Kaleidescope Visualization of Fine-Grain Parallel Programs

E. Tick and D.-Y. Park

CIS-TR-91-18
October 1991

Abstract

A software visualization tool is described that transforms program execution trace data from a multiprocessor into a single color image. The image is essentially the program's logical procedure-invocation tree, displayed radially from the root. An algorithm is described that condenses the image both radially and laterally, producing a color-dense abstraction of the program's execution behavior: a *program signature*, within a workstation window. Three views are offered, implemented by painting the image with colors corresponding either to processors, procedures, or time. Time is sequentially mapped onto the color spectrum, exploiting the user's perception of spatial locality. The signature can be thought of as an animation where successive frames are superimposed; however, our algorithm has the advantage, like that of a text pretty printer, of analyzing the entire trace to produce a single, clear picture. An implementation of the tool was made in X-Windows, including a user interface that can zoom the image and annotate any node in the image with the corresponding procedure name. Experimentation with the system was performed with trace data from Panda, a shared-memory multiprocessor logic programming system. We demonstrate how the tool helps the programmer develop intuitions about parallel performance and how condensation successfully abstracts large traces.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

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., [7, 10, 16, 6, 4]): 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. Visualization is still very much art, and from that perspective, we consider ourselves "abstractionists," i.e., we are searching for the ideal abstract pictures to clearly convey parallelism, speedup, utilization, and other metrics.

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. Section 3 of this paper describes an X-windows application called VISTA that is our prototype trace-driven implementation of kaleidescope visualization. Both condensation and coloration algorithms used in VISTA are explained. In Section 4, we describe the parallel logic programming platform upon which experiments with VISTA were conducted, and analyze the performance of a few logic programs to illustrate the power of the tool. In Section 5, conclusions are summarized and future research is outlined.

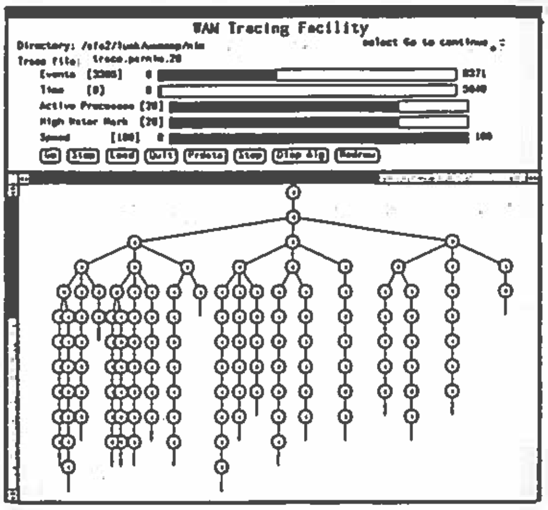

Figure 1: A WAMTRACE Snapshot

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 [9]. Aurora creates a proof tree over which processors ("workers") travel in search of work. As illustrated in the snapshot given in Figure 1, 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:

- generalize the tool for other language paradigms. Specifically, AND-parallel execution is more prevalent in most languages, and needed to be addressed.
- to summarize the animation, effectively superimposing frames.
- abstract away as much information as possible 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.
- 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. Currently, the signature in VISTA is not animated, although it could easily be displayed in a manner to suggest animation. In this paper, we analyze a system with simple on-demand scheduling [12], and so animation is not critical to understanding program behavior.

There are numerous views of performance data, quite different than WAMTRACE, e.g., [7, 10, 16, 6, 4]. 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 [16] and Moviola [6] are closest in concept to VISTA. These animators have great benefit, but this limits the complexity that can be realistically viewed. The relation of these other tools with VISTA is reviewed by Park [11]. Other related research by Kahn and Saraswat [8] concerns a visual representation of committed-choice languages, mainly for programming benefits, not for performance analysis.

3 VISTA: Visualization Tool for Parallel Programs

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, as described in this section.

In addition, VISTA enables a user to view the tree from different perspectives, such as 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.

These goals were accomplished with the algorithms described here. As a small example, consider that the trace-processing and redisplay times of the 15,419 trace records (524,248 bytes) for the Semigroup benchmark (see Section 4) are 45 seconds and 4 seconds, respectively, for a processing rate of approximately 300 records/second (SUN SPARC). These response times encourage a user to frequently redisplay data in different views.

