PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Visualizing Parallel Logic Programs with VISTA

E. Tick
Dept. of Computer Science
University of Oregon
Eugene OR 97403

ABSTRACT

A software visualization tool is described that transforms program execution trace data from a multiprocessor into a single color image: a *program signature*. The image is essentially the program's logical procedure-invocation tree, displayed radially from the root, with possible radial and lateral condensation. An implementation of the tool was made in X–Windows, and experimentation with the system was performed with trace data from Panda, a shared-memory multiprocessor implementation of FGHC. We demonstrate how the tool helps the programmer develop intuitions about the performance of long-running parallel logic programs.

1 Introduction

Parallel programming is difficult in two main senses. It is difficult to create correct programs and furthermore, it is difficult to exploit the maximum possible performance in programs. One approach to alleviating these difficulties is to support debugging, visualization, and environment control tools. However, unlike tools for sequential processors, parallel tools must manage a distinctly complex workspace. The numbers of processes, numbers of processors, topologies, data and control dependencies, communication, synchronization, and event orderings multiplicatively create a design space that is too large for current tools to manage.

The overall goal of our research is to contribute to processing this massive amount of information so that a programmer can understand it. There is no doubt that a variety of visualization tools will be needed (e.g., [6, 9, 12, 5]): no one view can satisfy all applications, paradigms, and users. Yet each view should be considered on its own merits: what are its strong and weak points, how effective is it in conveying the information desired, and hiding all else. In this paper we introduce one view in such a system: based on a new technique, called "kaleidescope visualization," that summarizes the execution of a program in a single image or signature.

Unlike scientific visualization, i.e., the graphical rendering of multi-dimensional physical processes, in parallel performance analysis there are no "physical" phenomena; rather, abstract interactions between objects. Thus renderings tend to be more abstract, are less constrained by "reality," and are certainly dealing with many interacting parameters controlling the design space. Kaleidescope visualization is the graphical rendering of a dynamic call tree of a parallel program in polar coordinates, to gain maximum utilization of space. To fit the entire tree into a single workstation window, condensation transformations are performed to shrink the image without losing visual information.

This paper concentrates on the analysis of parallel logic programs with VISTA, an X–Windows realization of kaleidescope visualization. Although we concentrate on committed-choice reduction-based languages, VISTA is applicable to a wider class of procedure-based AND-parallel languages. The paper is organization as follows. Section 2 summarizes similar types of visualization tools, and Section 3 reviews the VISTA algorithms (summarizing [14]). In Section 4, we describe the parallel logic programming platform upon which VISTA experimentation was conducted, and analyze the performance of logic programs to illustrate the power of the tool. In Section 5, conclusions and are summarized.

2 Literature Review

Earlier work on WAMTRACE [2, 3], a visualization tool for OR-parallel Prolog, has influenced our work a great deal. WAMTRACE is a trace-driven animator for Aurora Prolog [8] (originally for ANLWAM [2]). Aurora creates a proof tree over which processors ("workers") travel in search of work. WAMTRACE shows the tree, growing vertically from root (top) to leaves (bottom), with icons representing node and worker types (e.g., live and dead branchpoints, active and idle workers). The philosophy of WAMTRACE was that an experimental tool should present as much information to the programmer as is available. This often results in information overload, especially because the animation progresses in time, leaving only the short interval of the near-present animation frames in the mind of the viewer.

With comparison to WAMTRACE, our goals in VISTA were: (1) generalize the tool for other language paradigms. Specifically, AND-parallel execution is more prevalent in most languages, and needed to be addressed; (2) to summarize the animation; (3) abstract away information so as not to detract the viewer from understanding one thing at a time. Thus we introduce different views of the same static image, to convey different characteristics; (4) more advanced use of color to reduce image complexity and increase viewer intuitions.

Note that the emphasis of WAMTRACE on animation is a feature, not a bug — the animation enables the gross behavior of the dynamic scheduling algorithms to be understood. Animation is implemented, but not stressed in VISTA. In this paper, we analyze a system with simple on-demand scheduling [10], and so animation is not critical to understanding program behavior.

There are numerous views of performance data, quite different than WAMTRACE, e.g., [6, 9, 12, 5]. In general, these methods are effective only for large-grain processes, and either do not show logical (process) views of program execution, or cannot show such views for large numbers of processes. Voyeur [12] and Moviola [5] are closest in concept to VISTA. These animators have great benefit, but this limits the complexity that can be realistically viewed. Related research concerns a visual representation [7] and visual debugger [4] for committed-choice languages, but these are not for performance analysis.

3 Inside VISTA

The main goal of VISTA is to give *effective* visual feedback to a programmer tuning a program for parallel performance. To achieve this goal, VISTA displays an entire reduction tree in one (workstation) window, with image *condensation* if needed. Two types of condensation are performed: level and node condensing (described below). In addition, VISTA enables a user to view the tree from different perspectives (PE, time, or procedure) and zoom-up different portions of the tree. Since the tree is usually dense for small-grain parallel programs (even after condensation), and the tree must be redisplayed when the user desires different views, the tree-management algorithm must be efficient and the window space must be utilized effectively.

