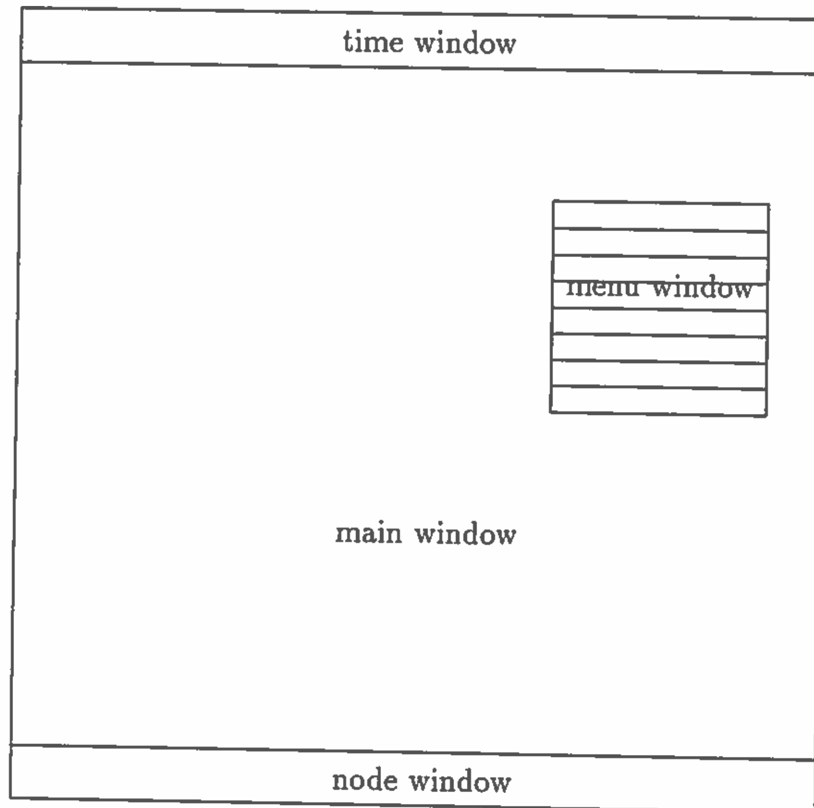


Figure 6: Window System in VISTA

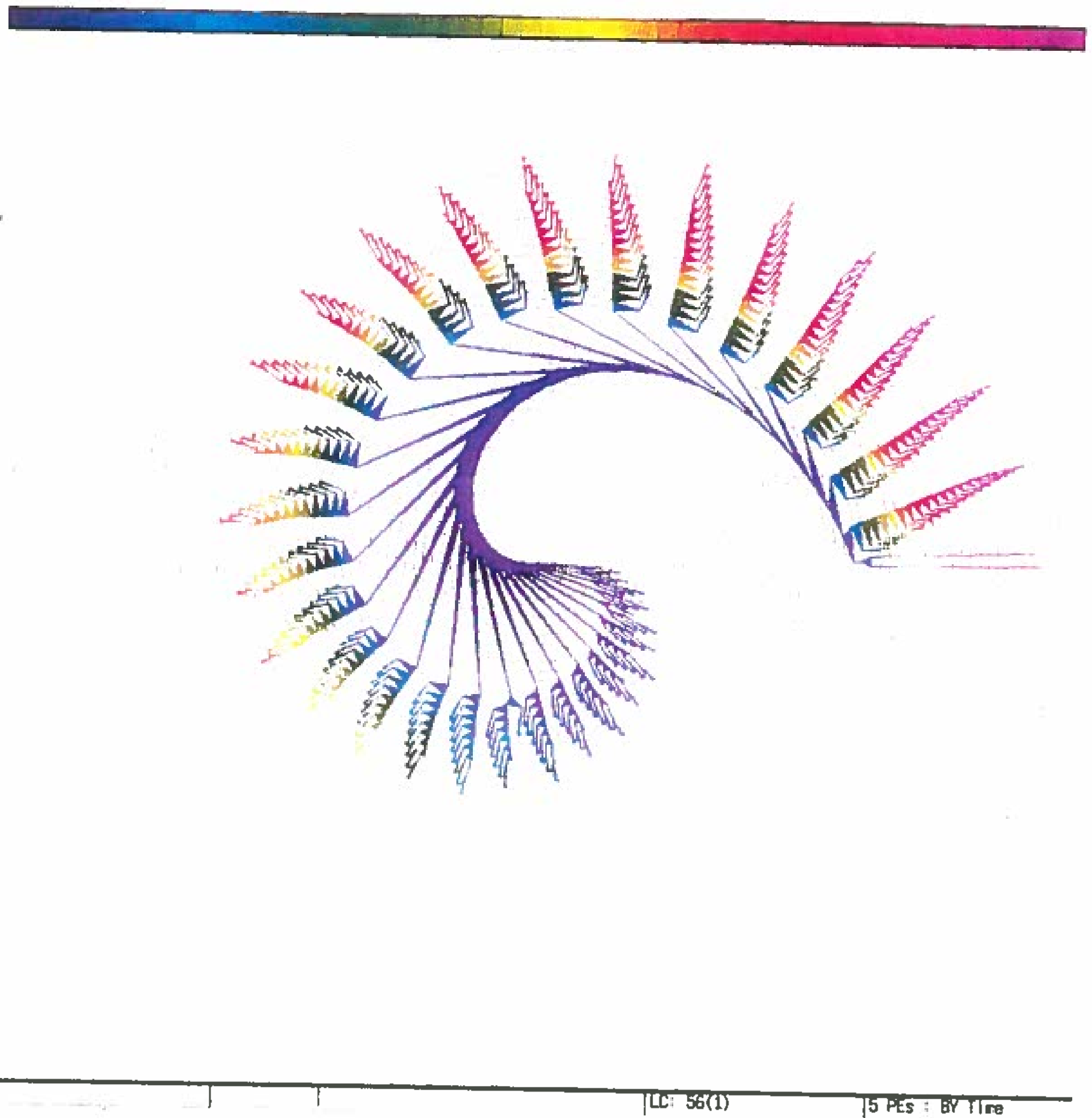


Figure 7: Example of Execution Graph from Time View

- Reduction tree from different perspectives

  VISTA provides four different kinds of execution graphs: PE Graph, Time Graph, Procedure Graph, and PE–Time Graph. Except for the Procedure graph, all of them are explained in the previous section (Problems and Solutions). Procedure Graph is the reduction tree from the procedure perspective. This graph is the exactly the same as the other graphs (PE Graph and Time Graph) except that the color represents the procedure. This graph can be useful also in a sequential tool.

- Changing the size or position of the tree

  - Zooming-up or Zooming-down

    If the user is not satisfied with the density of the displayed tree, a zooming function can be used. This function enlarges the tree while sacrificing the outer nodes (i.e., later reductions).

  - Changing the position of the tree

    The default position of the root node is the center of the window. This position can be changed to any place of the window. This function can be used effectively when the user wants to see some part of the tree in detail.

- Node Inquiry

  When the user wants to know about the node in detail, this function is used. If the user requests this function, VISTA displays in the node window, the procedure name, PE number, and the number of the nodes condensed to that node and time.

- Subtree Request

  If the user wants to view a subtree in more detail, VISTA displays the descendants of the node without condensation. In this subtree, the user can request zooming-up or node inquiry, and view the reduction tree from different perspectives. However, recursive subtree request is not allowed. That is, the user can request a subtree only in the main tree.

How to operate VISTA

In order to use VISTA from a unix shell, do the following:

- type *vista -tracefile -sourcepgm*, where *sourcepgm* is the file name of the source program.
- wait until the execution graph (PE graph) is displayed on the screen.
- then the user can select the function he or she wants.

Three mouse buttons are used to operate VISTA:

- right button: display menu
- center button: change the position of the tree
- left button: make an inquiry for a node, or to request a subtree after selecting the *display-subtree* menu item.

If the user presses the right button, eight menu items are displayed, as shown in Figure 4. The menu disappears after selection (except *zoom-up* and *zoom-down* menu items), and the appropriate graph is displayed. To make a node inquiry, the user puts the cursor on the node, and presses the right button. VISTA displays the information of the node nearest to the cursor position.

In order to request a subtree, the user first selects the *display-subtree* menu item. Then the user points to the node, and presses the left mouse button. VISTA displays the subtree with the node nearest to the cursor position as the new root node. It may take some time to view the subtree, if the trace is large.

Summary

In the first section of this chapter, the problems for visualizing parallel programs are discussed and the solutions for these problems are then given. In the rest of this chapter, the functions VISTA provides are explained. VISTA provides four different execution graphs, zoom-up or zoom-down, node-inquiry, and subtree display. Next chapter explains how these functions are implemented with an example for each step.

CHAPTER IV

IMPLEMENTATION OF VISTA

This chapter describes the implementation of VISTA. I discuss how the main goals of VISTA were accomplished, and why the algorithms used are efficient. For example, the trace-processing time and the redisplay time of 15,419 trace records (524,248 bytes) for the Semigroup problem are 45 seconds and 4 seconds, respectively, on a SUN SPARC station. This short time allows users to frequently redisplay data in different views.

Overview

The main goal of VISTA is to give effective visual feedback to the user analyzing a program. In order to achieve this goal, VISTA displays a whole reduction tree in one window, with some condensations if needed. Also, VISTA enables a user to see the tree from different perspectives, such as PE, time, or procedure, and zoom-up different portions of the tree.

Since the tree is usually large for small-grain parallel programs (even after condensation), and the tree must be redisplayed if the user wants to see it from a different view (different perspectives, zooming, or position change), the tree management algorithm must be carefully implemented. Also, window space must be utilized effectively because for large graphs, space is valuable property. That is, effective tree handling and window space utilization are the key factors in implementing VISTA. There are two kinds of condensations in VISTA: level condensing

and node condensing.

Definition: level condensing is a mapping from a tree T to a tree T' with the same ancestor and descendant relationships, such that a node n at level l in T is mapped into a node n' at level l' in T', where $l' = \lfloor l/c \rfloor$ and c is level-condensing ratio (a constant, defined in later sections). □

Definition: node condensing is the removal of all the descendants of the node n from the tree, if the allocated sector for n in the window space is less than one pixel. □

Since the reduction tree can be arbitrarily large, the problem of mapping the large tree to the limited window space must be resolved. There are several ways to resolve this problem. One method is to make a whole tree from the trace and reduce the tree whenever the tree is displayed. At first glance, this method appears reasonable because it needs only one pass over the trace file. However, this method causes two big problems in space and time. Since the tree can be arbitrarily large, it may be impossible to implement the tree in main memory. If the tree were implemented in secondary storage (i.e., disk), access would be too inefficient. Since the frequent redisplay of the tree is encouraged, as the user experiments with various views, it is not efficient to reduce the tree each time it is redisplayed.

Another method is to reduce the tree, by the ratio of the maximum dimension of the window, to the total number of trace records. With this method, the resulting tree will not correspond to user intuitions, because the trace is sequenced by time. This method is just a *contraction* of a tree, nothing else.

VISTA employs a more accurate abstraction, although it requires more I/Os. That is, two passes are required over the trace in VISTA: one pass to find the

height of the tree, and a second pass to create a tree with level condensing, if needed. Since the tree is condensed by level, the tree after condensation still keeps the original shape of the whole tree (even though the resulting tree is not scaled precisely from the original tree). This original shape allows user intuitions to carry over from analyzing tool view to tuning performance of actual programs.

The overall control for VISTA is shown in Figure 8. The trace management algorithm has five steps. Each of step is discussed in detail in the next sections. There are two inputs to the algorithm: a trace file and a source program. The first trace record is PE_{max}, the number of PEs allocated for the program. Figure 9 shows the format of the remaining trace records. After reading a trace record, the children count (the number of subgoals) is obtained from the source program using *clause#*. The children count is denoted as *ccnt* in the following algorithms. The basic management unit in the window space is assumed to be one pixel throughout this thesis.

In order to describe the algorithms more easily, consider a simple Quick Sort program in FGHC [15], an AND-parallel logic program. The sample program is shown in Figure 10. Figure 11 shows a trace executed by four PEs for this program, consisting of 23 records, plus the initial PE_{max} record.

First Pass: Calculating Tree Height

Definition: The level of a node is the path length from the root to the node. The level of the root is 0. An example is shown in Figure 12. □

Definition: The height of a tree is the longest path from the root to a leaf, plus one. The height of a node is the height of the tree minus the level of the node. In Figure 12, node A has height 3, node B height 2,

```
vista(trace file, source program) {
      1. First pass(trace file, source program);
            Calculate height of reduction tree, without actually creating tree.
            Trace file is read once.

      2. Second pass(trace file, source program);
            Level condensing is done if needed, and a tree is created.
            If this step is called by a subtree request, no level condensing is done.

      3. Weight calculation(tree);
            Transform tree into DAG, and calculate a weight for each node.

      4. Node allocation(DAG);
            Calculate the location of each node in polar coordinate system
                  centered at root node, using weights.

      5. Display graph(DAG);
            Set up graphic environment;
            Display default graph;
            repeat {
                  wait until user initiates some action;
                  display the graph the user wants
                  if (user wants a subtree at selected root)
                        second-pass() through node-allocation();
            until (user wants exit)
}           }
```

Figure 8: Overall Control Algorithm for VISTA

item	*type*	*range*	*explanation*
time	integer	$0 \sim \infty$	the time the reduction is executed, in *msec*
clause#	integer	$1 \sim$ max clause #	sequence # in source program by clause
pe	integer	$0 \sim$ (max-PE $-$ 1)	PE on which the reduction is executed
index	integer	$1 \sim \infty$	sequence number PE executed
ppe	integer	$0 \sim$ (max-PE $-$ 1)	parent PE (*ppe* of the root node is 0)
pindex	integer	$1 \sim \infty$	parent index (*pindex* of the root node is 0)

Figure 9: Trace Record Format

```
go(In, Out) :- qsort(In, Out-[]).

qsort([], R-A) :- R=A.
qsort([X|L], R0-R) :- split(L,X,L1,L2),
                      qsort(L1,R0-[X|R1]),
                      qsort(L2,R1-R).

split([],_,S,L) :-  S=[], L=[].
split([X|Xs],A,S,L) :- A>X | L=[X|L1], split(Xs,A,S,L1).
split([X|Xs],A,S,L) :- A=<X | S=[X|S1], split(Xs,A,S1,L).
```

Figure 10: Sample Program : Quick Sort in FGHC

4

time	*clause#*	*pe*	*index*	*ppe*	*pindex*
1992789729	1	0	1	0	0
1992807958	3	0	2	0	1
1992810675	5	0	3	0	2
1992812185	6	0	4	0	3
1992813240	3	3	1	0	2
1992815552	6	0	5	0	4
1992816664	3	2	1	0	2
1992818430	6	0	6	0	5
1992819512	6	2	2	2	1
1992820367	4	0	7	0	6
1992821448	3	1	1	2	1
1992823316	4	3	2	3	1
1992824409	5	2	3	2	2
1992825264	2	3	3	3	1
1992826331	2	1	2	3	1
1992827191	4	2	4	2	3
1992828039	3	3	4	2	1
1992829087	4	1	3	1	1
1992829939	4	3	5	3	4
1992831078	2	0	8	1	1
1992832208	2	2	5	3	4
1992833068	2	0	9	3	4
1992834202	2	1	4	1	1

Figure 11: Trace for Quick Sort Query "?- go([2,1,5,4,3],X)."

and node H height 1. The height of a tree is the height of the root. □

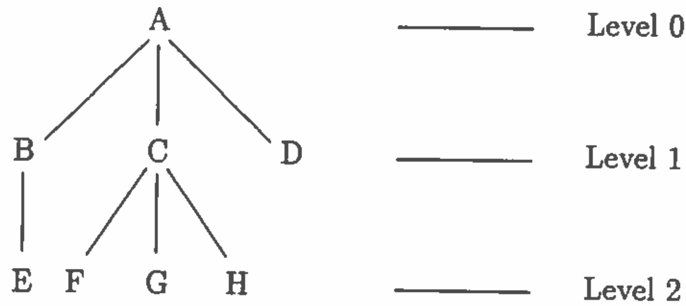


Figure 12: Tree Definition (Height of Tree = 3)

In the first pass, the tree height is computed from the trace, without creating a tree data structure. To calculate the height of the tree, VISTA uses a closed hash table [1], whose data structure is shown in Figure 14. The hash index (*hindex*), node pointer (*nodeptr*), and flag (*flag*) are not used in the first pass. The hash function h is defined as follows:

$$h(pe, index) = \frac{B}{\mathrm{PE}_{max}} \times pe + (\frac{B}{\mathrm{PE}_{max}} \bmod index),$$

where B is the total bucket count. Recall that *index* is trace index number. If a collision occurs, that is, the returned bucket already holds another element, a rehash strategy is used. VISTA uses the simplest rehash strategy, called linear hashing, where

$$h_i(pe, index) = (h(pe, index) + i) \bmod B.$$

The algorithm for the first pass is shown in Figure 13. If the reduction has children, *hash-producer* produces the slots of *ccnt* (i.e., puts *ccnt* in the hash table). When its child is processed, *hash-consumer* consumes one of parent slots (i.e.,

subtracts one from *ccnt*) and returns the parent level plus one. In this way, VISTA calculates the tree height without actually creating a tree. Figure 14 shows the hash table after processing the 15th record of the Quick Sort program, assuming that the total bucket count is 40. In this table, i represent the index of the hash table. The nodes (0,7), (3,3), and (1,2) are not in the hash table, because these reductions have no children. The pair represents (*pe*, *index*). Field $ccnt = 0$ in the hash table means that all of the nodes' children are processed, and that the bucket can be reused.