It is easiest to describe the algorithms by illustrating their effect on a small parallel program. To this end, we first introduce a parallel logic programming language, FGHC, because it has an elegant and extremely simple concurrent semantics. FGHC is a

```
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 2: Sample Program : Quick Sort in FGHC

committed-choice parallel logic programming language, related to languages such as Parlog and FCP [15]. There is nothing particular to this family of languages that is required for the algorithms described here, or the VISTA tool. Currently we are entertaining plans to instrument other parallel languages such as PCN [5].

Committed-choice programs are built of guarded Horn clauses of the form:

$$H :- G_1, G_2, ..., G_m \mid B_1, B_2, ..., B_n$$

where $m \geq 0$ and $n \geq 0$. H is the clause head, G_i is a guard goal, and B_i is a body goal. The commit operator '|' divides the clause into a passive part (the *guard*) and active part (the *body*). Note that if the guard is empty, no commit operator is given. The informal semantics[1] of such programs are: to execute a procedure $p(A_1, A_2, ..., A_k)$ with arguments A_i, the arguments must match (passively unify with) the formal parameters of the head of some clause with name p. In addition, the guard goals must all succeed. If these conditions hold, the procedure invocation may "commit" to that clause (or *any* other clause so satisfying these conditions). The body of the committed clause is then executed. A program successfully executes when no goals remain to be executed.

Synchronization in committed-choice programs in enforced implicitly in the commit rule. If no clause can commit to a procedure invocation, that invocation *suspends*. Suspension occurs because a passed parameter is not sufficiently instantiated to allow successful head matching or guard reduction. A suspended invocation is *resumed* when a later binding is made to a variable associated with the suspended invocation.

To describe the algorithms, we consider the simple Quick Sort program shown in Figure 2. We now give some terminology for describing logical call trees.

Definition: The *level* of a node is the path length from the root to the node.

[1] For more formal semantics, see for instance Shaprio [14].

The level of the root is zero. The root is the initial procedure invocation, called the *query*. □

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. □

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 (defined in Section 3.1). □

Definition: *Node condensing* is the removal of all the descendants of the node n from the tree, if the allocated sector (defined in Section 3.3) for n in the window space is less than one pixel. □

3.1 Condensation Algorithms

There are several ways to map an arbitrarily large tree onto a limited window space. One method is to create an entire tree from the trace, and then reduce the tree whenever the tree is displayed. Although this method needs only one pass over the trace file, it may be impossible to fit the entire tree in main memory. If the tree resides in secondary storage, access time would be too slow. Another method is to reduce the tree, by the ratio of the maximum dimension of the window to the total number of trace records. However, the resulting tree, a simple *contraction* of the original tree, will not correspond to user intuitions, because the trace is sequenced by time, not ancestor relationships.

We employ a more accurate abstraction, although it requires more I/O. Two passes are required over the trace: one to find the tree height and a second pass to create a level-condensed tree. Since the tree is condensed by level, the tree after condensation 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.

VISTA management is summarized in Figure 3 and explained in the following 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 4 shows the format of the remaining trace records. Each field in a record is a non-negative integer value. After reading a trace record, the children count (the number of body goals in the procedure clause corresponding to the invocation) is obtained by indexing the source program with *clause#*. The *index* is an element of an ordered, independent

```
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 3: Overall Control Algorithm for VISTA

sequence for each PE, e.g., if *index* = k, then the trace record corresponds to the k^{th} procedure executed on *pe*.

In the first pass, the tree height[2] is computed from the trace, without creating a tree data structure, as summarized in Figure 5. To calculate the height of the tree, we use a closed hash table, accessed with the function hash(*pe*, *index*). The hash table is illustrated in Figure 6.[3] The algorithm places nodes in the table with function *hash-producer*, and looks-up the parent of a node with function *hash-consumer*. When the child count, *ccnt*, in an entry is decremented to zero, the table entry can be reused. Notice that the hash function naturally partitions the trace records corresponding to each PE, to different segments of the table. In the example, table entries 11, 21, and 23 still have children to process, potentially calculating a new tree height.

[2]The basic unit of window space is assumed to be one pixel throughout the paper.

[3]Assume for now that the *hindex* and *nodeptr* fields are invalid, as they are used only in the next pass.

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

Figure 4: Trace Record Format

```
first-pass(trace file, source program) {
        read first and second trace record;              — $\mathrm{PE}_{max}$ and first node record
        hash-producer(current record);
        tree-height := 0;
        while (read next record (rec) $\neq$ end of file) {
                level := hash-consumer(rec);
                if (level > tree-height)
                        tree-height := level;
                $ccnt$ := child-count(rec.$clause\#$);
                if ($ccnt > 0$)                           — process the node's children
                        hash-producer(rec);
        }
        return (tree-height);
}

hash-producer(rec : trace record) {
        $i$ := hash(rec.$pe$, rec.$index$);               — $i$ is hash index of node
        $ccnt$ := child-count(rec.$clause\#$);
        hash-table[$i$] := (rec.$pe$, rec.$index$, rec.$level$, $ccnt$);
        return ($i$);                                    — return node's level
}

hash-consumer(rec : trace record) {
        $i$ := hash(rec.$ppe$, rec.$pindex$);             — $i$ is hash index of parent
        hash-table[$i$].ccnt := hash-table[$i$].ccnt $-$ 1;  — consume one child
        return (hash-table[$i$].level + 1);              — return child's level
}
```

Figure 5: Algorithm for Calculating Tree Height

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 6: Hash Table after Processing 15^{th} Record (Other Entries Invalid)

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:

$$h_{max} = \left\lfloor \frac{w}{2d} \right\rfloor,$$

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 $h \le h_{max}$, then no level condensing is needed and the entire tree is displayed (unless node condensing is performed, as described in Section 3.3). If $h > h_{max}$, level condensing is performed with the level-condensing ratio, c, calculated as follows:

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

Parameter t is called the *threshold* and 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. At this stage of the algorithm, the levels to be displayed, and to be discarded, are decided.

It should first be noted that a remaining implementation 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. It is inefficient to allocate the maximum out-degree for the children pointers, and thus a binary tree is used to emulate the logical tree. This decision is primarily an artifact of the implementation language C.

The level-condensing algorithm is summarized in Figure 7. The closed hash table used in the first pass is used here also. Field *hindex*, the parent hash index, is shown only for clarity, but need not be stored in the table. Field *nodeptr* is a pointer to the node of the trace record if the trace record is not condensed or a pointer to the parent node if the trace record is condensed. Note that the parent itself may be condensed, and so on. This guarantees that a tree branch is built skipping over all condensed nodes.

Figure 8 shows the original tree for Quick Sort (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.