We now define terminology for describing logical call trees. The level of a node is the path length from the root to the node. The root level is zero. The root is the initial procedure invocation, i.e., the query. 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. 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 (defined below). Node condensing is the removal of all the descendants of the node n from the tree, if the allocated sector (defined below) for n in the window space is less than one pixel.

With these definitions in hand, VISTA management is now reviewed. There are two inputs to the algorithm: a trace file and a source program. A trace file entry consists information corresponding to a time-stamped procedure reduction. Although not currently implemented, VISTA could easily be extended to accept arbitrary events logged in the trace, as does WAMTRACE.

There are alternative ways to map an arbitrarily large tree onto a limited window space. We employ an abstraction requiring two passes over the trace: finding the tree height and creating a level-condensed tree. The condensed tree keeps the shape of the original tree (although scaling is not precise). This original shape allows us to carry our intuitions over from the tool's view to tuning performance of actual programs. In order to calculate the level-condensing ratio, c, the maximum tree height to be displayed (i.e., limitation of the window space) is needed: $h_{max} = \lfloor w/2d \rfloor$, where w is the maximum window width, and d is the distance between two adjacent levels. If tree height $h \leq h_{max}$, level condensing is not needed. If $h > h_{max}$, level condensing is performed with the c ratio calculated as follows:

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

where l is the node level. This condensation scheme puts more emphasis (space) on the levels closer to the root because earlier reductions are generally more "important" than later reductions. The heuristic corresponds to the user's intuition that processes responsible for distribution of many subprocesses should appear larger.

An open question is the categorization of programs into those which abide by this heuristic, and those which do not. A program that would "frustrate" VISTA heuristics has the most significant computation near the leaves, where distribution of this work (near the root) is less important. A trivial example is a tree of parallel tasks, each a very heavy sequential thread of computation. VISTA will condense the graph to fit within the window, in the limit (of very long threads) producing a star shape. Although this may be considered "intuitive," it is abstracts away all information except the threads, which themselves are difficult to view against the background. Alternative views, such as condensing each thread into a polygon, perhaps colored as a function of the condensation, may be more informative because the freed-up window space would allow the work distribution at the root to be viewed also. How this and other types of con-

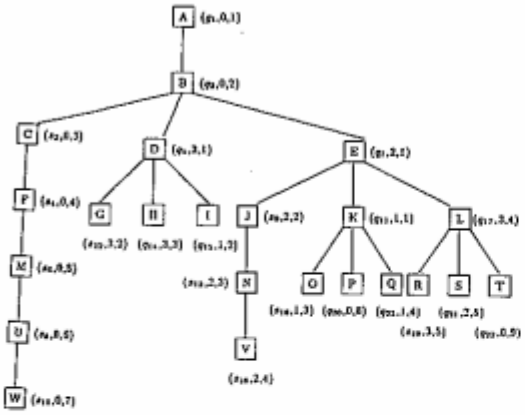

Figure 1: Whole Tree for Qsort: "?- qsort([2,1,4,5,3],X)"

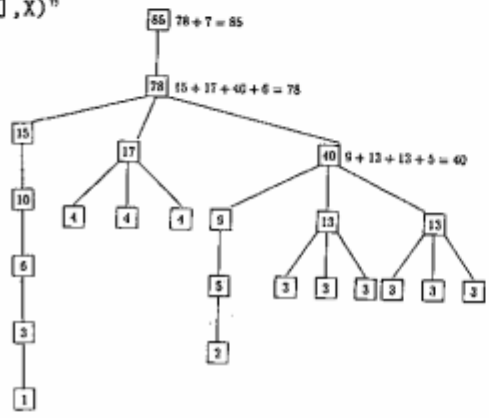

Figure 2: Weight Calculation for Qsort

densation, while not scaling the tree linearly, can lead to better understanding of certain programs, is a topic of future research.

At this stage of the algorithm, the levels to be displayed, and to be discarded, are decided. To illustrate, consider Fig. 1 showing the original tree for a Quick Sort program (the trace executed on four PEs and consists of 23 records). Each node is labeled with a triple, $(s_i, pe, index)$, where s_i is procedure s (abbreviated) invoked at trace index i, pe is the PE number, and $index$ is the sequence index of that PE. For example, $(q_5, 3, 1)$ of node D denotes that procedure *qsort* was invoked as the 5^{th} trace record, and reduced on PE = 3 as the first goal executed by that processor. After level condensing with c=2, nodes B, F–L, U, and V are contracted into their parents. These odd-level nodes are removed because if $(l \bmod c) \neq 0$, all nodes in level l are condensed.