Second Pass: Level Condensing

Because there is a limitation to how much can be displayed in one window space, if the tree height is greater than the maximum window dimension, then some condensation must be done. In VISTA, this is called *level condensing.*

In order to calculate the level-condensing ratio, the maximum tree height to be displayed (i.e., limitation of the window space) is needed. The maximum tree height is calculated as $h_{max} = \left\lfloor \frac{w}{2d} \right\rfloor$, where w is the maximum window width, and d is the distance between two adjacent levels. With tree height h, level condensing is performed. If $h \leq h_{max}$, then no level condensing is needed. That is, the whole tree is displayed (unless node condensing is done in *display-graph* — see later sections). If $h > h_{max}$, level condensing is performed with the level condensing-ratio. The level condensing ratio c is calculated as follows :

$$
\begin{aligned}
c_0 &= \left\lfloor \frac{h}{h_{max}} \right\rfloor \\
t &= (h_{max} \times c_0) - ((h - h_{max} \times c_0) \times c_0) \\
c &= \begin{cases} c_0 & l \leq t \\ c_0 + 1 & l > t \end{cases}
\end{aligned}
$$

```
first-pass(trace file, source program) {
      read first and second trace record;            ; PEmax and 1st node record
      hash-producer(current record);
      tree-height := 0;
      while (read next record ≠ end of file) {
            level := hash-consumer(current record);
            if (level > tree-height)
                  tree-height := level;
                  ccnt := child-count(clause#);
            if (ccnt > 0)                            ; if the node has children
                  hash-producer(current record);
      }
      return (tree-height);
}


hash-producer(trace record) {
      i := h(pe, index);                             ; rehash if collision occurs
      hash-table[i] := pe, index, ccnt, level;
}


hash-consumer(trace record) {
      i := h(ppe, pindex);                           ; find parent hash index
      hash-table[i].ccnt := hash-table[i].ccnt − 1;  ; consume one child
      return (hash-table[i].level + 1);              ; return parent level + 1
}
```

Figure 13: Algorithm for Calculating Tree Height

i	*pe*	*index*	*level*	*ccnt*	*hindex*	*nodeptr*	*flag*
1	0	1	0	0			
2	0	2	1	0			
3	0	3	2	0			
4	0	4	3	0			
5	0	5	4	0			
6	0	6	5	0			
...							
11	1	1	3	3			
...							
21	2	1	2	1			
22	2	2	3	0			
23	2	3	4	1			
...							
31	3	1	2	0			
32	3	2	3	0			
...							

Figure 14: Hash Table after Processing 15th Record in First Pass

where t is the threshold and l is the level of the node.

For example, let $h_{max} = \frac{800}{4\times2} = 100$, where 800 is the maximum window width, and 4 is the level distance. The level distance four was obtained experimentally. With a level distance less than four, it is difficult for the user to visually distinguish between two levels. A level distance greater than four resulted in decreasing h_{max}, which is unacceptable because we wish to display as many nodes as possible. Consider a trace with a tree height $h = 300$,

$$\begin{aligned} c_0 &= \left\lfloor \frac{300}{100} \right\rfloor = 3 \\ t &= (100 \times 3) - ((300 - 100 \times 3) \times 3) = 300 \\ c &= 3 \end{aligned}$$

So every node is condensed by the ratio of three. However, if the tree height is 340,

then

$$
\begin{aligned}
c_0 &= \left\lfloor \frac{340}{100} \right\rfloor = 3 \\
t &= (100 \times 3) - ((340 - 100 \times 3) \times 3) = 180 \\
c &= \begin{cases} 3 & l \leq 180 \\ 4 & l > 180 \end{cases}
\end{aligned}
$$

So the first 180 levels are condensed by a ratio of three and the remaining 160 levels are condensed by a ratio of four. This condensation scheme results in putting more emphasis (space) on the levels closer to the root because earlier reductions are generally more "important" than later reductions. At this stage of the algorithm, the levels to be displayed are decided.

The remaining problem is how the tree should be represented physically in memory. Since the nodes do not all have the same number of children, the maximum out-degree is highly variable among programs. Since it is inefficient to allocate the maximum out-degree for the children pointers, a binary tree is used to emulate the logical tree, as illustrated in Figure 15. That is, this binary tree :

- retains branches from parents to leftmost sons, but deletes branches to other children.
- introduces branches from children to their next right sibling.

The algorithm for the second pass is shown in Figure 16. The closed hash table used in the first pass is used in the second pass again. In the second pass, fields *hindex* and *nodeptr* are used to make a tree and do level-condensing. Field *hindex* is the parent hash table index. Field *nodeptr* is the pointer to the node of the trace record if the trace record is not condensed, or the pointer to the parent

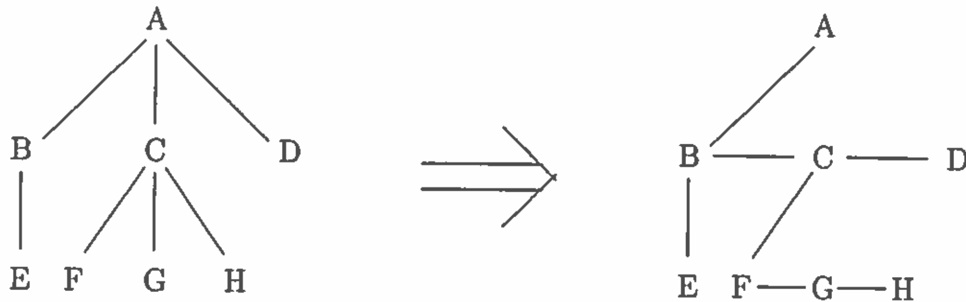


Figure 15: Transformation from Tree to Binary Tree

node if the trace record is condensed.

Figure 17 shows the hash table after processing the 15th record in the second pass, assuming that the total bucket count is 40 and that the level condensing ratio is two. Values A, C, M, E, N, and D in field *nodeptr* of the hash table represent pointers to the nodes of Figure 19.

Figure 18 shows the whole tree for the trace of Quick Sort program. In (s_k, p, j) of Figure 18, s, k, p, and j denote the goal name, the trace index number, *pe*, and *index*, respectively. For example, $(q_5, 3, 1)$ of node D denotes that q is *qsort*, 5 is the 5th trace record, $pe = 3$, and $index = 1$. Figure 19 shows the resulting tree after level condensing for Figure 18, assuming level-condensing ratio $c = 2$. As Figure 19 shows, the nodes in the odd levels are all condensed because if $(l \bmod c) \neq 0$, all nodes in the level l are condensed.

If the *second pass* is called by a subtree request, the trace index, which is the root of the subtree, is passed to this procedure. Therefore it would be possible to create a subtree by reading the trace record directly from the trace index. Reading from the beginning of the trace and just skipping to the trace index would have the same effect as the direct reading. However, neither method is employed in VISTA,

```
second-pass(trace record, source program) {
        c0 := tree-height / h_max;
        calculate threshold;
        read first and second record;             ; PE_max and first node record
        hash-producer(current record);
        while (read next record ≠ end of file) {
                level := hash-consumer(current record);
                if ((tree-height ≤ h_max) or (subtree-request))
                        level-condense-no(current record);
                else {
                        if (level ≤ threshold)
                                rem := level mod c0;
                        else rem := level mod (c0 + 1);
                        if (rem = 0)
                                level-condense-no(current record);
                        else level-condense-yes(current record);
}       }       }

level-condense-no(trace record) {
        node := make-node(trace record);
        hindex := find a parent hash table index;
        connect to the parent or siblings using parent node ptr;
        ccnt := child-count(clause#);
        if (ccnt > 0) {
                i := hash-producer();             ; pe,index,level,ccnt,hindex
                hash-table[i].nodeptr := node;    ; i : index of the node in hash-table
}       }

level-condense-yes(trace record) {
        hindex := find a parent hash table index;
        ccnt := child-count(clause#);
        if (ccnt > 0) {
                i := hash-producer();             ; pe,index,level,ccnt,hindex
                hash-table[i].nodeptr := hash-table[hindex].nodeptr;
                                                  ; copy parent nodeptr
                                                  ; i : index of the node in hash-table
}       }
```

Figure 16: Algorithm for Second Pass: Level Condensing

i	*pe*	*index*	*level*	*ccnt*	*hindex*	*nodeptr*	*flag*
1	0	1	0	0	0	A	
2	0	2	1	0	1	A	
3	0	3	2	0	2	C	
4	0	4	3	0	3	C	
5	0	5	4	0	4	M	
6	0	6	5	0	5	M	
...							
11	1	1	3	3	21	E	
...							
21	2	1	2	1	2	E	
22	2	2	3	0	21	E	
23	2	3	4	1	22	N	
...							
31	3	1	2	0	2	D	
32	3	2	3	0	31	D	
...							

Figure 17: Hash Table after Processing 15th Record in Second Pass

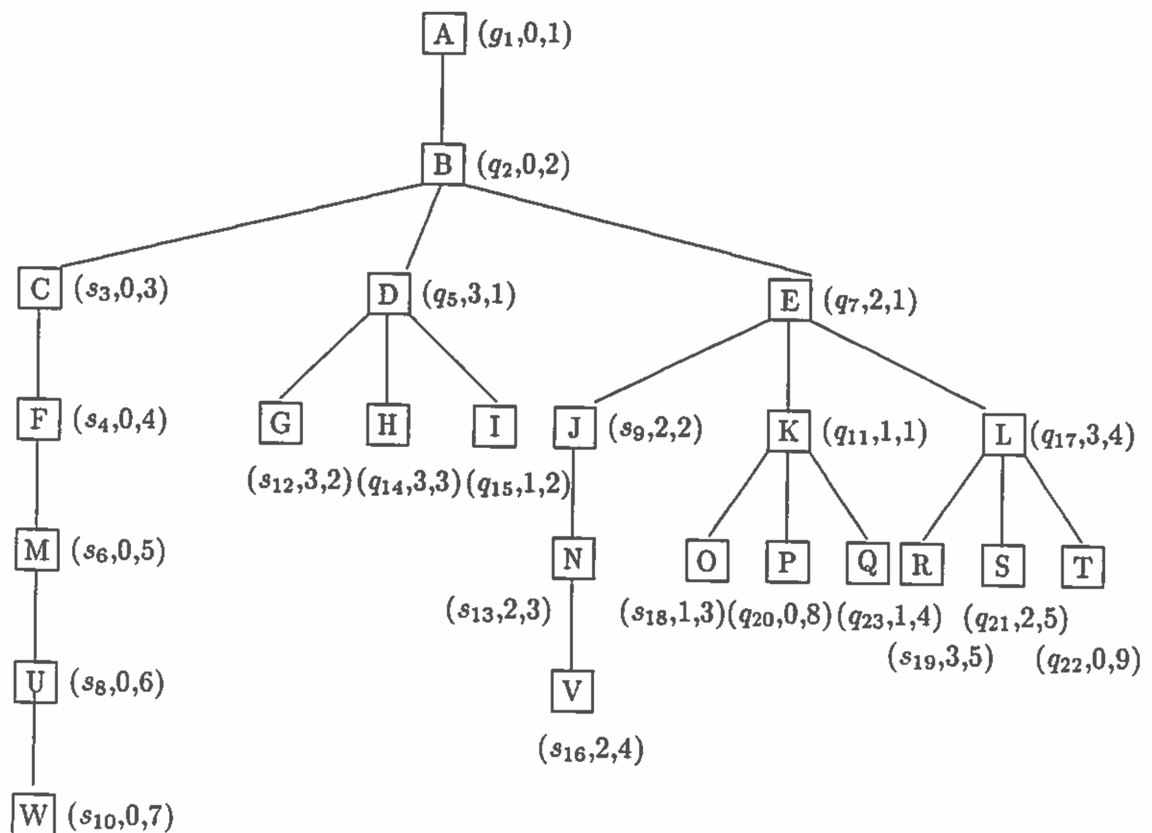


Figure 18: Whole Reduction Tree for Sample Program

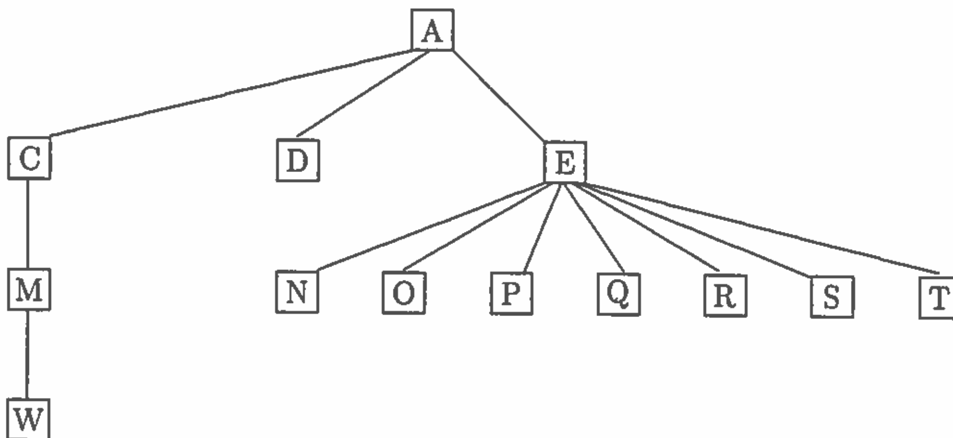


Figure 19: Reduction Tree After Level Condensing for Sample Program by 2:1

because the whole hash table must be searched for the nodes not in the subtree. Instead, VISTA reads the trace from the beginning, and sets the flag in the hash table *invalid* for the nodes not in the subtree.

Weight Calculation

As discussed in Chapter III, each node is displayed in the concentric circle with radius $r = d * l$, with a polygon or line, where d is the distance between two adjacent levels and l is the level of the node. The distance between any two adjacent levels is the same throughout the whole tree.

After level condensing has completed, the main problem is how the nodes in each level are allocated to the corresponding concentric circle. The simplest way would be to allocate the nodes evenly around the circle. However, this scheme is not good because some polygons might overlap and the shape might be difficult to understand. In order to avoid this from happening and to utilize the window space efficiently, a *sector* is allocated to each node depending on the weight of the node.