Figure 9 shows the resulting tree after level condensing, assuming level-condensing ratio $c = 2$. All odd-level nodes are removed because if $(l \bmod c) \neq 0$, all nodes in level l are condensed. Figure 6 shows the hash table after processing the 15^{th} record in the second pass, assuming that no collisions occur. Values A, C, M, E, N, and D in field *nodeptr* of the hash table represent pointers to the nodes of Figure 9.

The final hash table field, *flag*, is required if the second pass is called by a subtree request, in which the user requires the tree to be redisplayed from a given root. The trace index of this selected root is accessed from the tree structure and passed to the algorithm. It is obvious that all trace records preceding the selected root cannot be part of the subtree. However, if we attempt to process the trace file *starting* at the selected root, it is not possible to determine which nodes are subtree members because ancestor information has been lost. Therefore we reprocess the trace, from the beginning, using the *flag* field. The subtree root has a valid *flag*, and each node inherits the *flag* of its parent. Thus we can build a correct subtree. Level condensing is *not* performed for a subtree in order to view detail near the selected root.

3.2 Weight Calculation

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

```
second-pass(trace file, source program) {
        $c_0$ := 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 $\neq$ end of file) {
                level := hash-consumer(current record);
                if ((tree-height $\leq h_{max}$) or (subtree request))
                        level-condense-no(current record);
                else {
                        if (level $\leq$ threshold)
                                rem := level mod $c_0$;
                        else rem := level mod ($c_0 + 1$);
                        if (rem = 0)
                                level-condense-no(current record);
                        else level-condense-yes(current record);
}       }       }

level-condense-no(rec : trace record) {
        node := make-node(rec);
        $hindex$ := hash(rec.$ppe$, rec.$pindex$);
        connect to the parent or siblings using parent node ptr;
        $ccnt$ := child-count(rec.$clause\#$);
        if ($ccnt$ > 0) {
                $i$ := hash-producer(rec);
                hash-table[$i$].nodeptr := node;     — hash entry points to new node
}       }

level-condense-yes(rec : trace record) {
        $hindex$ := hash(rec.$ppe$, rec.$pindex$);
        $ccnt$ := child-count(rec.$clause\#$);
        if ($ccnt$ > 0) {                            — hash entry points to parent
                $i$ := hash-producer(rec);
                hash-table[$i$].nodeptr := hash-table[$hindex$].nodeptr;
}       }
```

Figure 7: Algorithm for Second Pass: Level Condensing

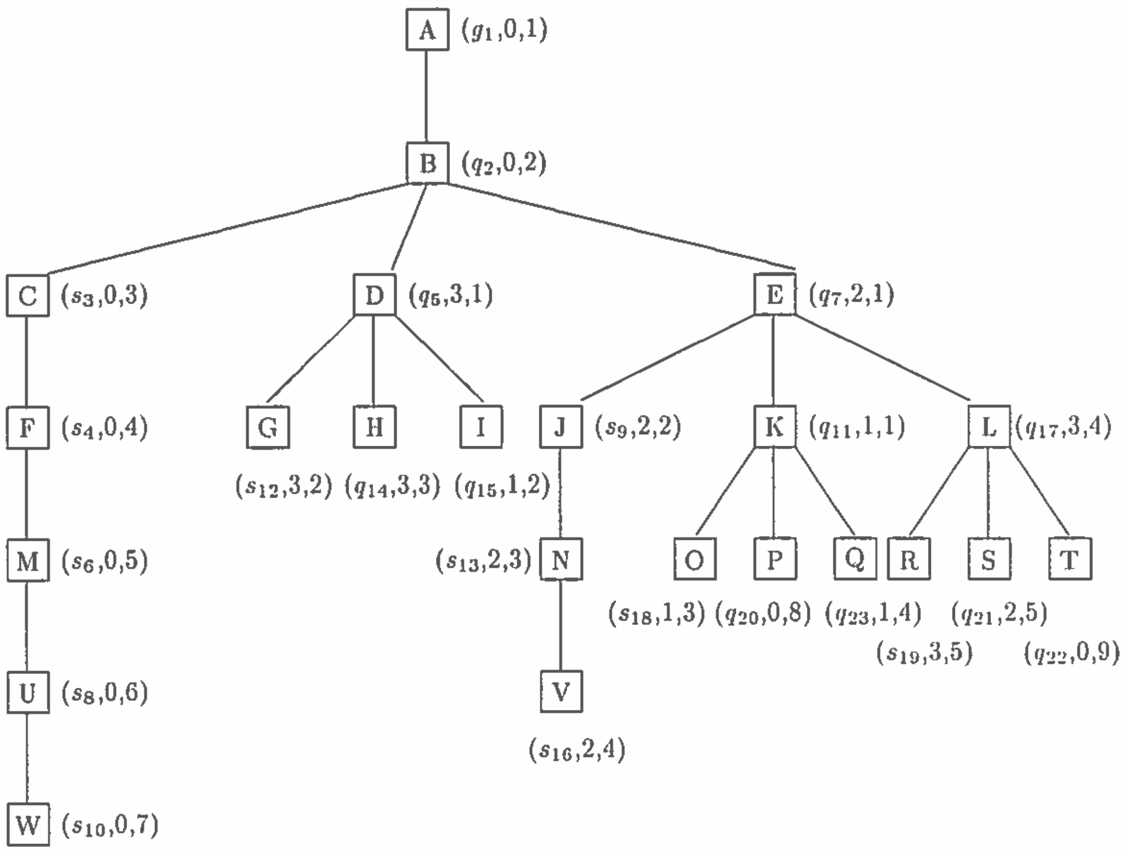

Figure 8: Whole Reduction Tree for Quick Sort Program: "?- `qsort([2,1,4,5,3],X)`"