Each node at level l in the logical tree is displayed in the window at a locus defined by radius $r = d \times l$, where d is the (constant) distance between two adjacent levels. The node is illustrated by a point, however it is connected to its children (at the next level) by a closed polygon around the "family" (the polygon degenerates into a line if there is one child). The polygon itself is colored, representing an attribute of the parent. After

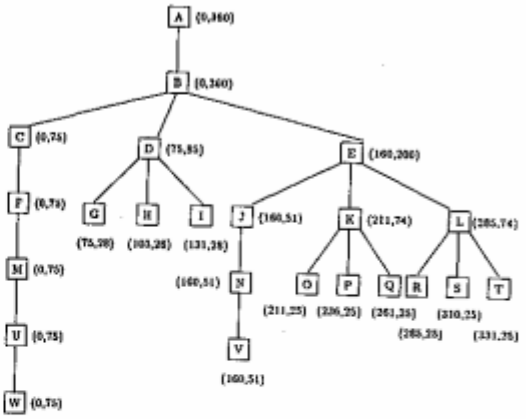

Figure 3: Node Allocation for Qsort

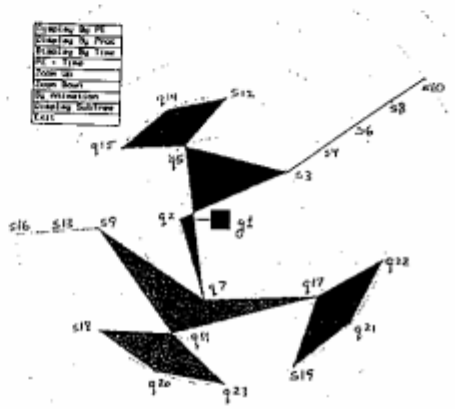

Figure 4: Execution Graph of Qsort: PE View (4 PEs)

level condensing has completed, the nodes at each level are allocated to the corresponding locus (a concentric circle). This is analogous to the pretty printing problem for text. We solve this problem heuristically by allocating a *sector* to each node depending on its *weight*. The node and its children are then displayed within the range of the sector only. The weight w for each node is heuristically defined as the sum of the weights of its children plus the height of the node. Thus 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. Fig. 2 shows an example of the weight calculation for the Quick Sort program.

A sector is defined as the subset of the concentric circle within which a node can be displayed. To formalize the sector calculation, consider a unique labeling of each node by a path from the root $\{x_1, x_2, ..., x_k\}$, where x_i is the sibling number traversed in the path. For example, in Fig. 2, node J with weight 9 has label $\{1, 3, 1\}$. The sector of a node at path p is represented as a pair (s_p, a_p), where s_p and a_p 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_p for a node at level k is calculated as follows:

$$s_{x_1,x_2,\ldots,x_k} = \begin{cases} s_{x_1,x_2,\ldots,x_{(k-1)}} & \text{leftmost child} \\ s_{x_1,x_2,\ldots,(x_k-1)} + a_{x_1,x_2,\ldots,(x_k-1)} & \text{otherwise} \end{cases}$$

In other words, if the node is the leftmost child, then the starting degree is equal to the starting degree of the node's parent. Otherwise, the starting degree of the node is equal to the sum of the starting and allocation degrees of its left sibling. The allocation degree a_p for a node at level k is calculated as follows:

$$a_{x_1,x_2,\ldots,x_k} = \frac{w_{x_1,x_2,\ldots,x_k}}{\sum_{j=1}^{m} w_{x_1,x_2,\ldots,(x_k-j)}} \times a_{x_1,x_2,\ldots,x_{(k-1)}}$$

where w_p is the node's weight, the summation is the total weight of all m siblings (including the node itself), and the final factor is the allocation degree of the parent. Fig. 3 shows the sectors of the nodes for Fig. 2.

After the previous steps, the execution graph is ready for display in the X-Window System [11] with VISTA as a client. When drawing the graphic, if the sector calculated is less than one pixel, node condensing is done, i.e., the node and its children are not displayed. 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 node position (x, y) in the window is calculated in the next step as $x = d \times l \times cos(s + a/2)$ and $y = d \times l \times sin(s + a/2)$, where, d is the level distance, l is the level of the node, and (s,a) are the start/allocation degrees of the node. A complete description of the internal algorithms is given in [14]. To put the algorithms into perspective, Fig. 4 shows the VISTA display showing a PE view of the Quick Sort program (corresponding to Figs. 1–3).

4 Program Analysis with VISTA

Our initial experimental testbed for VISTA is an instrumented version of the parallel FGHC system, Panda [10, 13]. Tuning a fine-grain parallel FGHC program for increased performance involves understanding how much parallelism is available and what portion is being utilized. In experimenting with parallel logic programs using VISTA, we have found a number of approaches useful for understanding performance characteristics. Our experiments consisted of a set of execution runs on a Sequent Symmetry, and involved both modifying the benchmarks and varying the numbers of PEs. We examine three such benchmarks here.

4.1 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. Our first benchmark [13] computes the 35^{th} row of coefficients, with bignum arithmetic.

The easiest way to understand a program in VISTA is with a procedure view or graph. Fig. 5 shows the reduction tree from the procedure view. This graph, displayed here without any condensation, has 2,235 nodes and a height of 56. The interesting snail shape, where the radial arms correspond to row calculations, indicates that the rows, and therefore the computations, are growing in size. Near the root, a cyan distribution procedure spawns the rows, and near the leaves, a sky-blue bignum procedure adds coefficients. 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 only the first half of the row is ever computed, 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. This program illustrates how the user can roughly understand execution characteristics from the procedure view, even without knowing the precise details of the source code.