The node and its children are then put within the range of the sector only.

Before calculating the weight, a *level-connection* transformation is made in order to traverse the tree effectively. Given a graph $G(V,E)$ with $|V| = n$ and $|E| = e$, a conventional recursive DFS (depth-first-search) or BFS (breadth-first-search) takes $O(e)$ time [1]. However, the constant is so high that it is too inefficient to implement with recursive DFS or BFS for a large graph. Even though the recursion can be avoided with a stack for DFS and a queue for BFS, the overhead is still high, especially in space complexity. Thus I chose to transform the tree into a more efficient data structure, as shown in Figure 20. Figure 20 represents the reduction tree after level-connection for Figure 18 of Quick Sort, assuming no level condensing is done. The tree is implemented with a binary tree, as discussed in the second pass. In order to traverse a tree by the level, a *cousin pointer* (represented as "..." in Figure 20) is introduced.

This level-connection transformation can not be done in the second pass because at that time, the nodes on the same level are not necessarily processed in order. After the transformation, the array level-start-node[i] points to the first node in each level i and all the nodes in the same level are connected. After level-connection, the tree (actually, a DAG, or Direct Acyclic Graph) can be processed from any level. When the subtree is requested or a node inquiry is made, the tree is searched from the nearest level to the cursor position.

The weight w for each node is defined as the sum of its child weights plus the height of the node. In this definition, more weight is put on nodes closer to the root, because the closer the node is to the root, the fewer nodes the corresponding circle can contain. Figure 21 shows an example of the weight calculation for the Quick Sort program. For example, the weight of the node in level 2 in Figure 21 is 78,

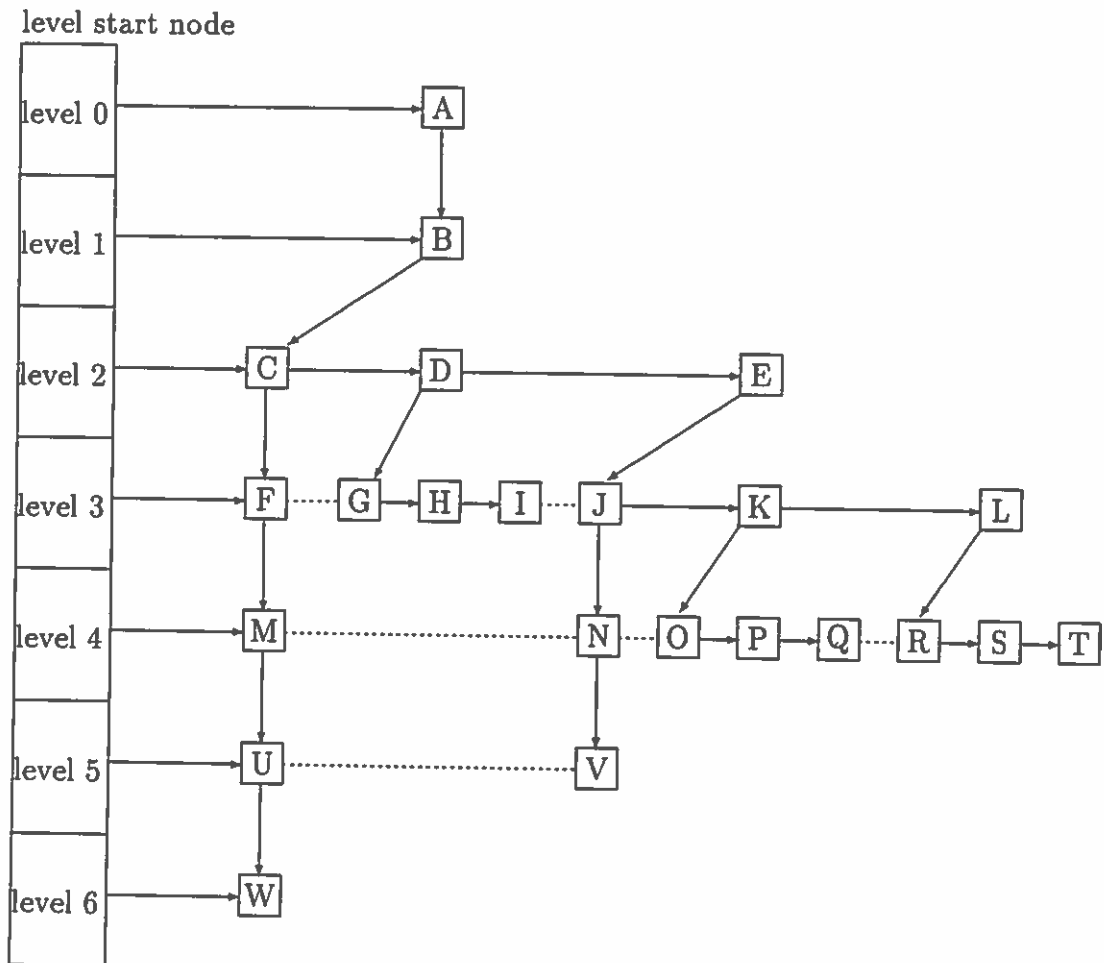


Figure 20: Reduction Tree After Level Connection for Sample Program

which is calculated from 72 (the sum of its child weights) plus 6 (the height of the node).

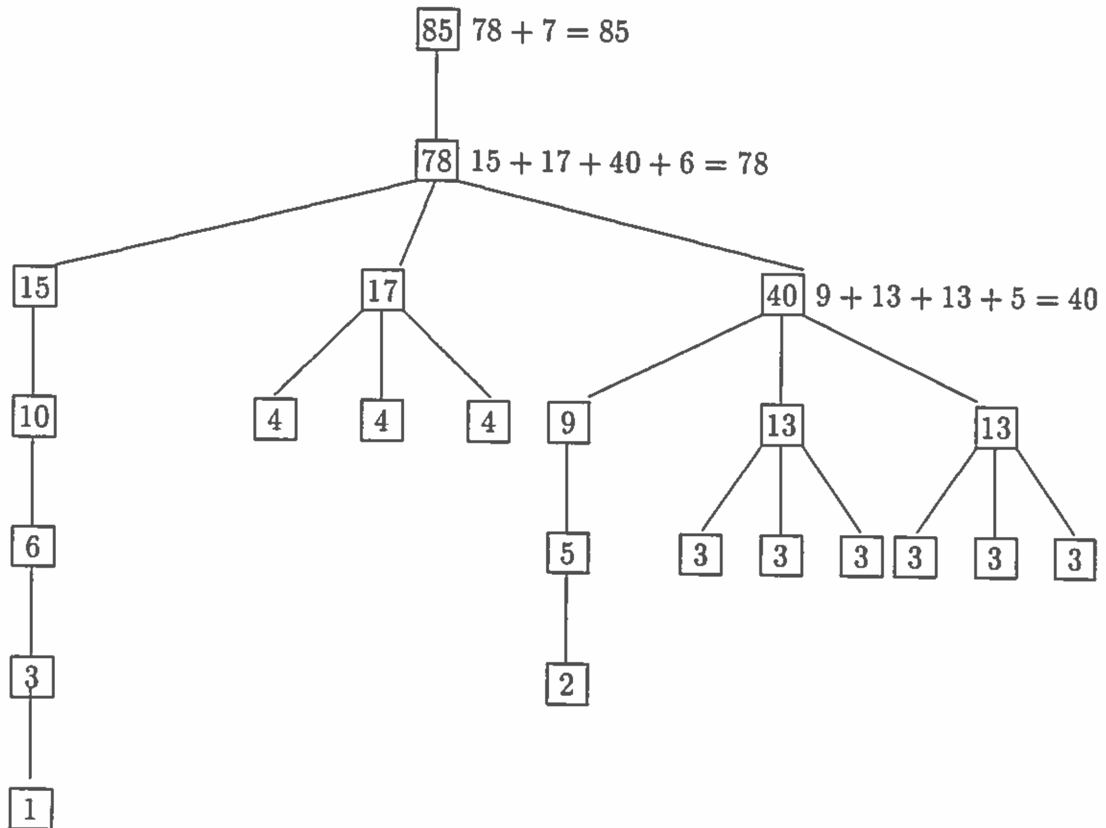


Figure 21: Weight Calculation for Sample Program

The algorithm for the weight calculation is given in Figure 22. As discussed above, before calculating a weight for each node, the level-connection transformation (*connect-level*) is made. In procedure *connect-level*, level is connected from level 0 by the level. In each level, the first node in the next level is found and then put in level-start-node[*i*]. To connect all the nodes in the same level, the rightmost child points to the leftmost child of the next node with children by a cousin pointer. In procedure *weight-calculation*, the weight for each node is calculated from the bottom level by the level, because each node needs its children weight to compute its weight.

```
weight-calculation() {
        connect-level();
        for (i = tree-height; i ≥ 0; i := i − 1) {
                node := level-start-node[i];
                while (node ≠ NULL) {                ; reach end of level
                        node→weight := height of the node + sum of child weights;
                        node := node→next;
}       }       }


connect-level() {
        level-start-node[0] = root-node;
        for (i = 0; i ≤ tree-height; i := i + 1) {
                node := level-start-node[i];
                while (node ≠ NULL) {                ; reach end of level
                        if (node has children) {
                                if (the child is the first node in the level)
                                        level-start-node[i+1] := node→child;
                                connect right-most child to left-most child of the next node
                                        with children by cousin pointer
                        }
                        node = node→next;
}       }       }
```

Figure 22: Algorithm for Weight Calculation

Node Allocation

After weight calculations, a sector is allocated to each node depending on its weight. The sector is defined as the subset of the circle within which a node can be put. The sector is represented by a pair (s, a), where s and a are the starting degree and the allocation degree of the node, respectively. The sector of the root is defined as (0, 360). The starting degree s for each node is calculated as follows:

$$s = lt_s + lt_a,$$

where lt_s and lt_a are the starting degree and the allocation degree of the left twin (if the node is the leftmost child, lt_s and lt_a are the starting degree of its parent and zero), respectively. The allocation degree a for each node is calculated as follows:

$$a = \frac{w}{T_w} \times p_a,$$

where w, T_w, and p_a are the weight of the node, the total weight of all its twins, the allocation degree of its parent. Figure 23 shows the sectors of the nodes for Figure 21. For example, the sector of node D is calculated as follows: $s = lt_s + lt_a = 0 + 75 = 75$ and $a = \frac{w}{T_w} \times p_a = \frac{17}{72} \times 360 = 85$. The sector of node J is calculated as follows: $s = lt_s + lt_a = 160 + 0 = 160$ and $a = \frac{w}{T_w} \times p_a = \frac{9}{35} \times 200 = 51$.

The exact position for each node in the window space isn't calculated until the tree is displayed, since the size and the center of the tree may be changed. The exact position (x, y) of a node in the window is calculated in the next step by the following equation :

$$x = d * l * cos(s + a/2)$$

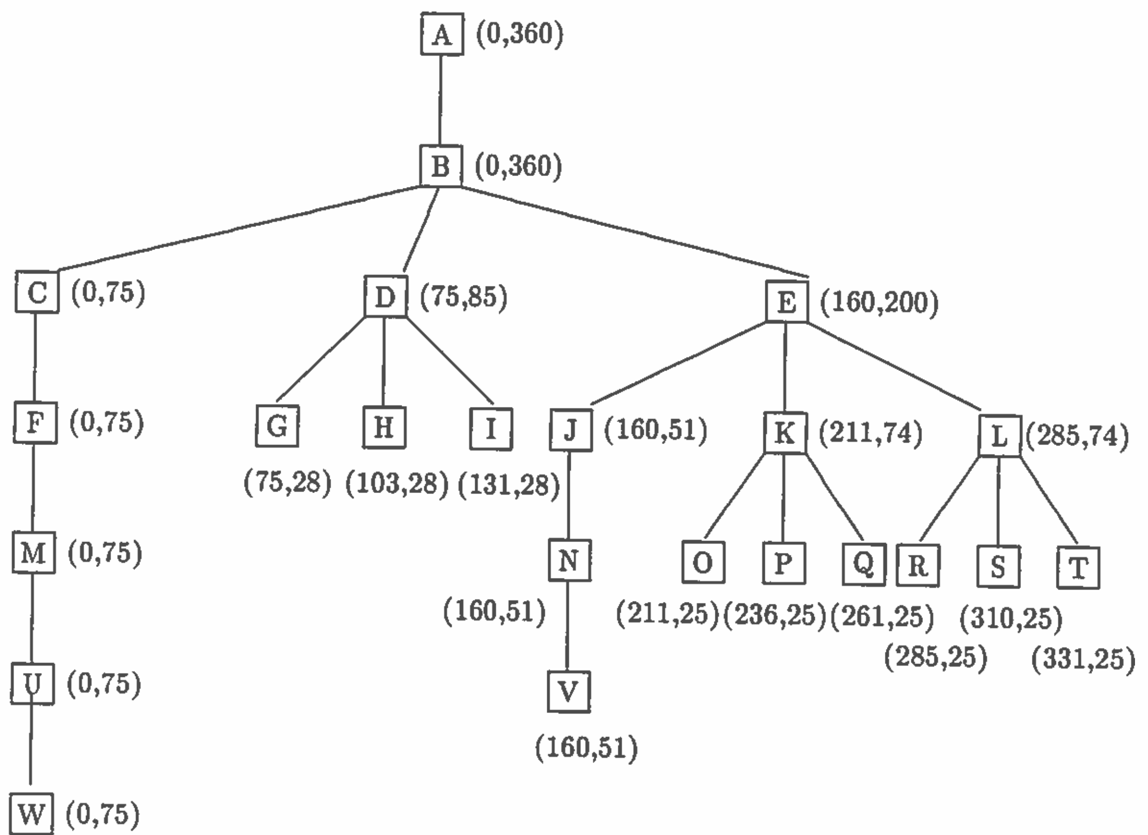


Figure 23: Node Allocation for Sample Program

$$y = d * l * sin(s + a/2)$$

where, d is the level distance, l is the level of the node, s is the start degree and a is allocation degree. The algorithm for the node allocation is shown in Figure 24. As Figure 24 shows, the sector for each node is calculated from level 0 (root) by the level because each node needs the allocation degree and starting degree of its parent.

```
node-allocation() {
      root-node→alloc-degree = 360;
      root-node→start-degree = 0;
      for (i = 0; i < tree-height; i := i + 1) {
            node := level-start-node[i];
            while (node ≠ NULL) {                    ; reach end of level
                  ad := node→alloc-degree;
                  sd := node→start-degree;
                  tw := node→weight − height of the node;
                                                     ; tw = sum of child weights
                  child := node→child;
                  while (child ≠ NULL) {
                        child→alloc-degree := child→weight / tw × ad;
                        child→start-degree := sd;
                        sd := sd + child→alloc-degree;
                        child := next-child();
}     }     }     }
```

Figure 24: Algorithm for Node Allocation

Display Graph : X–Window Interface

After the previous steps, the execution graph is ready for display. The graphic is drawn in the X–Window system [11]. X–Window provides windows on a bit-mapped terminal by the server-client model, with the facility of easy portability. The subsystem that controls the display and input devices is named the server.

At the other end are clients— programs written using X-lib to interface with the X-protocol. Therefore, VISTA is a client of the X–Window System.

After setting up the window environment (i.e., connecting to the server, loading a font, creating windows, etc.), the execution graph from the PE view is displayed. The tool then waits until the user initiates some action. Whatever the user does is controlled in this step: for example, window size change, window position change, cursor move, or button press.

If the user action concerns window attributes (size or position) or tree attributes (zooming or center change), VISTA displays the current tree type (PE, time, procedure) in proportion to the window size. If the action is about the tree type, the requested tree is displayed. If the action is an inquiry for a node, VISTA first finds the node which is the nearest to the cursor position. The information about the node is then displayed. If the action is about a subtree request, the following steps are executed: First, the current main tree status is saved. Second, the nearest node to the cursor position is found. Then the node becomes the root of the subtree. Third, the second-pass through node allocation are performed. Last, the PE Graph is displayed.