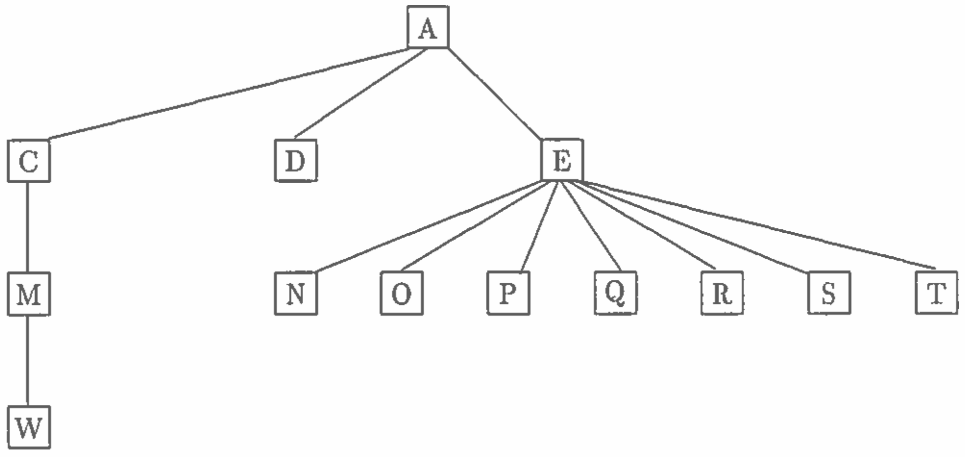

Figure 9: Reduction Tree After Level Condensing Quick Sort Program by 2:1

level) by a closed polygon around the "family" (the polygon degenerates into a line if there is one child). The polygon itself is filled, as is discussed in Section 4. After level condensing has completed, the main problem is *how* the nodes at each level are allocated to the corresponding locus (a concentric circle). This is analogous to the pretty printing problem for text.

A naive, evenly-spaced allocation has the serious disadvantage that nodes at further levels are not taken into account, resulting in overlapping polygons and a general mess. 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.

Before calculating the weight, a *level-connection transformation* is made to enable effective traversal of the tree. Standard depth-first and breadth-first search techniques, even if implemented without recursion, have high space complexity for large graphs. We transform the tree into a more efficient data structure, as illustrated in Figure 10. This figure represents the Quick Sort reduction tree (Figure 8) after level connection, assuming no level condensing has been done. A binary tree has been implemented, as discussed in the second pass. In order to traverse the tree by level, a *cousin pointer* (represented as "···" in Figure 10) is introduced. This level-connection transformation cannot 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. The tree (actually a DAG) can now be quickly processed from any level. When a subtree is requested or a node inquiry made, the tree is searched from the level nearest to the mouse cursor position.

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. Figure 11 shows an example of the weight calculation for the Quick Sort program. For example, the weight of the node in level two is 78, which is calculated from 72 (the sum of its child weights) plus 6 (the height of the node).

The weight-calculation algorithm is summarized in Figure 12. Procedure *connect-level*, connects levels, starting from the root. In each level, the first node in the next level is found and stored in array 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, starting at the leaves because each node needs its child weights to compute its weight.