To analyze the parallelism of the execution graph, we first examine load balancing among PEs. Good load balancing among PEs does not necessarily mean efficient exploitation of parallelism. However, without fair load balancing, full exploitation of parallelism cannot be achieved. In VISTA, a fair color distribution in the PE view or graph represents good load balancing. Figs. 6 (PE graph) and 7 (time graph) represent the execution on five PEs. In the time view, the RGB color spectrum from blue to magenta represents the complete execution time. Because there are few visibly distinct colors in this range, 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), then the reductions were executed sequentially.

All five colors are distributed almost evenly in the PE graph for Pascal, representing good load balancing. To further analyze parallelism, both PE and time graphs are used in conjunction. In the time view, the spectrum is distributed *radially*, although not perfectly so. This indicates that most rows were executed in parallel. Although the maximum parallelism is limited by the PEs at five, again the vagueness of the RGB spectrum can be misleading, making it appear as if there is *more* parallelism. This problem can be overcome to some extent with a subtree display, where the spectrum is recycled to represent time relative to the selected root.

Fig. 8 shows the single-PE time graph for Pascal. In this graph, the spectrum is distributed laterally, around the spiral. This distribution indicates that the nodes were executed by depth-first search, the standard Panda scheduling when no suspensions occur. By comparing the two time graphs, we can infer the manner of scheduling: breadth-first on five PEs, and depth-first on one PE, but without a PE view, we cannot conclusively infer parallelism. Fig. 6 shows that some rows are not executed entirely by the same PE (i.e., task switches

occur within some rows). These characteristics indicate that suspensions are occurring due to data dependencies between successive rows of coefficients. All three figures in conjunction indicate the "wavelike" parallelism being exploited as the leftmost coefficients of the Triangle propagate the computation down and to the right.

4.2 Semigroup Problem

The Semigroup Problem is the closure under multiplication of a group of vectors [13]. The benchmark uses an unbalanced binary hash tree to store the vectors previously calculated so that lookups are efficient when computing the closure. Fig. 9 (PE graph) and Fig. 10 (time graph) were executed on five PEs. The total number of nodes in the reduction tree is 15,419, and the tree height h=174. In this experiment, the window size was 850×850 and the level distance was four. The maximum tree height to be displayed is calculated as $h_{max} = \lfloor \frac{850}{4} \times 2 \rfloor = 106$. Level condensing is performed because $h > h_{max}$. The first 38 levels are not condensed, but the remaining 136 levels are condensed by 2:1. This example demonstrates some strong points of VISTA: (1) After level condensing, the tree keeps its original shape; (2) The window-space efficiency is very good. If the tree were represented in a conventional way (propagating from the top of the window), representation would be difficult, and space efficiency would be poor.

To understand the parallelism characteristics of Semigroup, load balancing among processors is analyzed first. The most immediate characteristic of the PE graph is that the reductions form the shape of many spokes or *threads* of procedure invocations. Near the root are distribution nodes. Each thread represents a vector multiplication. As the graph shows, almost all threads were executed without task switch. This indicates few suspensions due to lack of data dependencies, i.e., the vectors are *not* produced in the pipelined fashion of the Pascal program. By eye, we judge that the five colors in the PE graph are evenly distributed, indicating that load balancing is good.

Lack of data dependencies between nodes is confirmed by a single-PE time graph (not shown). The color distribution of this graph is similar to that of Fig. 10, indicating that as soon as the first node of the new thread is spawned, both the child and parent threads were executed in parallel, without any suspensions. The reason that the threads are not executed clockwise or anticlockwise in Semigroup, as in Pascal, is that there were some initial data dependencies near the root. These dependencies, caused by hash-tree lookups for avoiding recomputation of a semigroup member, cause critical suspensions that "randomize" the growth pattern.

When the PE graph (Fig. 9) is viewed in conjunction with the time graph (Fig. 10), parallelism can be analyzed in more detail. Threads with the same colors in the time graph, and different colors in the PE graph, are executed in parallel. The PE graph still has a fair number of threads per PE, indicating that not all potential parallelism has been exploited and additional PEs will improve speedup. These approximations can be refined by examining subtree displays.

Historically, the Semigroup program analyzed above was the result of a number of refinements from an original algorithm written by N. Ichiyoshi [13]. Earlier algorithms utilize a pipeline process structure, wherein new tuples are passed through the pipe, and duplicates are filtered away. Any tuple surviving the pipe is added as a new filter at the end, and all of its products with the "kernel" tuples (the program's inputs) are sent through the pipeline. Although these algorithms are elegant, the pipeline structure is a performance bottleneck. The version analyzed above utilizes a binary tree instead of a pipeline, increasing the parallelism of the checks.