When drawing the graphic, if the sector calculated is less than one pixel, node condensing is done. That is, the node and all its children are not displayed.

The algorithms for the display-graph (main routine) and the draw-graphics (sub-routine) are shown in Figures 25 and 26, respectively.

Summary

VISTA is implemented with a five step algorithm. In the first step *(first pass)*, the height of the reduction tree is calculated without creating a tree. In the second

```
display-graph() {
        set up graphic environment;                     ; X-Window
        make color-map;
        draw-graphics();                                ; display default graph (PE)
        repeat {
                wait until user does some action;
                if (input is from main window) {
                        if (left button is pressed)
                                if (subtree-request = YES) {
                                        subtree-request := NO;
                                        save current status;
                                        node := find-node(); ; find the nearest node to cursor
                                        perform second-pass through node-allocation;
                                        draw-graphics();
                                } else {
                                        node = find-node();
                                        display node by the text format;
                        } else if (center-button is pressed) {
                                move center position;
                                draw-graphics();
                        } else                                  ; right button is pressed
                                display menu window;
                } else {                                        ; input is from menu window
                        if (request is about view-kind) {
                                view := request-view;           ; PE, time, proc., PE+time
                                draw-graphics();
                        } else if (zoom-request) {
                                zoom := zoom-request;           ; zoom-up, zoom-down
                                draw-graphics();
                        } else {                                ; tree-kind-change request
                                if (subtree-request)
                                        subtree-request := YES;
                                else {                          ; main-tree request
                                        release subtree;
                                        restore main-tree status;
                                        draw-graphics();
                }       }       }
        } until (user wants exit);
}
```

Figure 25: Algorithm for Display Graph (Main Routine)

```
draw-graphics() {
      calculate level-distance;
      for (i:=0; i<tree-height; i:=i+1) {
            node := level-start-node[i];
            perimeter := 2 π (level-distance) (i+1);
            while (node ≠ NULL) {            ; not end of level
                  sector := perimeter * node→adegree / 360;
                  if (sector < 1)
                        node-condensing;      ; make node and all its descendents invalid
                  else {
                        set color depending on kind (PE, time, procedure);
                        calculate positions (x,y) for the node and its children;
                        draw point for n=0, line for n=1 or polygon for n>1,
                              where n = the number of children
                  }
                        node := node→next;
}     }     }
```

Figure 26: Algorithm for Draw Graph (Sub Routine)

step *(second pass)*, the reduction tree is made while doing level-condensing if needed. In the third step *(weight calculation)*, the tree is transformed into a more efficient data structure (to traverse the tree efficiently), and the weight for each node is calculated. In the fourth step *(node allocation)*, the position for each node in the concentric circles is calculated using the weight of the node. In the last step *(display graph)*, the execution graph is displayed in an X-Window.

CHAPTER V

ANALYSIS OF PARALLEL PROGRAMS USING VISTA

In this chapter, I give several examples of how VISTA can be used to improve one's understanding both of parallel programs, and of the system that provides the parallel execution environment.

The most difficult problem in evaluating parallel programs is knowing how much parallelism the program has, and what portion of the parallelism is used in execution. The easiest way to represent the parallelism of a program would be by constructing a data-dependency graph. Figure 27 shows an example of a data-dependency graph, where each node, indicated by its number, represents a reduction. A path into the node means that the reduction at the other end of the path must be executed before the current node can be executed. In Figure 27, there are several possible execution orders ranging from purely sequential (e.g., 1, 2, 4, 3, 5, 6, 7) to parallel (e.g., 1, [2,3], [4,5,6], 7) where the numbers in brackets indicate possible parallel executions.

In VISTA, a data-dependency graph is not constructed because the data dependency graph with a reduction tree might result in losing the shape of the reduction tree. Instead, VISTA provides two kinds of graphs to analyze parallelism: the PE Graph and the Time Graph. With these graphs, characteristics of parallelism, such as load balancing among processors, speedup, and processor scheduling, can be analyzed to some extent.

In VISTA, the color spectrum from blue to magenta represents the complete

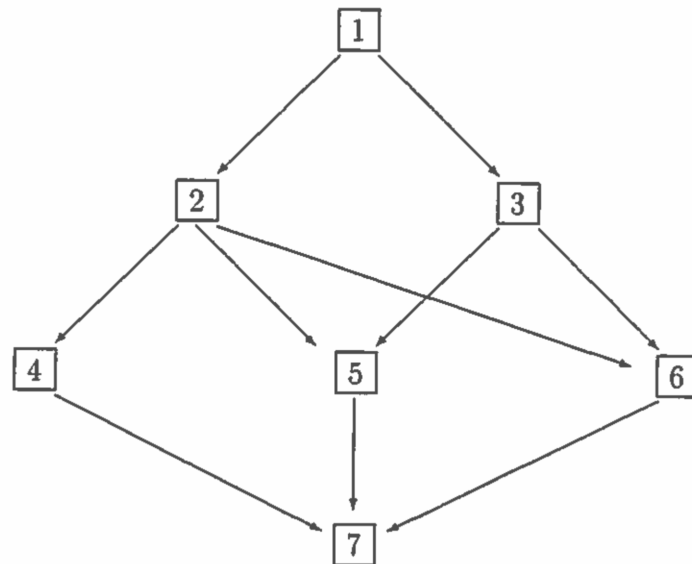


Figure 27: Example of Data Dependency Graph

execution time. Because there are not many distinct colors in this range (though it is possible to generate a huge number of colors even in this spectrum, human eyes can't tell the difference between many of the colors), the same color in the Time Graph does not necessarily represent the same time. If some nodes are represented with the same color within the Time Graph, and by the same PE color within the PE Graph (i.e, all the nodes are executed by the same PE), it means that the reductions were executed sequentially. This problem (i.e., the same color does not necessarily represent the same time in Time Graph) can be overcome to some extent with a subtree display (see Case Study 2: Semigroup Problem of this chapter).

In experimenting with parallel logic programs using VISTA, I have found a number of approaches useful for understanding the results of the experiments. The experiments consisted of a set of execution runs on a multiprocessor (here a 26 processor Sequent Symmetry [10]) and involved both modifying the benchmarks and varying the numbers of processors. I kept examples (the number of PEs, program,

and data size) small so that the explanation is clear. All the example programs tested are written in FGHC [15].

Case Study 1: Pascal's Triangle Problem

Pascal's Triangle is composed of the coefficients of $(x + y)^n$ for $n \geq 0$. The binomial coefficients of degree n are computed by adding successive pairs of coefficients of degree $n - 1$. A set of coefficients is defined as a *row* in Pascal's Triangle. The zero row is (1), the first row is (1, 1) and so on. In general, row n has $n + 1$ coefficients. In addition, the coefficients are symmetric around vertical axis of the triangle. Any row can easily be calculated from the previous row by adding pairs of successive coefficients. Figure 28 shows the program for Pascal's Triangle.

This program computes left-half rows of Pascal's Triangle in *makeRows/4* and then expands the final half row into a full row in *fillout/3*. Infinite precision ("bignum") addition must be used because the coefficient of row 100 has 30 decimal digits. The bignum definition used here is based on a radix $R = 10^5$. A bignum is a list $[X_0, X_1, \ldots, X_{j-1}, X_j]$, with the initial value $X_0 + X_1 + \ldots + X_{j-1}R^{j-1} + X_jR^j$. All illustrations in this section represent the execution graph of Pascal's Triangle for the query *?- go(35,Row).*

The easiest way to understand a program in VISTA is to view the reduction tree from the Procedure Graph. Figure 29 shows the reduction tree from the procedure perspective. This Procedure Graph is displayed without any condensation. The total number of nodes is 2,235. The height of the tree is 56. This interesting tree, with the shape of a snail or a spiral, is very unbalanced, but has some regular patterns. In Figure 29, most of the nodes represent procedures *makeRows* (cyan) and *makeRow* (green). Procedure *makeRows* calls itself recursively and calls

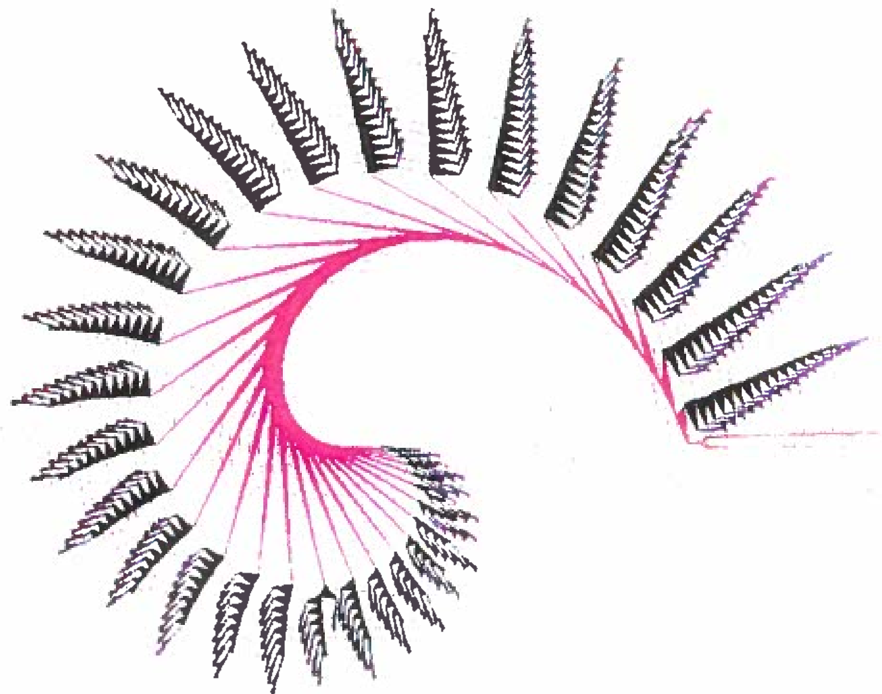


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

```
go(N,Row) :- N>1 | makeRows(1,N,[],Row).

makeRows(N,N,HalfRow,Row) :-
    Odd := N mod 2 |
    fillout(Odd,[[1,0]|HalfRow],Row).
makeRows(K,N,HalfRowK,Row) :- K1 := K+1, Odd := K mod 2 |
                    makeRow([[1,0]|HalfRowK],Odd,HalfRowK1),
                    makeRows(K1,N,HalfRowK1,Row).

makeRow([_],0,A) :- A=[].
makeRow([X],1,A) :- A=[S],bigPlus(S,X,X).
makeRow([X1,X2|Xs],Odd,A) :- A=[S|Ss],
                             bigPlus(S,X1,X2),
                             makeRow([X2|Xs],Odd,Ss).

fillout(0,HalfRow,Row) :- rev(HalfRow,[_|Rev]),
                          append(HalfRow,Rev,Row).
fillout(1,HalfRow,Row) :- rev(HalfRow,Rev),
                          append(HalfRow,Rev,Row).

bigPlus(A,X,Y) :- bigp(X,Y,A,0).

bigp([X|Xs],[Y|Ys],A,C) :- T := X+Y+C |
                           setc(T,A1,C1), A=[A1|As],
                           bigp(Xs,Ys,As,C1).
bigp([],[Y|Ys],A,C) :- T := Y+C |
                       setc(T,A1,C1), A=[A1|As],
                       bigp([],Ys,As,C1).
bigp([X|Xs],[],A,C) :- T := X+C |
                       setc(T,A1,C1), A=[A1|As],
                       bigp([],Xs,As,C1).
bigp([],[],A,0) :- A=[].
bigp([],[],A,C) :- C>0 | A=[C].

setc(T,A,C) :- T<100000 | C=0,A=T.
setc(T,A,C) :- T>=100000, R := T-100000 | C=1,A=R.

append([A|X],Y,Z):- Z=[A|Z1], append(X,Y,Z1).
append([],   Y,Z):- Z=Y.

rev(X,Y) :- rev(X,[],Y).
rev([A|X],Y,Z) :- rev(X,[A|Y],Z).
rev([],   Y,Z) :- Y=Z.
```

Figure 28: FGHC Program for Pascal's Triangle

makeRow. Procedure *makeRow* calls itself recursively and calls *bigPlus* (sky blue). One main branch (cyan) represents one row in Pascal's Triangle. The size of the subtree (i.e., one row) is increased by one for every two rows. This means that $\frac{n}{2}$ and $\frac{n}{2}+1$ rows have the same number of coefficients (because *makeRow* makes only the first half of the row, taking advantage of the symmetry of a row). The two lines at the east side represent the expansion of the final half row into a full row. Thus, the user can roughly understand a program from the Procedure Graph, even without looking into the source program. If the user is interested in the procedure name, this information can be accessed from the node inquiry.

To analyze the parallelism of the execution graph, an analysis of the load balancing among processors can be used first. Good load balancing among processors does not necessarily mean efficient exploitation of parallelism. However, without fair load balancing, full exploitation of parallelism cannot be achieved. That is, good load balancing is a necessary but not a sufficient condition for good speedups.

In VISTA, analyzing the load balancing is easy. A fair color distribution in the PE Graph represents good load balancing. Figure 30 (PE Graph) and Figure 31 (Time Graph) are the reduction trees executed by five PEs for Pascal's Triangle.

In Figure 30, all five colors are distributed almost evenly in the execution graph, representing good load balancing. To further analyze parallelism, the PE Graph and Time Graph are used. In Figure 31 (Time Graph), most clusters (one cluster represents one row) have a similar color pattern: the first node of each row is sky blue, the second, third, and fourth green, and so on. It means that some rows were executed in parallel. The maximum rows executed in parallel is limited at five, the number of PEs used. However, more than five rows appear as if they had been executed in parallel. As discussed above, the same color in the Time Graph does

not necessarily represent the same time. In other words, after the nodes in at most five rows were executed in parallel, the nodes in the other rows were executed.

Figure 32 shows the reduction tree (Time Graph) executed by *one* PE for Pascal's Triangle. In this graph, the nodes were always executed by a depth-first search. From this we can infer that depth-first search scheduling is used in the system, if there is no suspension. However, the nodes in the five PE Time Graph (Figure 31) appear to have been scheduled by a breadth-first search. Figure 30 also shows that some rows are not executed by the same PEs (i.e., there are some processor switches within the same row). If there were no suspensions (i.e., data-dependency in this example), there is no reason for executing by a breadth-first search or switching the processor, as shown in the one PE Time Graph. This means that there were some suspensions. Because each row needs the previous rows' coefficients to calculate the current coefficients, if the previous coefficients are not ready, the current node is suspended. The PE Graph shows this. As shown by the PE and Time graphs, the parallelism in this execution arises in the rows. Therefore, I hypothesize that more speedups can be gained, if PEs are added, upto the number of rows.