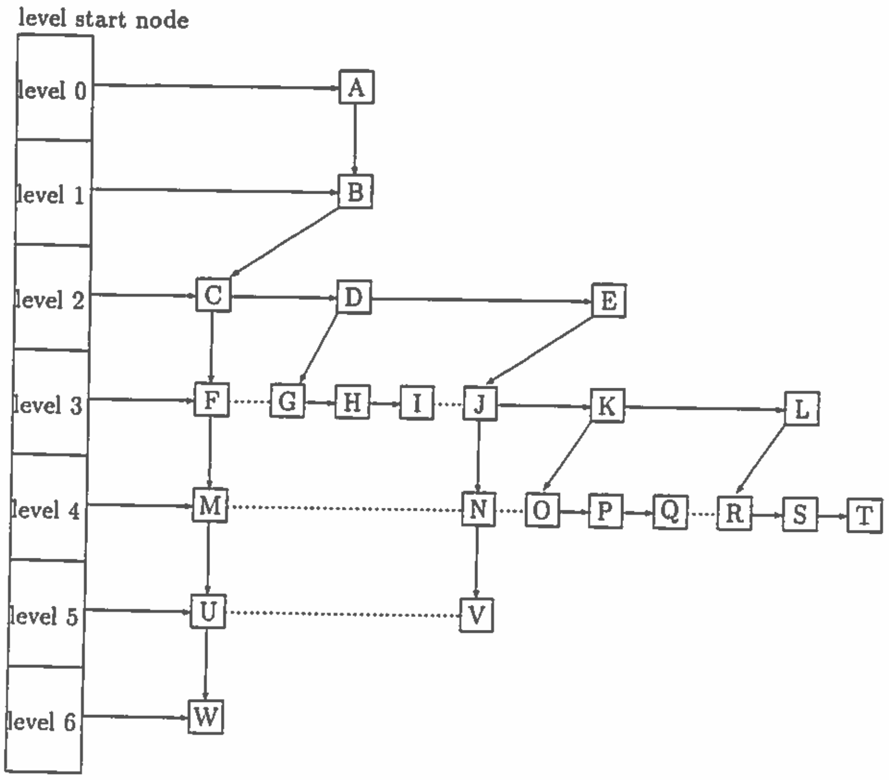

Figure 10: Reduction Tree After Level Connection for Quick Sort Program

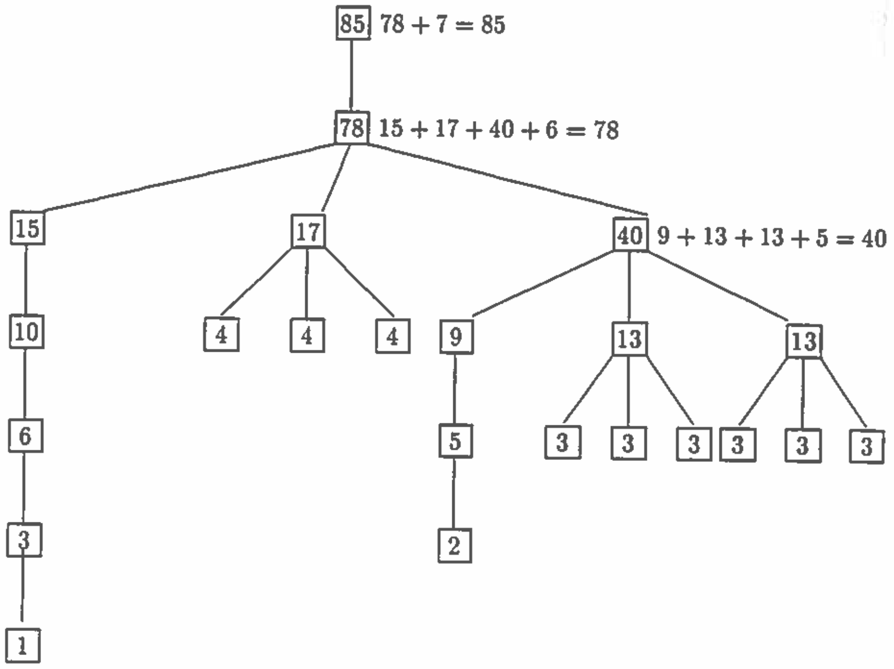

Figure 11: Weight Calculation for the Quick Sort Program

```
weight-calculation(tree) {
        connect-level(tree);
        for ($i$ = tree-height; $i \geq 0$; $i := i - 1$) {
                node := level-start-node[$i$];
                while (node $\neq$ NULL) {
                        node-height := tree-height $- i + 1$;
                        node→weight := node-height + sum of child weights;
                        node := node→next;
}       }       }

connect-level(tree) {
        level-start-node[0] = root-node;
        for ($i$ = 0; $i \leq$ tree-height; $i := i + 1$) {
                node := level-start-node[$i$];
                while (node $\neq$ NULL) {
                        if (node has children) {
                                if (child is the leftmost 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 12: Algorithm for Weight Calculation

3.3 Node Allocation and Display

After weight calculations, a sector is allocated to each node depending on its weight. The 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 Figure 11, 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:

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