In retrospect, we see how VISTA could have helped in developing these successive algorithms. Fig. 11 shows the time graph for an older version of the program that has the same complexity as the program analyzed above. Thus the main difference is the pipeline bottleneck, which is clearly indicated by the signature's snail shape. Unlike Pascal, time is not projecting radially, indicating lack of wave parallelism. Successive tuples are dependent on previous tuples surviving the pipeline, and this dependency is seen in the coloring (it could be better viewed if the RGB spectrum were more distinguished). The dependency is made explicit by clicking on nodes to indicate the corresponding procedures. Fig. 10 radiates from the query, indicating the potential parallelism afforded by a tree vs. a pipeline. The coloring further indicates that the tree is not bottlenecked.

4.3 Instant Insanity

The Instant Insanity problem is to stack four four-colored cubes so that the faces of each column of the stack display all four colors. This is a typical all-solutions search problem with eight solutions. There are several methods for doing the search in a committed-choice language: most notable are candidates/noncandidates and layered streams [13]. The candidates method builds an OR-tree where each node concerns whether the current candidate is consistent with the current partial solution. At the root, all orientations of all cubes are candidates and the partial solution is empty. At the leaves, no candidates remain and the partial solutions are complete. Each node has two branches: one branch contains the solutions that include the current candidate, and the other branch contains solutions that do *not* include the candidate. Layered streams is a network of filters that eagerly attempt to produce a stream of solutions of the form `H*T`. Here `H` is the first element shared by a set of solutions, and `T` are the tails of these solutions. To throttle excessive speculative parallelism, a "nil check" is inserted at each filter to ensure that `T` must have at least one element.

Layered streams has 9,094 reductions (nil check) and 9,775 reductions (without nil check). This increase of 7% reductions is because of two factors: the additional speculative execution and the bloated conversion of the now largely incomplete layered stream back into normal form. Both of these effects are seen by comparing the VISTA graphs (Fig. 12 and 13). The conversion routine is clearly viewed as a significant subtree without the nil check, compared to a single thread with checking. The user can now appreciate the relative weight of the conversion with respect to the entire search. The speculative branches, however, do not stand out. This would be an interesting application of user-defined trace records, where a "trace dye" could be introduced with a nil check that does not throttle the speculation.

The candidates program has 37,687 reductions, so that VISTA must condense the image. The final image, shown in Fig. 14, has 25,127 nodes. Examining the structure and coloring of the layered-streams and candidates programs, there are no obvious parallelism bottlenecks in either (measurements of all three programs showed equal PE utilization of 93–95%). From the time graph coloring, the fine-grain parallelism of the filter structure is apparent in the layered streams program. The candidates graph shows large-grain structure, although we must view the time and PE graphs together to ensure that PEs are equally distributed across time.

The simple examples analyzed here facilitate the exposition of VISTA. Intuitions gained for these programs have been confirmed by timing measurements [13]. Programs without as much parallelism, and on larger numbers of PEs, can be similarly analyzed. As the number of PEs grows, however, the tool approaches its limitations because the user can no longer distinguish between the multiple colors representing the PEs. This is an important area of future research.

5 Conclusions and Future Work

This paper described the performance analysis of parallel logic programs using "kaleidescope visualization." The VISTA system is an X–Windows realization of the method, and is demonstrated in the context of parallel FGHC programs. We showed how the user can tune a large-trace program for performance by examining alternative abstract views of the execution. VISTA, because of its efficient implementation, proved its merit in enabling rapid analysis of views. This tool complements, but by no means replaces, other visualization methods, e.g., animation of PE activity and message passing.

We are currently extending this research in several areas. First, we need to experiment more with the current VISTA prototype, for various programming languages, to determine its utility. Second, coloration methods for combining the time and processor views need exploration, e.g., a method of spectral superposition [1].

The author was supported by an NSF Presidential Young Investigator award, with funding from Sequent Computer Systems Inc. Computer resources were supplied both by OACIS and Argonne MCS. D.-Y. Park, in an outstanding effort, implemented VISTA.

[1] J. A. Berton. Strategies for Scientific Visualization: Analysis and Comparison of Current Techniques. *Proceedings of Extracting Meaning from Complex Data: Processing, Display, Interaction*, SPIE vol. 1259, pages 110–121. February 1990.

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

[3] T. Disz *et al.* Experiments with OR-Parallel Logic Programs. In *Inter. Conf. on Logic Prog.*, pages 576–600. MIT Press, May 1987.

[4] Y. Feldman and E. Shapiro. Temporal Debugging and its Visual Animation. In *Inter. Symp. on Logic Prog.*, pages 3–17. MIT Press, November 1991.

[5] R. Fowler *et al.* An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989.

[6] M. T. Heath and J. A. Etheridge. Visualizing the Performance of Parallel Programs. *IEEE Software*, pages 29–39, September 1991.

[7] K. M. Kahn and V. A. Saraswat. Complete Visualization of Concurrent Programs and their Executions. In *IEEE Visual Language Workshop*. IEEE Computer Society, October 1990.

[8] E. Lusk *et al.* The Aurora Or-Parallel Prolog System. In *Inter. Conf. on Fifth Gen. Comp. Systems*, pages 819–830, Tokyo, November 1988. ICOT.