Case Study 2: Semigroup Problem

The Semigroup problem computes a large group of vectors from a small initial group of vectors, by repeated piecewise multiplication and comparison. Essentially, the closure under multiplication is computed for the initial vectors. Figures 33, 34 show the source program for semigroup problem, which uses a unbalanced binary hash tree containing the vectors. Here a vector is represented by K-T, where K is the hash key and T is a list of elements. The unbalanced binary hash tree code is

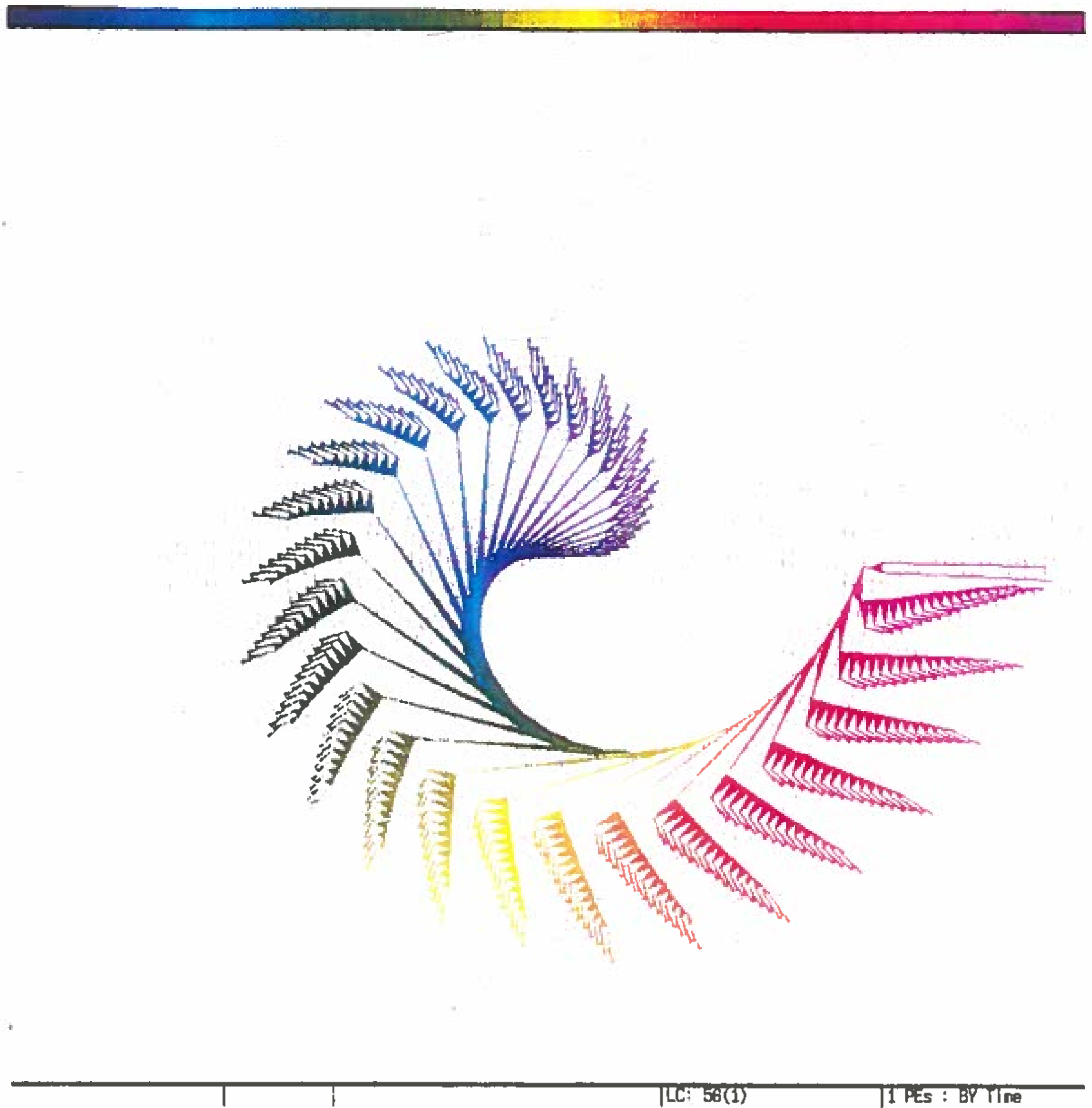


Figure 32: Time Graph for Pascal's Triangle (1 PE)

given in Figure 34. A node, containing a bucket B of vectors, is implemented by one of four types of processes: leaf (no children), right-child only, left-child only, and with both right and left children. A node process responds to various messages, instructing the node to insert and lookup vectors. For more details, see Tick [14].

Figure 35 (PE Graph) and Figure 36 (Time Graph) show the reduction tree executed by five PEs for the query *?- go(N)*. The total number of nodes in the whole reduction tree is 15,419 and the height of the tree is 174. In this experiment, the window size was 850 $\times$ 850 and the level distance was four. The maximum tree height to be displayed is calculated as $h_{max} = \lfloor \frac{850}{4} \times 2 \rfloor = 106$. Because the height of the whole reduction tree h is greater than the maximum tree height, level condensing is done. As discussed in Chapter 4, the level-condensing ratio c is calculated as follows:

$$\begin{aligned} c_0 &= \left\lfloor \frac{h}{h_{max}} \right\rfloor = \left\lfloor \frac{174}{106} \right\rfloor = 1 \\ t &= (h_{max} \times c_0) - ((h - h_{max} \times c_0) \times c_0) \\ &= (106 \times 1) - ((174 - 106 \times 1) \times 1) = 38 \\ c &= \begin{cases} c_0 & = 1 \quad l \leq t \\ c_0 + 1 & = 2 \quad l > t \end{cases} \end{aligned}$$

where t is the threshold and l is the level of the node. Therefore, the first 38 levels are not condensed, and the remaining 136 levels are condensed by the ratio of 2:1 (these numbers are displayed in the node window at the bottom Figure 35, for instance).

This example demonstrates some strong points of VISTA:

- the window space efficiency is very good in this example. If this reduction tree were represented by a conventional method (propagating from the top of the window), representation would be difficult. Even if possible, the window

```
go(Out) :- kernel(K), append(K,[end|R],T), initTree(T,R,Out).

initTree(T,R,Out) :- prefilter(T,T1,sawelement,R,Out),
                     t(T1,0,[]).

prefilter([end|T],T1,sawelement,R0,Out) :- T1=[end(R0,R1)|Ts],
                                           prefilter(T,Ts,sawend,R1,Out).
prefilter([end|_],T1,sawend,_,Out) :- T1=[out(Out,[])].
prefilter([X-Y|T],T1,_,R0,Out) :- T1=[m(X,Y,R0-R1)|Ts],
                                  prefilter(T,Ts,sawelement,R1,Out).

gen(X,S0-S1) :- kernel(K), g(K,X,S0,S1).

g([],_,S0,S1) :- S0=S1.
g([K|Ks],E,S,T) :- S=[P|Ps], g(Ks,E,Ps,T), mult(K,E,P).

insertBucket([],B,C,B1,E,S) :- B1=[C|B], gen(E,S).
insertBucket([C|_],B,C,B1,_,S0-S1) :- B1=B, S0=S1.
otherwise.
insertBucket([_|Cs],B,C,B1,E,S) :- insertBucket(Cs,B,C,B1,E,S).

send([],S0,S1) :- S0=S1.
send([T|Ts],S0,S2) :- S0=[T|S1], send(Ts,S1,S2).

append([A|X],Y,Z):- Z=[A|Z1], append(X,Y,Z1).
append([],    Y,Z):- Z=Y.

mult(_-X,Y,Out) :- Out=Key-R, mult(X,Y,R,0,Key,0).

mult([X|Xs],[Y|Ys],Out,InKey,OutKey,E) :- NewKey := E+InKey*3 |
                                          Out = [Z|Zs], m(X, Y, Z),
                                          mult(Xs,Ys,Zs,NewKey,OutKey,Z).
mult([], [], Z, InKey, OutKey, _) :- Z=[], OutKey=InKey.

m(2,1,Z):- Z=1. m(3,1,Z):- Z=1. m(4,1,Z):- Z=1. m(5,1,Z):- Z=1.
m(1,1,Z):- Z=1. m(2,2,Z):- Z=2. m(3,2,Z):- Z=1. m(4,2,Z):- Z=1.
m(5,2,Z):- Z=5. m(1,2,Z):- Z=1. m(2,3,Z):- Z=1. m(3,3,Z):- Z=3.
m(4,3,Z):- Z=4. m(5,3,Z):- Z=1. m(1,3,Z):- Z=1. m(2,4,Z):- Z=4.
m(3,4,Z):- Z=1. m(4,4,Z):- Z=1. m(5,4,Z):- Z=3. m(1,4,Z):- Z=1.
m(2,5,Z):- Z=1. m(3,5,Z):- Z=5. m(4,5,Z):- Z=2. m(5,5,Z):- Z=1.
m(1,5,Z):- Z=1.

kernel(K) :-
   K=[1833472791-[ 1,1,1,1,1,2,2,2,2,2,3,3,3,3,3,4,4,4,4,4,
                   5,5,5,5,5,3,3,3,3,3,5,5,5,5,5,4,4,4,4,4],
      -590019130-[ 1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,1,2,3,4,5,
                   1,2,3,4,5,1,2,3,4,5,1,3,2,4,5,1,2,3,4,5],
      1154844798-[ 1,2,3,5,4,1,2,3,5,4,1,2,3,5,4,1,2,3,5,4,
                   1,2,3,5,4,1,2,3,4,5,1,2,3,5,4,1,2,3,5,4]].
```

Figure 33: Part A of Semigroup Program

```
% leaf node (node with no child nodes)
% X,Y are keys, C is tuple, B is list of tuples (bucket)

t([],_,_) :- true.
t([m(Y,C,S) |T],X,B) :- X=:=Y | insertBucket(B,B,C,B1,C,S),t(T,X,B1).
t([m(Y,C,S) |T],X,B) :- X < Y | gen(C,S), tr(T,X,B,R), t(R,Y,[C]).
t([m(Y,C,S) |T],X,B) :- Y < X | gen(C,S), t(L,Y,[C]), tl(T,X,B,L).
t([out(S0,S)|T],X,B) :- send(B,S0,S), t(T,X,B).
t([end(S0,S)|T],X,B) :- S0=[end|S], t(T,X,B).

% node with right child only
tr([],_,_,R) :- R = [].
tr([m(Y,C,S) |T],X,B,R) :- X =:= Y | insertBucket(B,B,C,B1,C,S),
                                      tr(T,X,B1,R).
tr([m(Y,C,S) |T],X,B,R) :- X < Y | tr(T,X,B,R1),
                                    R = [m(Y,C,S)|R1].
tr([m(Y,C,S) |T],X,B,R) :- Y < X | gen(C,S), t(L,Y,[C]),
                                    tlr(T,X,B,L,R).
tr([out(S0,S)|T],X,B,R) :- send(B,S0,S1), tr(T,X,B,R1),
                           R = [out(S1,S)|R1].
tr([end(S0,S)|T],X,B,R) :- S0=[end|S], tr(T,X,B,R).

% node with left child only
tl([],_,_,L) :- L = [].
tl([m(Y,C,S) |T],X,B,L) :- X =:= Y | insertBucket(B,B,C,B1,C,S),
                                      tl(T,X,B1,L).
tl([m(Y,C,S) |T],X,B,L) :- X < Y | gen(C,S), tlr(T,X,B,L,R),
                                    t(R,Y,[C]).
tl([m(Y,C,S) |T],X,B,L) :- Y < X | L=[m(Y,C,S)|L1], tl(T,X,B,L1).
tl([out(S0,S)|T],X,B,L) :- L=[out(S0,S1)|L1], tl(T,X,B,L1),
                           send(B,S1,S).
tl([end(S0,S)|T],X,B,L) :- S0=[end|S], tl(T,X,B,L).

% node with both left and right children
tlr([],_,_,L,R) :-  L = [], R = [].
tlr([m(Y,C,S) |T],X,B,L,R) :- X =:= Y | insertBucket(B,B,C,B1,C,S),
                                         tlr(T,X,B1,L,R).
tlr([m(Y,C,S) |T],X,B,L,R) :- X < Y | tlr(T,X,B,L,R1),
                                       R = [m(Y,C,S)|R1].
tlr([m(Y,C,S) |T],X,B,L,R) :- Y < X | L = [m(Y,C,S)|L1],
                                       tlr(T,X,B,L1,R).
tlr([out(S0,S)|T],X,B,L,R) :- L=[out(S0,S1)|L1], tlr(T,X,B,L1,R1),
                              send(B,S1,S2), R=[out(S2,S)|R1].
tlr([end(S0,S)|T],X,B,L,R) :- S0=[end|S], tlr(T,X,B,L,R).
```

Figure 34: Part B of Semigroup Program: Unbalanced Binary Hash Tree

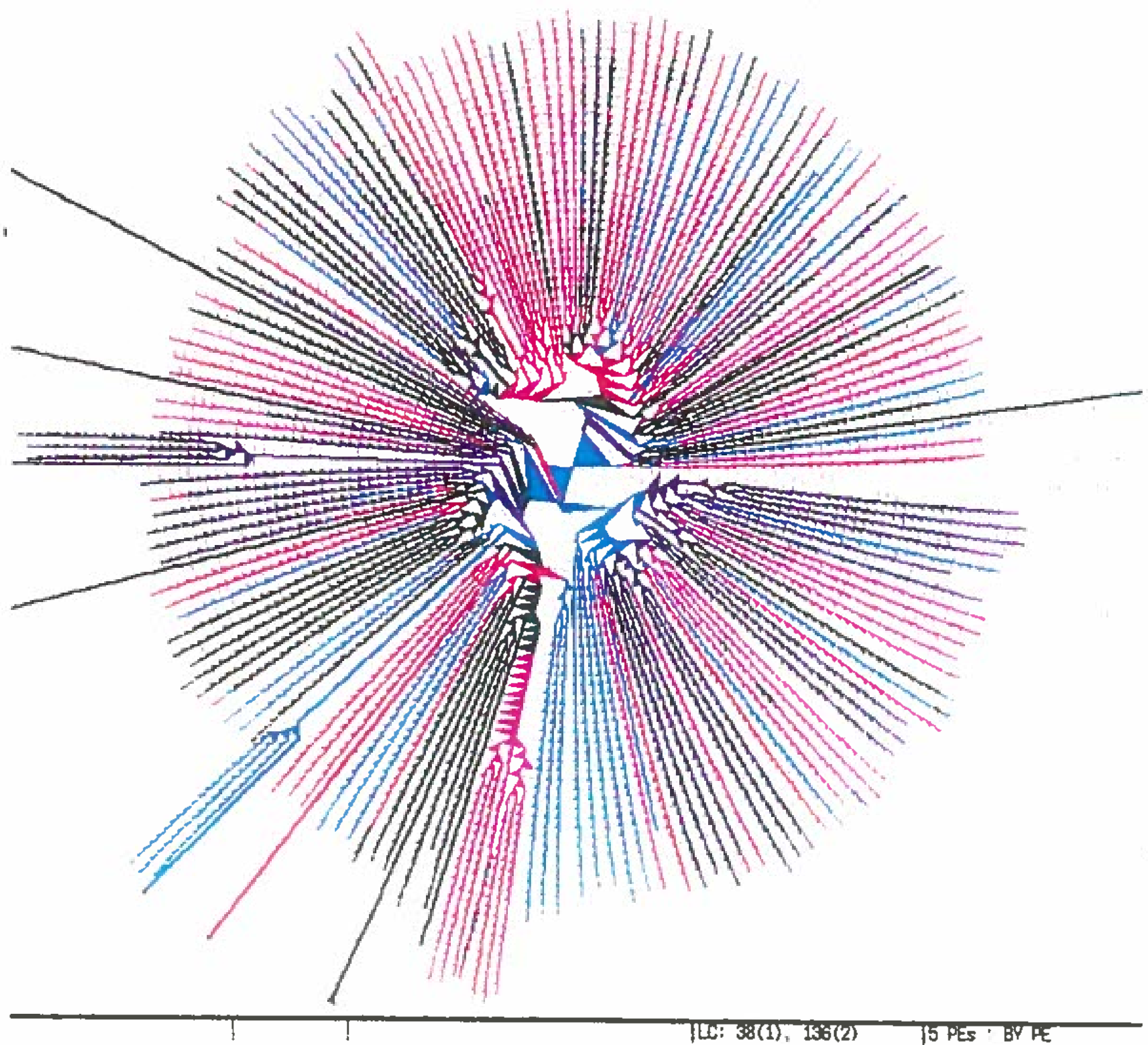


Figure 35: Execution Graph of Semigroup from PE View (5 PEs)