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. Figure 13 shows the sectors of the nodes for Figure 11. For example, the sector of node D is calculated as: $s = 0 + 75 = 75$, and $a = \frac{17}{72} \times 360 = 85$. The sector of node J is calculated as: $s = 160 + 0 = 160$ and $a = \frac{9}{35} \times 200 = 51$.

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 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. The algorithm for the node allocation is summarized in Figure 14. The sector for each node is calculated, starting at the root because each node needs the allocation and starting degrees of its parent.

After the previous steps, the execution graph is ready for display. The graphic is drawn in the X–Window System [13] with VISTA as a client. 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, mouse 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 a node inquiry, VISTA first finds the node which is the nearest the cursor position. The information about the node is then displayed. If the action is 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. This node becomes the root of the subtree, and the second pass through node allocation are reperformed. Lastly, the PE view 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 detailed display algorithms are given in Park [11].

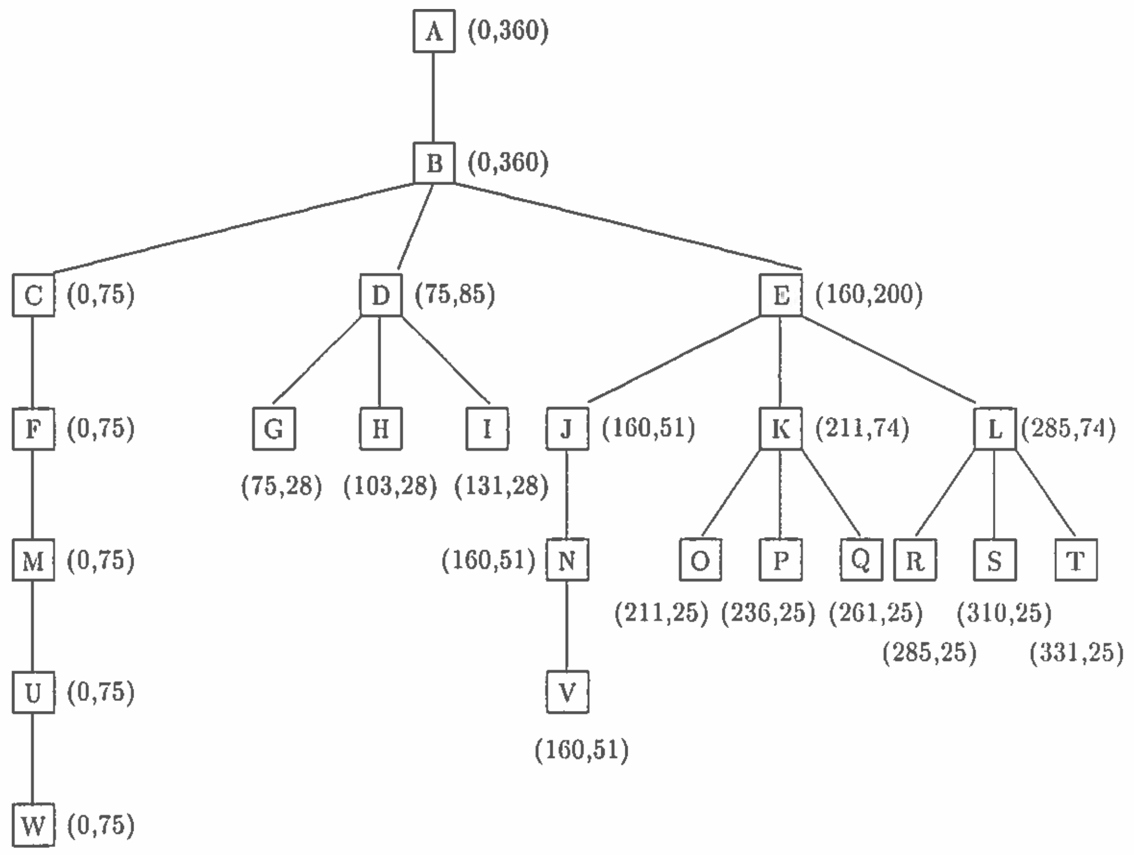

Figure 13: Node Allocation for the Quick Sort Program