[9] A. D. Malony and D. Reed. *Visualizing Parallel Computer System Performance*, pages 59–90. Addison-Wesley, 1990.

[10] M. Sato and A. Goto. Evaluation of the KL1 Parallel System on a Shared Memory Multiprocessor. In *IFIP Working Conference on Parallel Processing*, pages 305–318. Pisa, North Holland, May 1988.

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

[12] D. Socha *et al.* Voyeur: Graphical Views of Parallel Programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.

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

[14] E. Tick and D.-Y. Park. Kalaidescope Visualization of Fine-Grain Parallel Programs. In *Hawaii Inter. Conf. on System Sciences*, vol 2, pages 137–148. Kauai, IEEE Computer Society, January 1992.

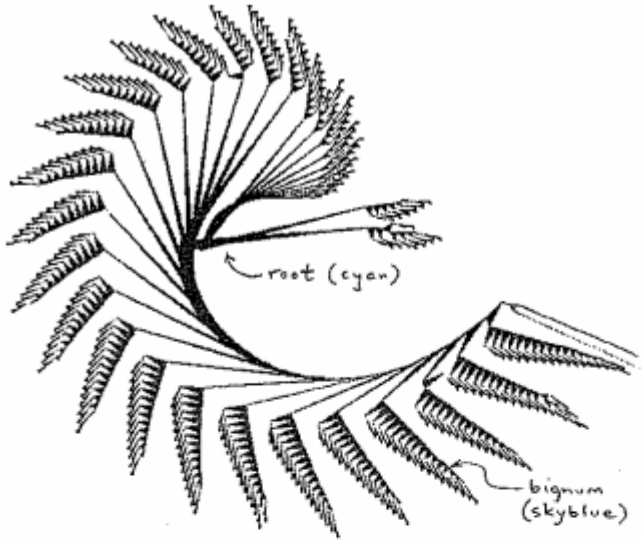

Figure 5: Graph of Pascal from Procedure View (5 PEs)

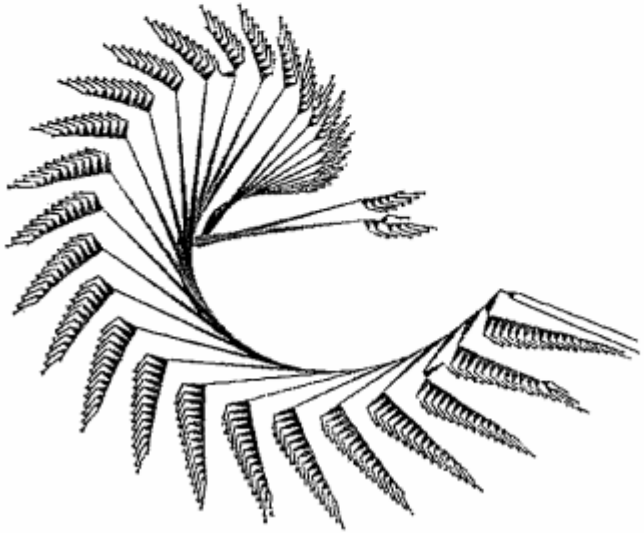

Figure 6: Graph of Pascal from PE View (5 PEs)

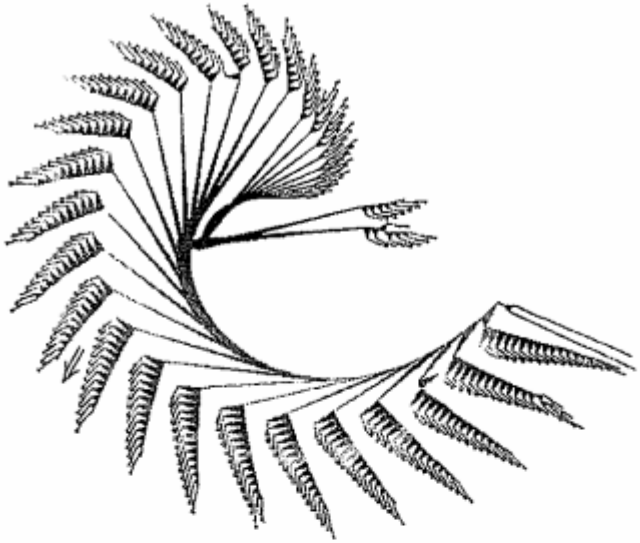

Figure 7: Graph of Pascal from Time View (5 PEs)