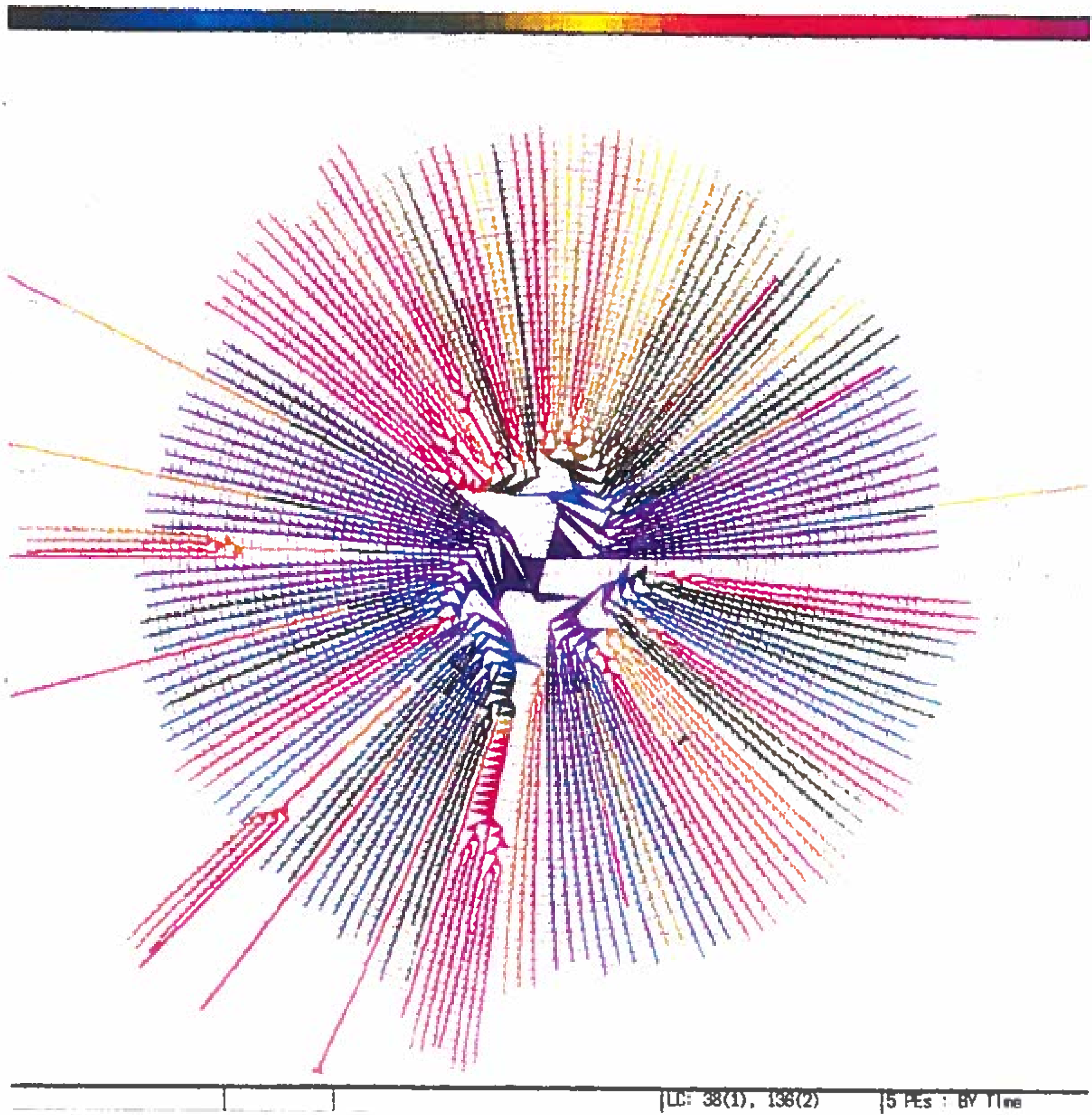


Figure 36: Execution Graph of Semigroup from Time View (5 PEs)

space efficiency would be poor.

- Even after level condensing, the tree keeps the shape of the original tree (the original tree is not shown, but the original tree can easily be inferred from this tree).

To understand the parallelism of the execution graph for the Semigroup, load balancing among processors is analyzed first. In Figure 35 (PE Graph), five colors are distributed evenly in the graph. Thus, load balancing is good in this execution. Figure 35 (PE Graph) shows that all reductions are scattered throughout the window with the shape of many *spokes.* Each spoke (also called a *line* in this example) represents a vector multiplication. As the graph shows, almost all lines were executed without a processor switch. This means that there were only a few suspensions in each line, if any. That is, the nodes in each line do not depend on the data of the other nodes except for their parent node. Lack of data-dependency between nodes can be confirmed from another execution graph: Figure 37 shows the reduction tree executed by *one* PE.

The color distribution of the single-PE graph is similar to that of Figure 36. This means that as soon as the first node of the new line is generated, both lines were executed in parallel until the end of the line without any suspension. The reason that the lines are not executed clockwise or anti-clockwise (as in the single-PE Time Graph of Pascal's Triangle) in Figure 37 is that there were some data dependencies in the initialization phase. That is, there were some suspensions in levels 2—5.

If the PE Graph (Figure 35) is viewed with the Time Graph (Figure 36), parallelism can be analyzed in more detail. The lines with the same colors in the Time Graph are represented by different colors in the PE Graph. This means that some lines were executed in parallel. However, there are also some lines with the

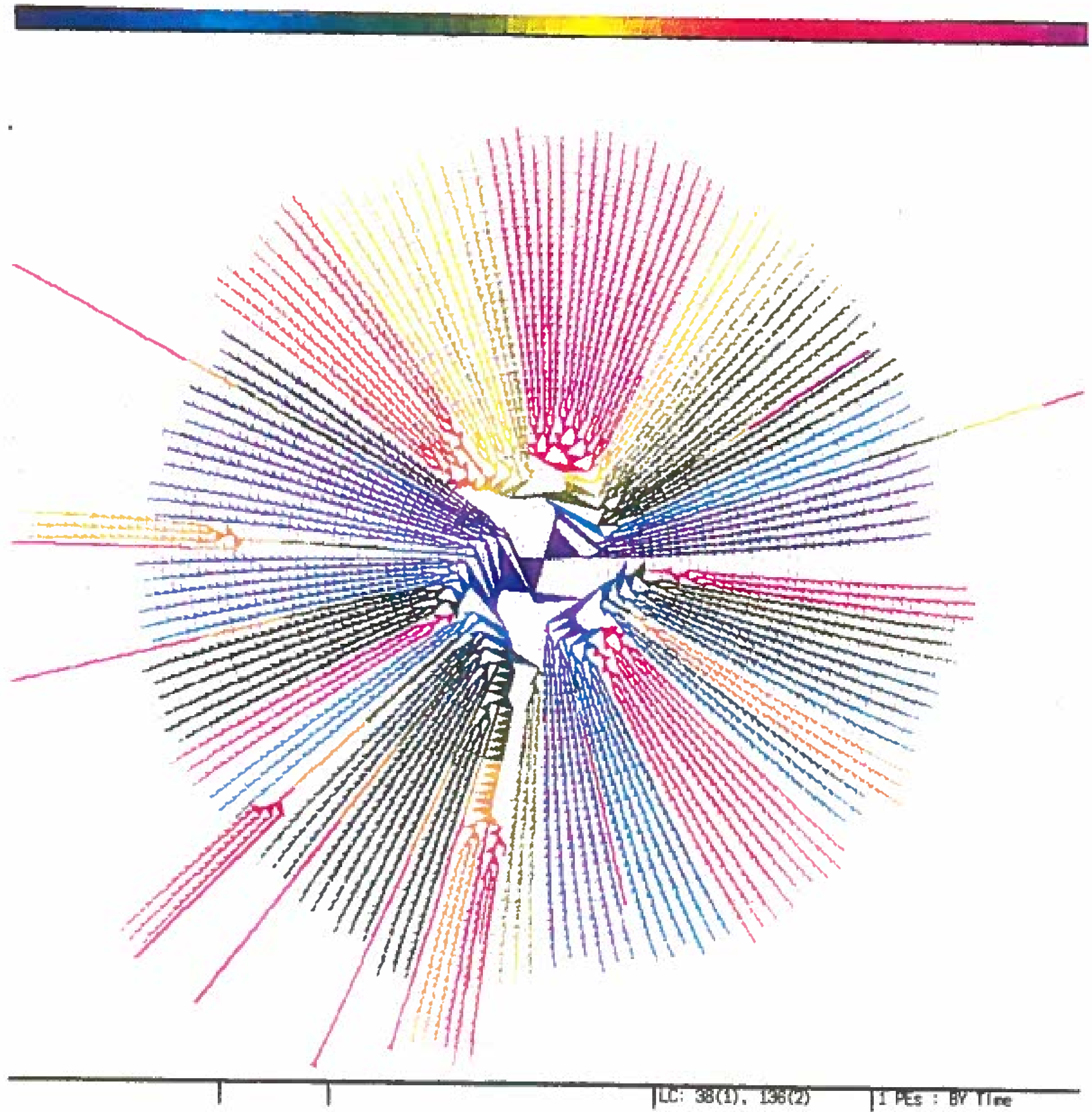


Figure 37: Execution Graph of Semigroup from Time View (1 PE)

same color in the PE Graph. This means that if more PEs are added in execution, greater speedup can be achieved.

Because the color spectrum from blue to magenta represents the complete time in VISTA, if the whole graph (i.e., without condensation) is large (i.e., the execution time is long), analyzing parallelism by the Time Graph is difficult. In that case, a subtree display can be useful. Figure 38 and Figure 39 show an example of subgraphs of Figure 35 and Figure 36, respectively. In the subgraph displays, the reduction tree is displayed without level condensing, and the time spectrum represents time *relative to* the subtree (i.e., not the whole execution time) as Figure 39 shows. Therefore, with the two subgraphs (PE Graph and Time Graph) by the way discussed above, more detailed parallelism can be analyzed.

Summary

To analyze parallel programs with VISTA, the user is encouraged to view several graphs. To understand a program itself without viewing parallelism, the Procedure Graph is used with node inquiry. To analyze characteristics of parallelism, the PE Graph and the Time Graph are used. If these graphs do not offer enough detail to analyze parallelism, the single-PE Time Graph can give further information. If the displayed graph is too densed to analyze a program, zoom-up or subtree-display can be useful. As the two example programs illustrate, window-space utilization in VISTA is very good. If conventional display methods were used, parallel program execution characteristics would be far more difficult to analyze.

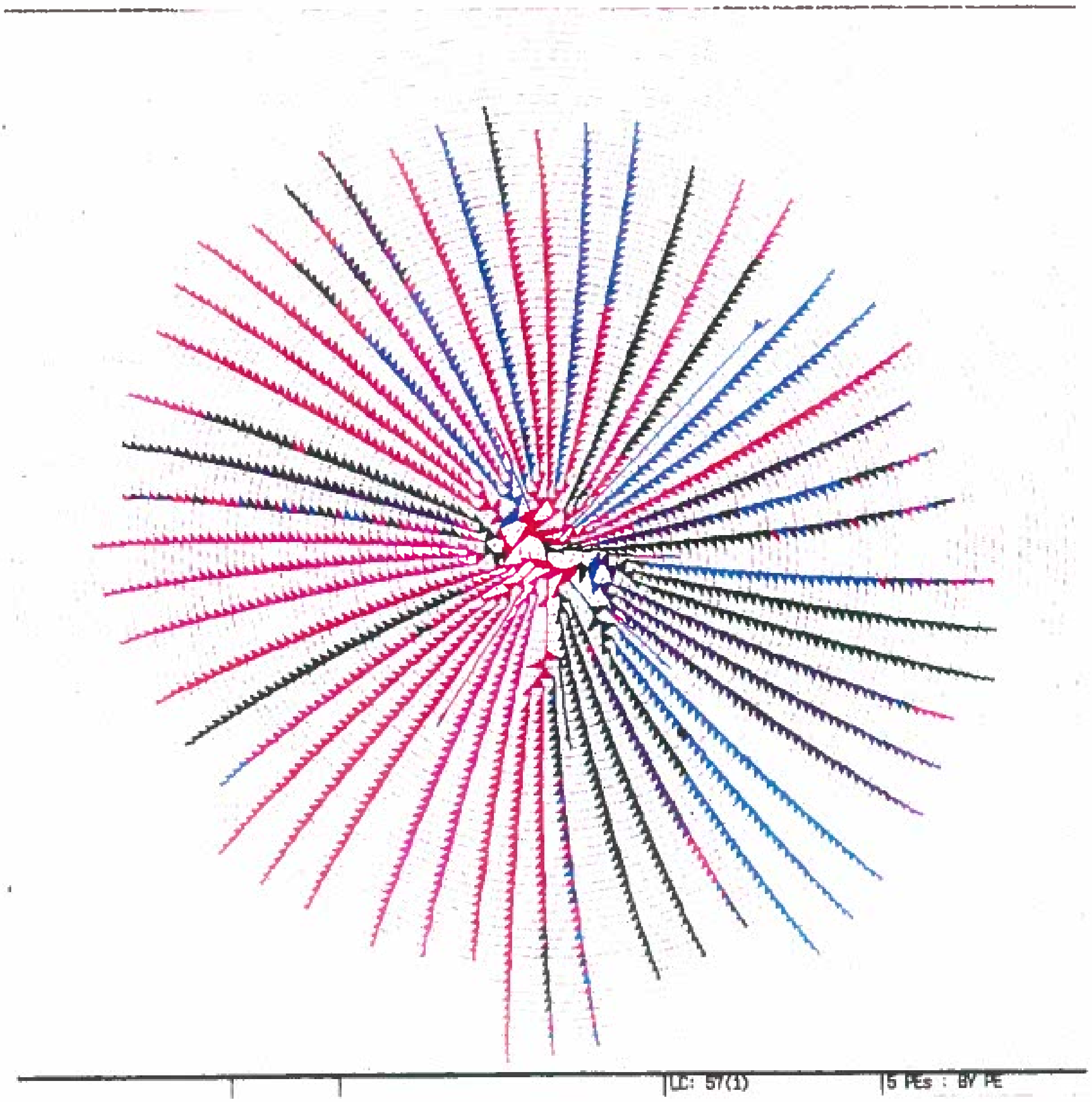


Figure 38: Subgraph of Semigroup from PE Perspective (5 PEs)