```
node-allocation(DAG) {
    root-node := level-start-node[0];
    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) {
            ad := node→alloc-degree;
            sd := node→start-degree;
            tw := node→weight − height of the node;     — tw = sum of child weights
            child := node→left-child;
            while (child ≠ NULL) {
                child→alloc-degree := (child→weight / tw) * ad;
                child→start-degree := sd;
                sd := sd + child→alloc-degree;
                child := node→right-sibling;
}   }   }   }
```

Figure 14: Algorithm for Node Allocation

4 Program Analysis: Panda Platform and Coloration

Our initial experimental testbed for VISTA is an instrumented version of the parallel FGHC system, Panda [12, 17]. 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. Due to space limitations, we give only two simple examples 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 computes the 35^{th} row of coefficients, using bignum arithmetic (see Tick [17] for details).

The easiest way to understand a program in VISTA is with a procedure view or graph. Figure 15 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. Figures 16 (PE graph) and 17 (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

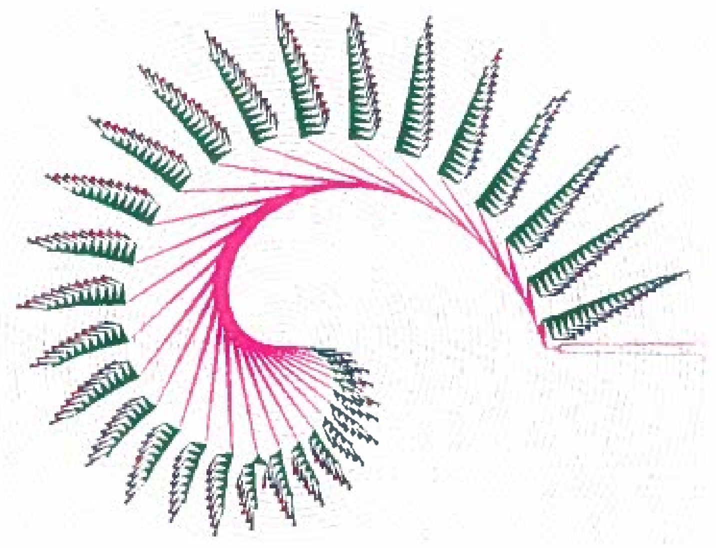

Figure 15: Execution Graph of Pascal's Triangle from Procedure View

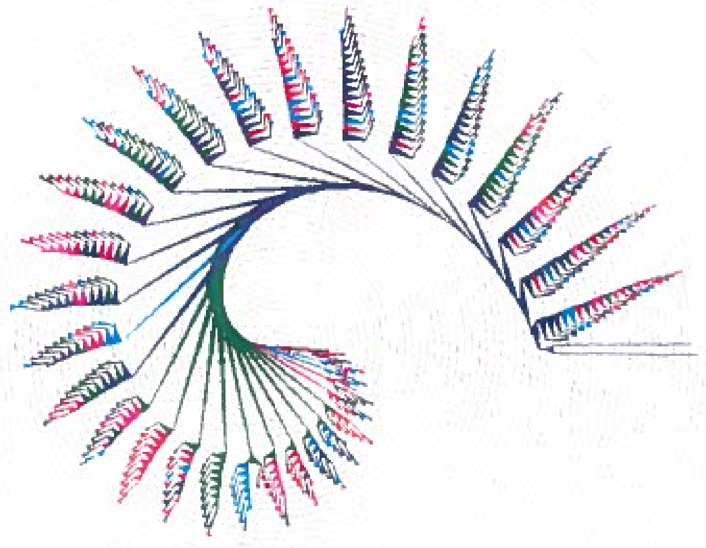

Figure 16: Execution Graph of Pascal's Triangle from PE View (5 PEs)

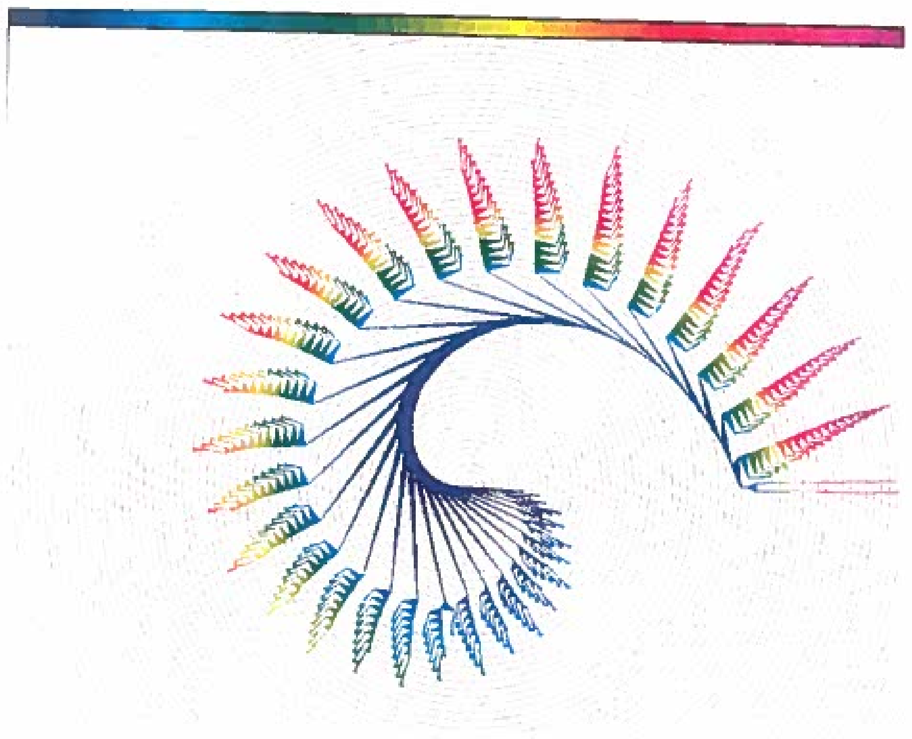
Figure 17: Execution Graph of Pascal's Triangle from Time View (5 PEs)

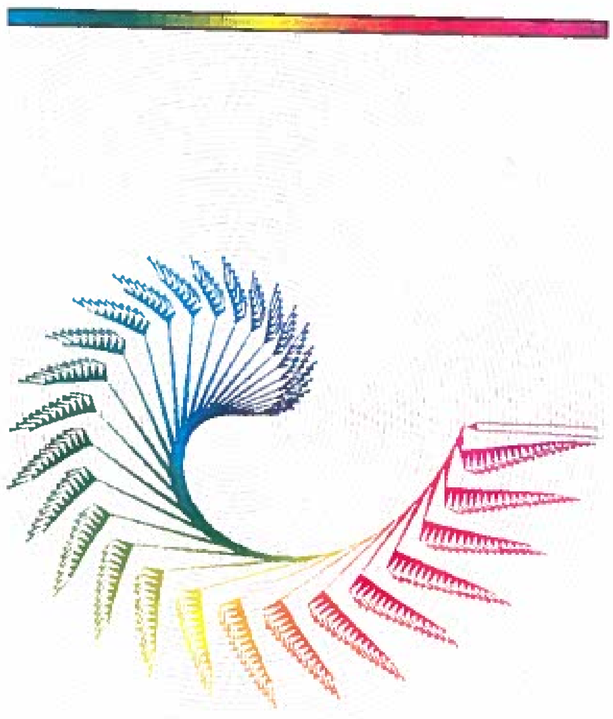
Figure 18: Execution Graph of Pascal's Triangle from Time View (1 PE)

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.

Figure 18 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. Figure 16 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 [17]. The benchmark program uses an unbalanced binary hash tree to store the vectors previously calculated so that lookups are efficient when computing the closure. Figure 19 (PE graph) and Figure 20 (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 $\times$ 850 and the level distance was four. The maximum tree height to be displayed is calculated as $h_{max} = \lfloor \frac{850}{4} \times 2 \rfloor = 106$. Level condensing is performed because $h > h_{max}$. The first 38 levels are not condensed, but the remaining 136 levels are condensed by the ratio of 2:1. This example demonstrates some strong points of VISTA:

- After level condensing, the tree keeps the shape of the original tree.

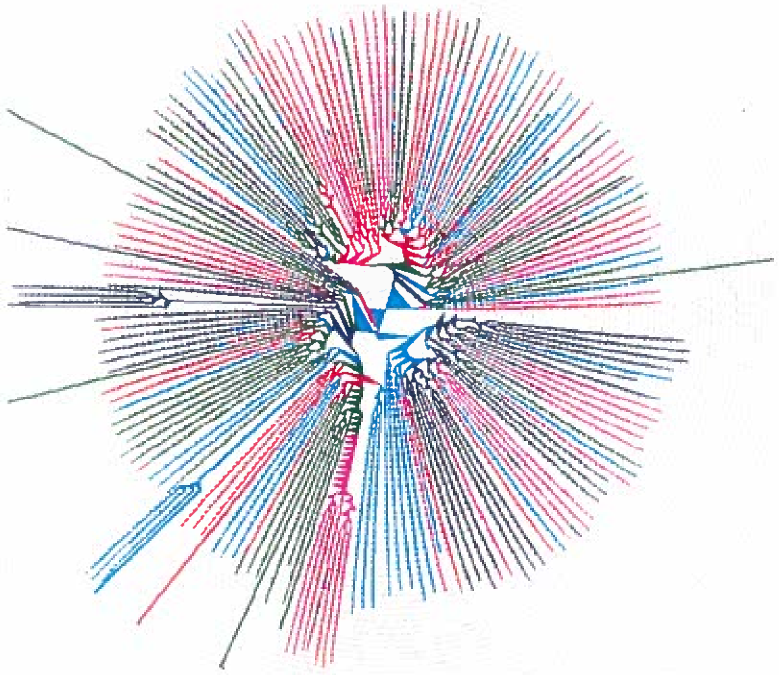
Figure 19: Execution Graph of Semigroup from PE View (5 PEs)

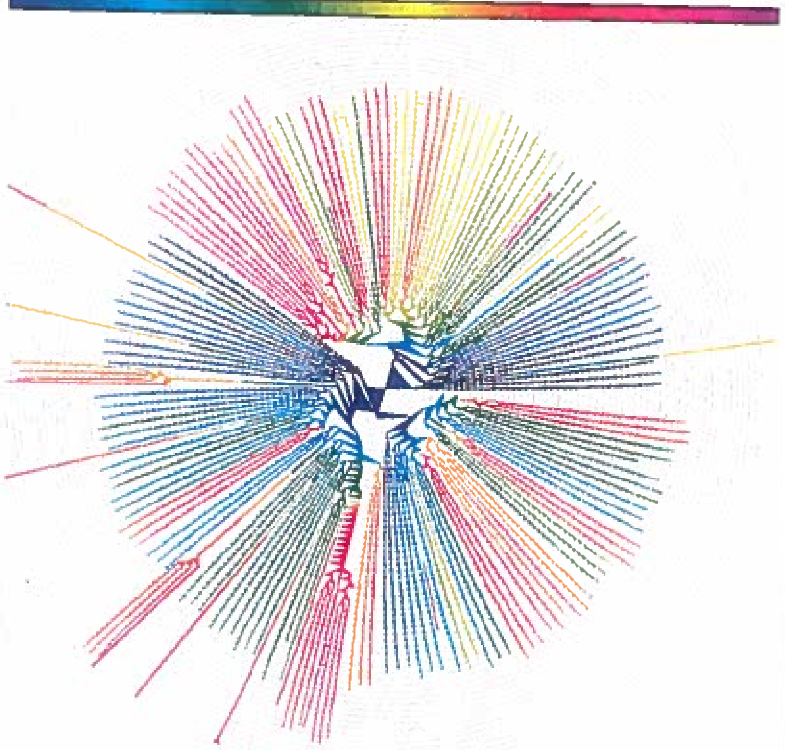
Figure 20: Execution Graph of Semigroup from Time View (5 PEs)

- The window-space efficiency is very good in this example. 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 Figure 20, 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 anti-clockwise 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 (Figure 19) is viewed in conjunction with the time graph (Figure 20), 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. Recall that subtrees are not condensed, and that the RGB spectrum is recycled starting at the subtree root, allowing detailed analysis.