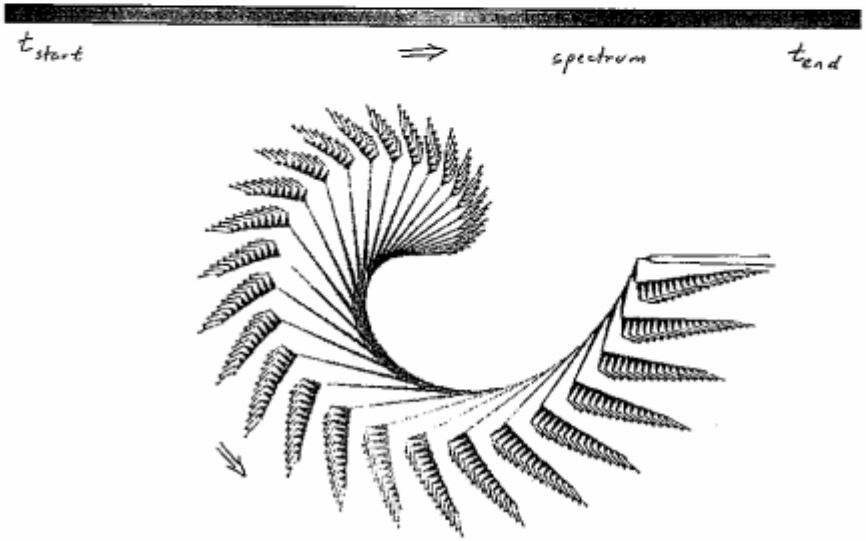

Figure 8: Graph of Pascal from Time View (1 PE)

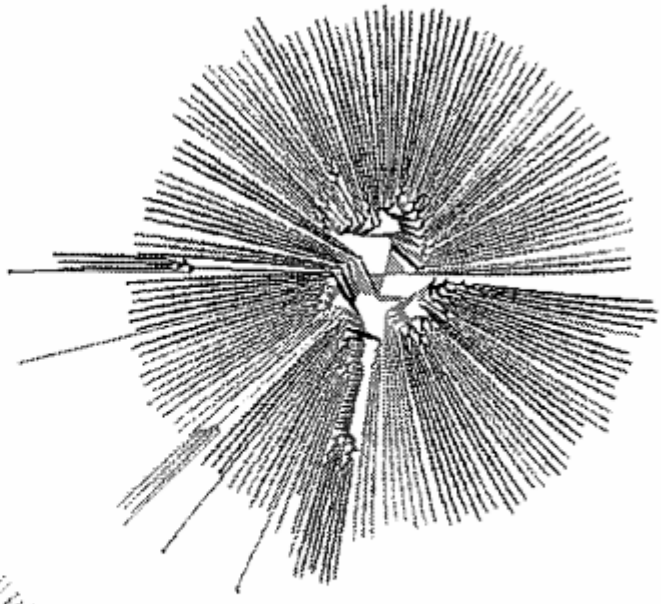

Figure 9: Graph of Semigroup from PE View (5 PEs)

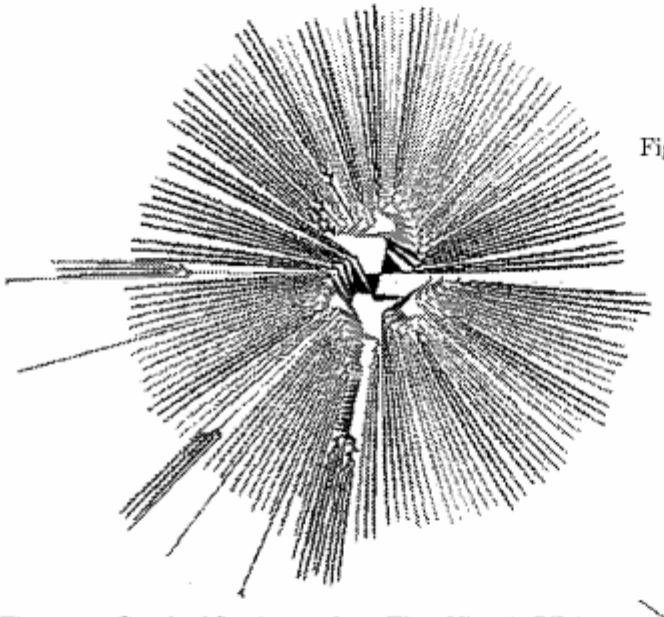

Figure 10: Graph of Semigroup from Time View (5 PEs)

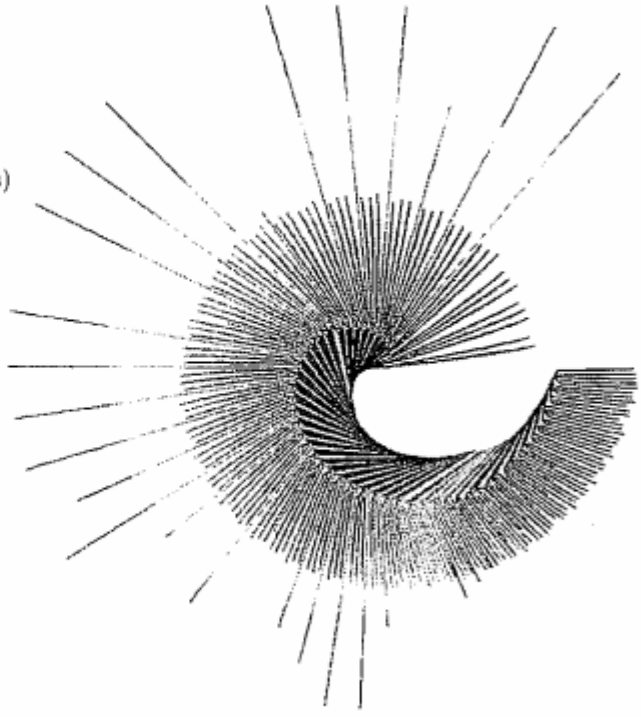

Figure 11: Graph of Old Semigroup Algorithm from Time View (5 PEs)