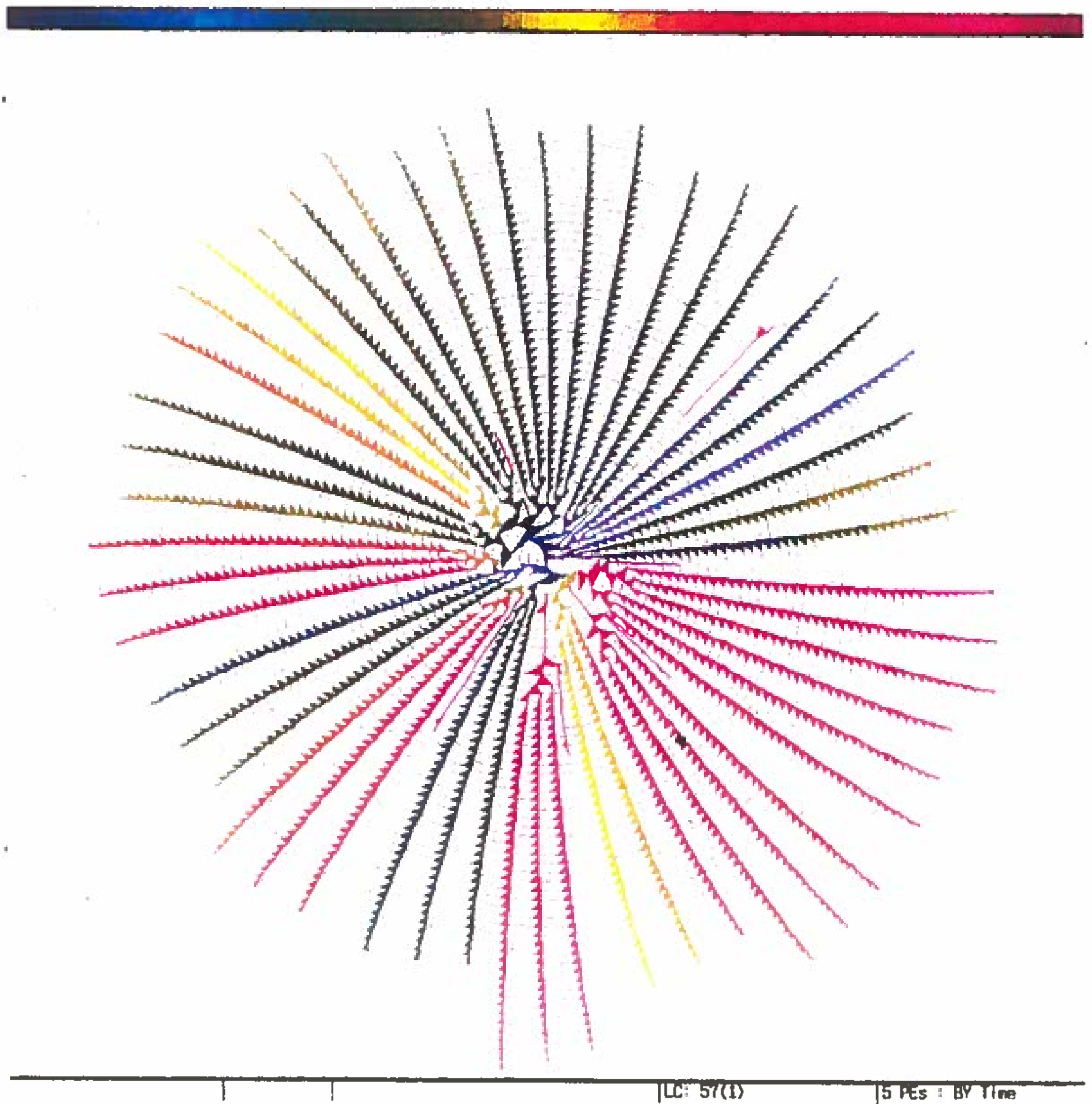


Figure 39: Subgraph of Semigroup from Time Perspective (5 PEs)

CHAPTER VI

CONCLUSIONS AND FUTURE RESEARCH

Visualizing the execution of parallel programs is difficult because low-level events do not reflect the programmer's view of the system. Program behavior is better understood at a higher level in terms of user-defined, abstract events. Perspective views enhance visualization, because they summarize user-defined, abstract events.

I have presented VISTA, a graphical tool for analyzing small-grain parallel programs. VISTA displays execution graphs from various perspectives (PE, time, PE–time, procedure). VISTA also provides some functions such as zoom-up, subtree display, and node inquiry to help users analyze parallel programs in more detail. The tool's use has resulted in insights both into the parallelism issues confronting the parallel programmer, and into the behavior of VISTA itself. My impression of using VISTA has been that VISTA made the understanding of what was occurring in parallel programs, both at the user and system levels, come much faster and with greater certainty.

Although VISTA provides parallel programmers with useful functions for understanding parallel programs, this tool also has drawbacks. First, VISTA does not show process communications. Therefore, it is difficult to understand a data dependency. Second, since the RGB spectrum is used to represent execution time, if the execution time is long, distinguishing is difficult, making analysis of parallelism difficult.

An interesting extension to VISTA is to incorporate an animation mechanism to make up for these drawbacks. Animation can enhance the understanding of parallelism, if a program is not executed by a large number of processors. Another possible extension to VISTA is to represent a data-dependency graph. If this graph is made during execution time, the run-time overhead might be very high. Therefore, the information for the data-dependency graph must be gathered at compile time as much as possible.

BIBLIOGRAPHY

[1] A. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, Reading MA, 1983.

[2] T. Disz and E. Lusk. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *International Symposium on Logic Programming*, pages 46–53. San Francisco, IEEE Computer Society, August 1987.

[3] T. Disz, E. Lusk, and R. Overbeek. Experiments with OR-Parallel Logic Programs. In *Fourth International Conference on Logic Programming*, pages 576–600. University of Melbourne, MIT Press, May 1987. Also available as ANL Technical Report TM-87.

[4] R. Evard and E. Lusk. A Visualization Tool for Parallel Languages. Research report, Argonne National Laboratory MCS, June 1990. unpublished draft.

[5] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[6] R. J. Fowler, T. J. LeBlanc, and J. M. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989.

[7] V. A. Gauarna, D. Gannon, D. Jablonowski, A. D. Malony, and Y. Gaur. Faust: An Integrated Approach to Parallel Programming. *IEEE SOFTWARE*, pages 20–27, July 1989.

[8] M. M. Gorlick and C. F. Kesselman. Timing Prolog Programs Without Clocks. In *International Symposium on Logic Programming*, pages 426–432. San Francisco, IEEE Computer Society, August 1987.

[9] A. A. Hough and J. E. Cuny. Pesrspective Views : A Technique for Enhancing Parallel Program Visualization. In *International Conference on Parallel Processing*, pages 124–132. Seattle, 1990.

[10] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems.* Prentice Hall, Englewood Cliffs, NJ, 2nd edition, 1989.

[11] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5:79–109, April 1986.

[12] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer*, 17(7):27–36, July 1984.

[13] D. Socha, M. M. Bailey, and D. Notkin. Voyeur: Graphical Views of Parallel Programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.

[14] E. Tick. *Parallel Logic Programming*. Logic Programming. MIT Press, Cambridge MA, 1991.

[15] K. Ueda. Guarded Horn Clauses. In E.Y. Shapiro, editor, *Concurrent Prolog: Collected Papers*, volume 1, pages 140–156. MIT Press, Cambridge MA, 1987.

[16] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025, 1983.