The examples analyzed here are limited because of lack of space, and for facility of explanation. Intuitions gained for these programs have been confirmed by timing measurements [17]. Programs without as much parallelism, and on larger numbers of PEs, can be similarly analyzed in VISTA. 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.

5 Conclusions and Future Work

This paper described the concept and implementation algorithms of "kaleidescope visualization" of small-grain parallel programs. The VISTA system is an X-Window 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. 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. A critical question is what program characteristics will frustrate VISTA's heuristics, producing non-intuitive signatures. This will lead to the development of better use of shape, layout, and color to handle these cases.

Second, coloration methods for combining the time and processor views need exploration. We are currently developing a method of spectral superposition [1] that appears quite exciting. Related to this, we have already introduced limited animation when displaying the signature, in an effort to gain information from "seeing the flower blooming."

A third, ambitious project is to "invert" the tool, allowing user inquiry (via mouse) into a node in the signature, to *decompile* that event *back* into the source program. Such a facility can be used for scheduling, e.g., inserting pragma for distributing work at certain events.

Acknowledgements

E. Tick 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. We thank Al Malony for supplying color laser output.

References

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

[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 *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. Unpublished draft, Argonne National Laboratory MCS, June 1990.

[5] I. Foster, C. Kesselman, and S. Taylor. Concurrency: Simple Concepts and Powerful Tools. Technical Report MCS-P164-0790, Argonne National Laboratory, August 1990.

[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] M. T. Heath. PICL: A Portable Instrumented Communications Library. Technical Report TM-11130, Oak Ridge National Laboratory, July 1990.

[8] K. M. Kahn and V. A. Saraswat. Complete Visualization of Concurrent Programs and their Executions. Technical Report SSL-90-38, Xerox PARC, August 1990.

[9] E. Lusk *et al.* The Aurora Or-Parallel Prolog System. In *International Conference on Fifth Generation Computer Systems*, pages 819–830, Tokyo, November 1988. ICOT.

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

[11] D. Park. Visualizing Small-Grain Parallel Programs for Performance Evaluation. Master's thesis, University of Oregon, June 1991. Also available as Technical Report TR-91-14.

[12] 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.

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

[14] E. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[15] E.Y. Shapiro, editor. *Concurrent Prolog: Collected Papers*, volume 1,2. MIT Press, Cambridge MA, 1987.

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

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