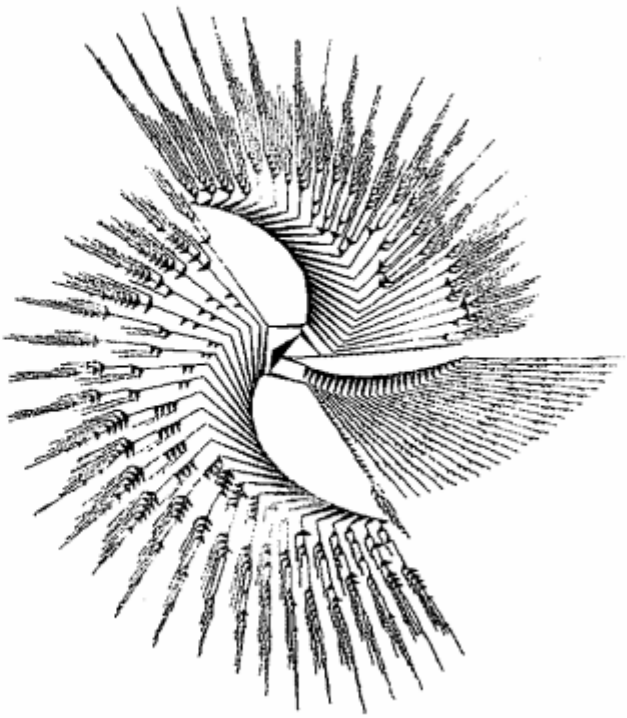

Figure 12: Graph of Layered Stream Cubes with Nil Check from Time View (5 PEs)

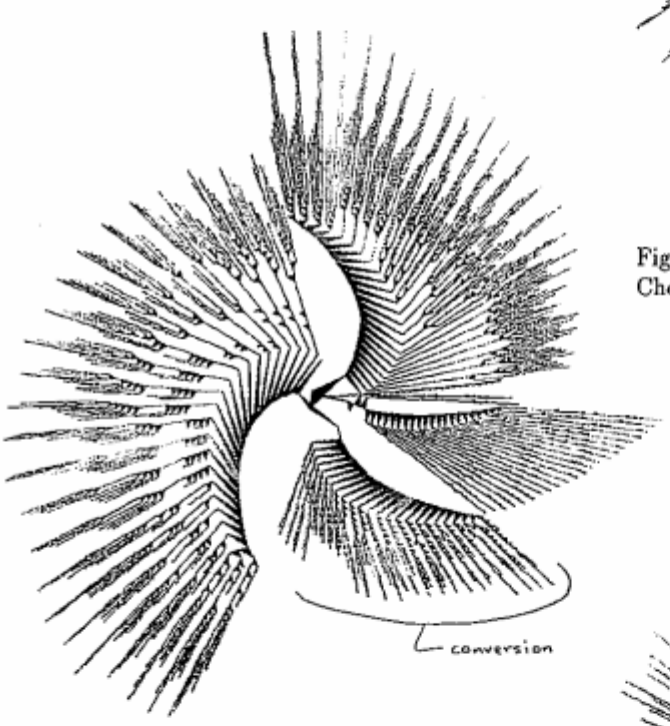

Figure 13: Graph of Layered Stream Cubes without Nil Check from Time View (5 PEs)

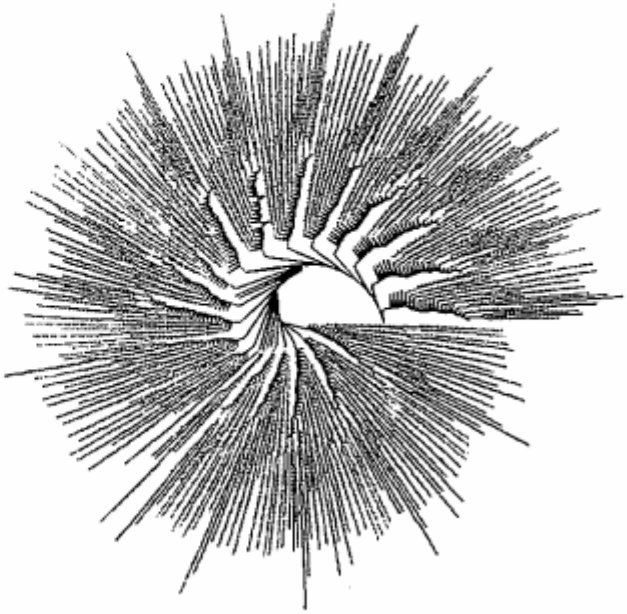

Figure 14: Graph of Candidates Cubes from Time View (5 PEs)