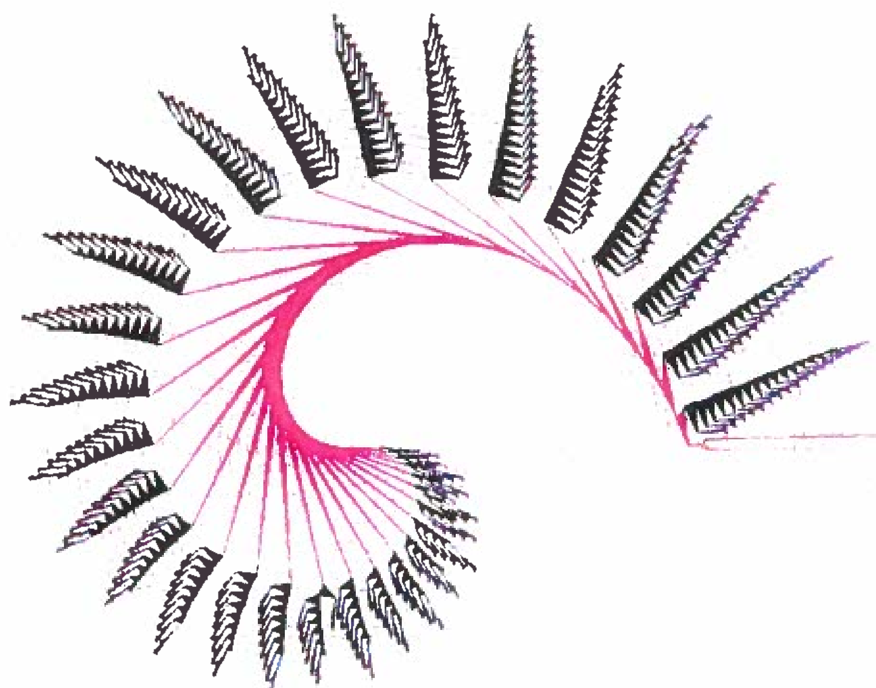


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

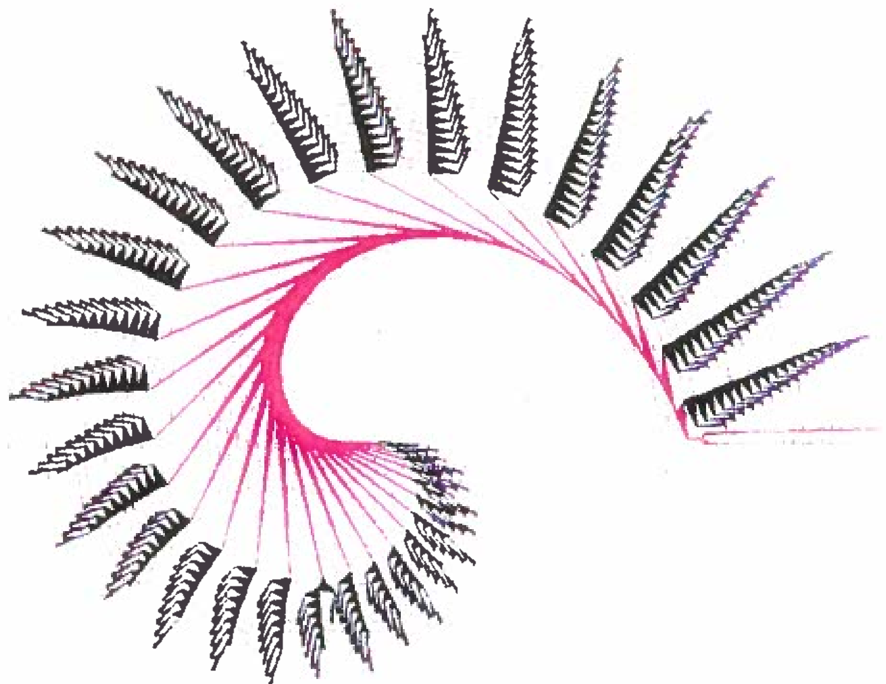


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

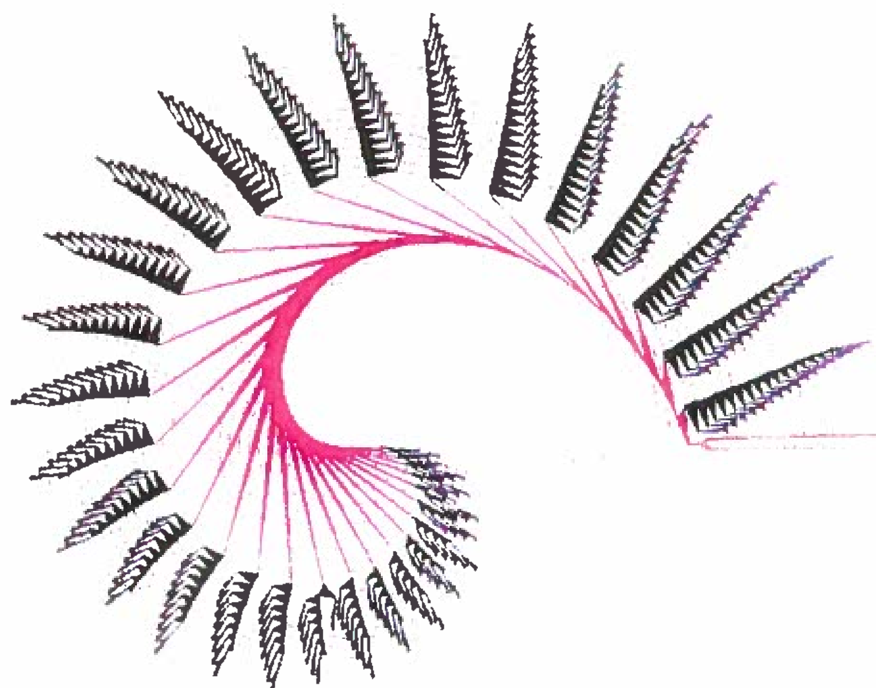


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

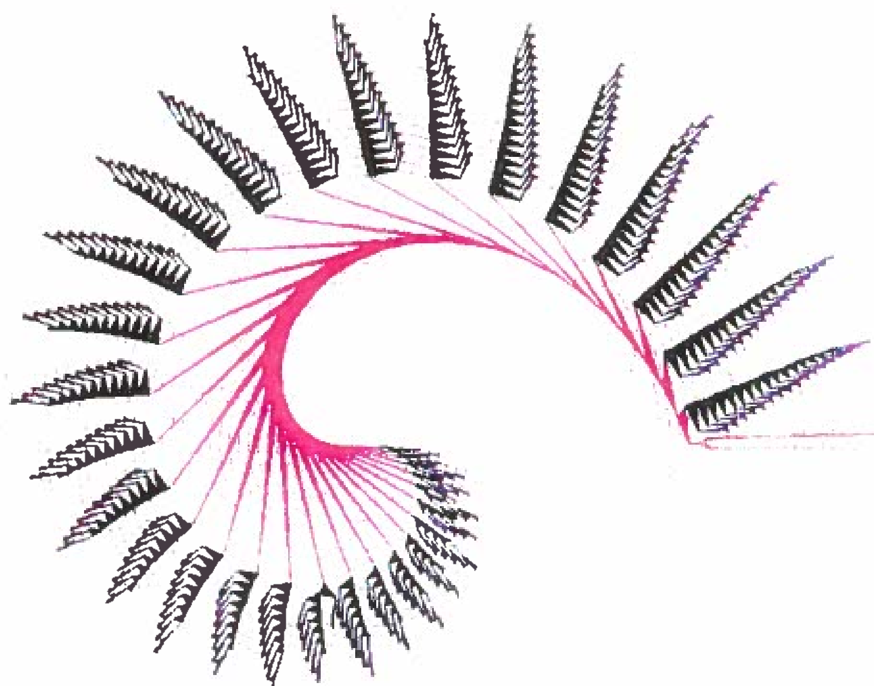


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

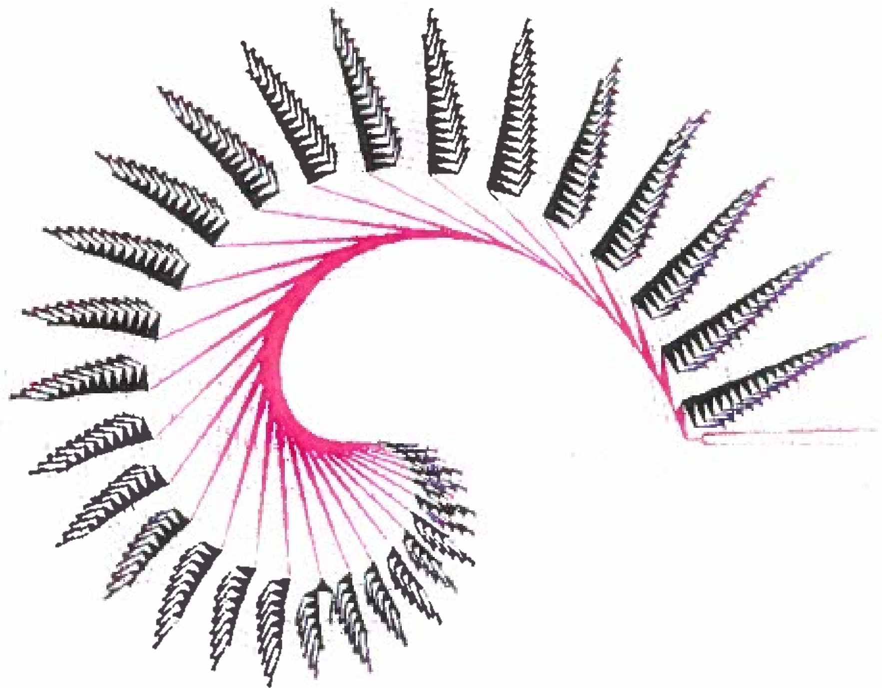


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

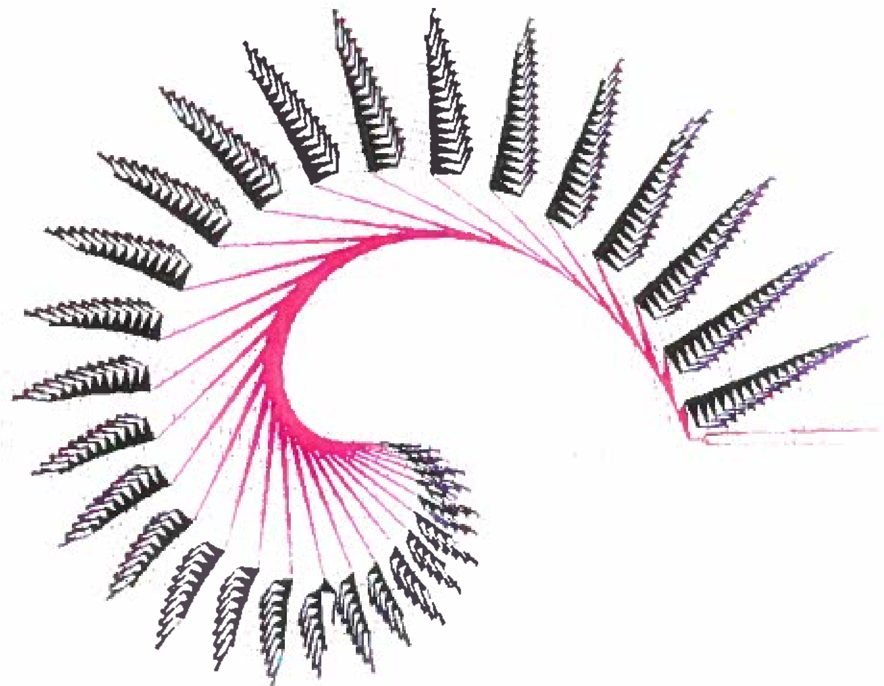


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

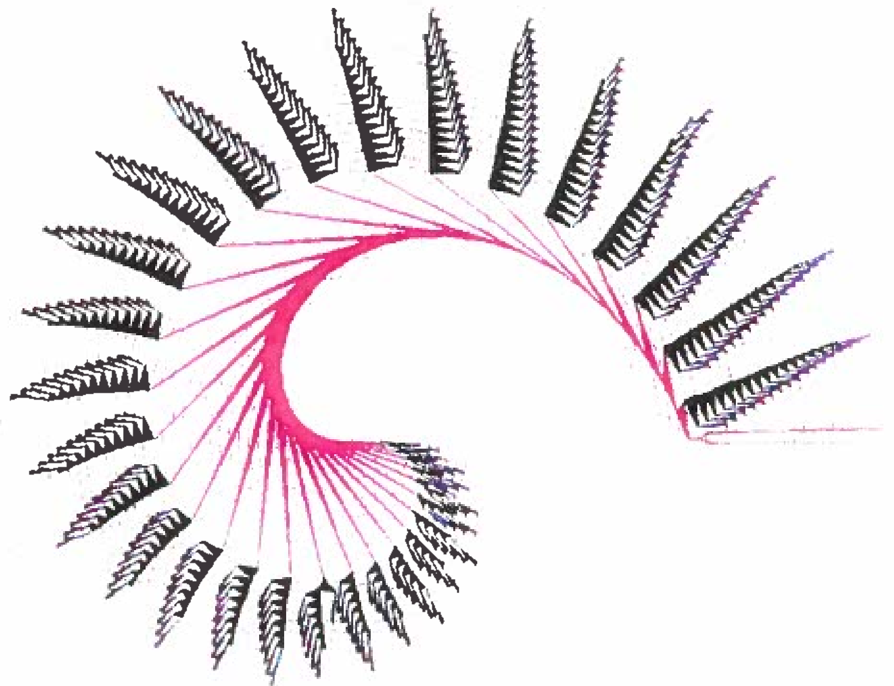
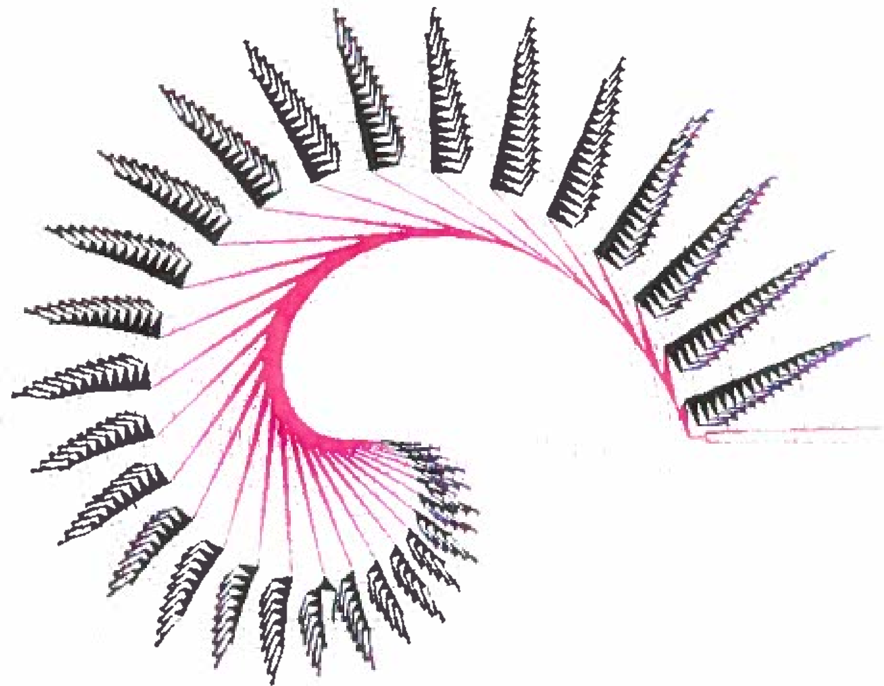


Figure 29: Execution Graph of Pascal's Triangle from Procedure View



LC: 58(1) | 5 PEs : BY Proc.

Figure 29: Execution Graph of Pascal's Triangle from Procedure View

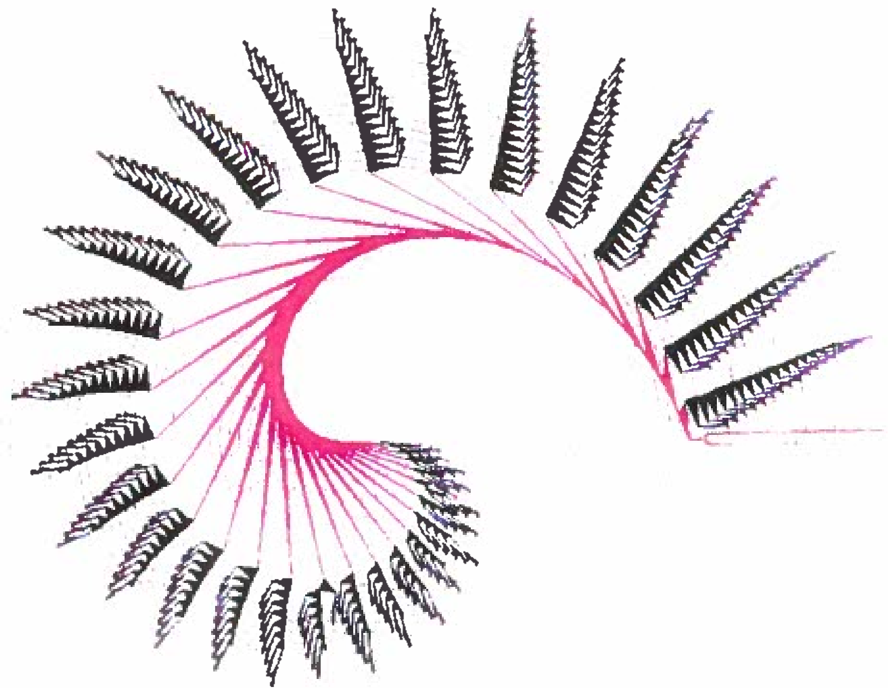


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

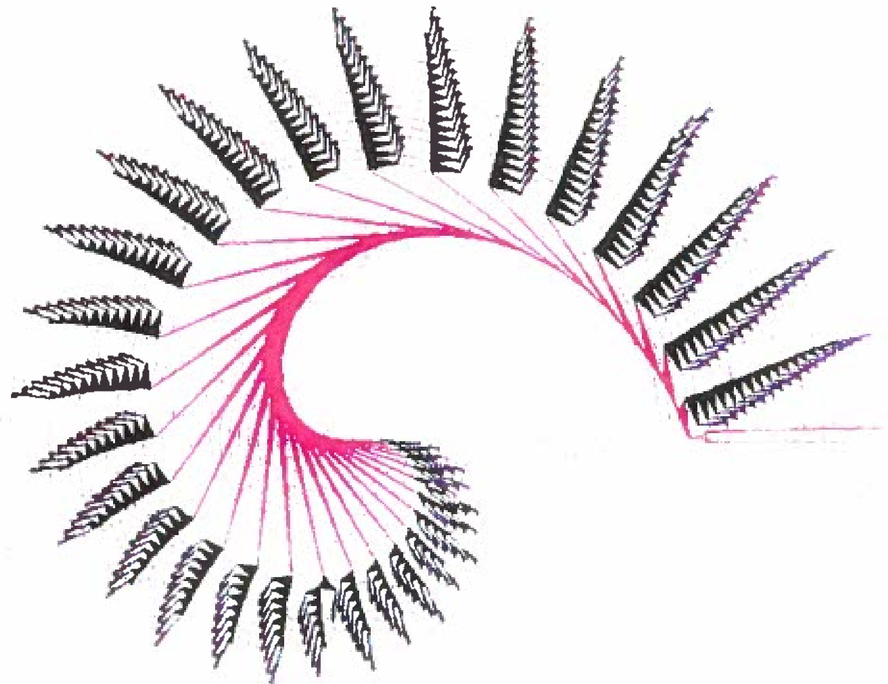


Figure 29: Execution Graph of Pascal's Triangle from Procedure View

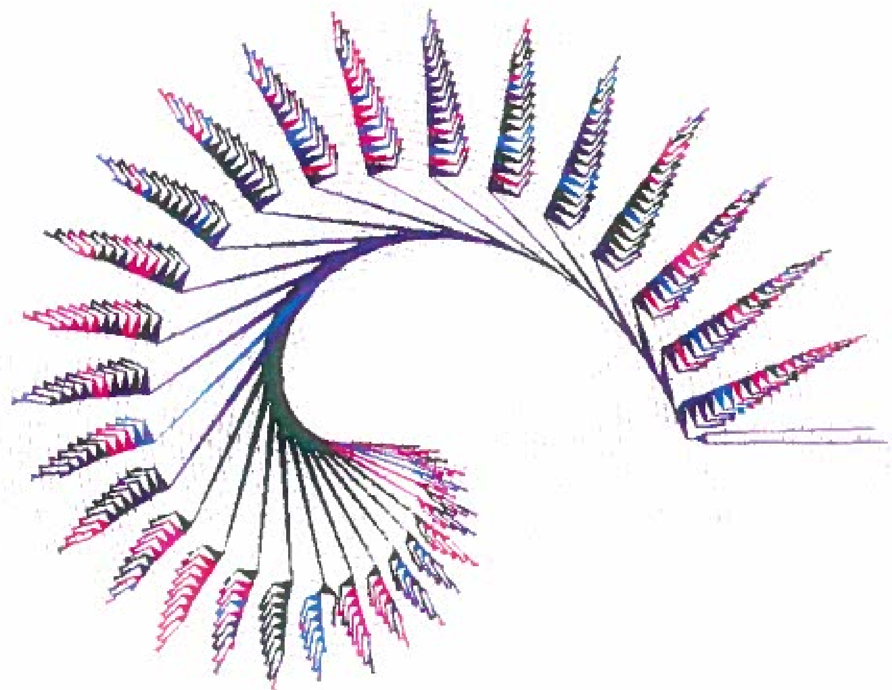


Figure 30: Execution Graph of Pascal's Triangle from PE View (5 PEs)

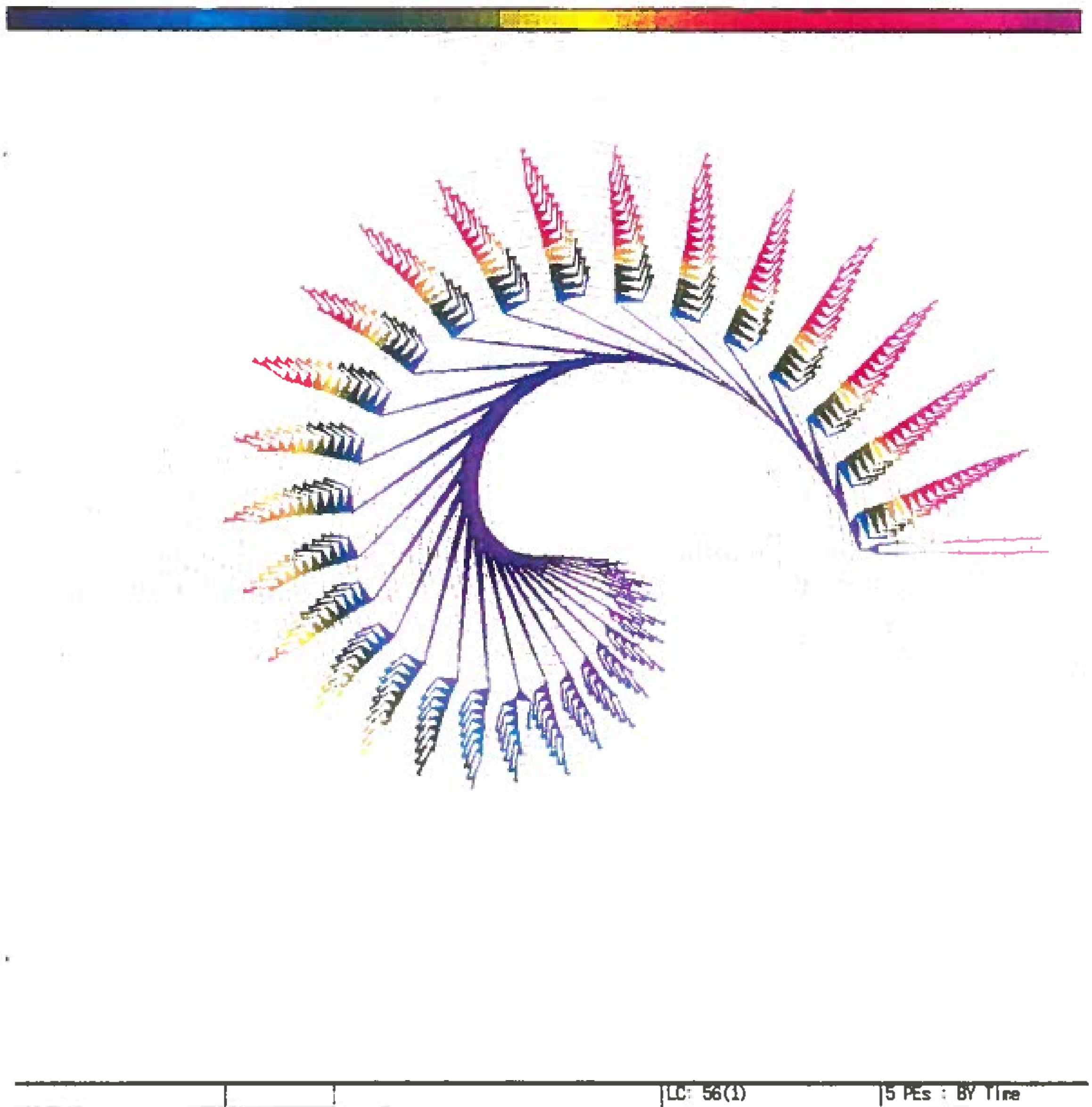


Figure 31: Execution Graph of Pascal's Triangle from Time View (5 PEs)