VIPACT: A Visualization Interface for Analyzing Calling Context Trees

Huu Tan Nguyen*, Lai Wei†, Abhinav Bhatele‡, Todd Gamblin‡, David Boehme‡,
Martin Schulz‡, Kwan-Liu Ma*, Peer-Timo Bremer‡

*Department of Computer Science, University of California, Davis, California 95616 USA
†Department of Computer Science, Rice University, Houston, Texas 77251 USA
‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
*{htpnguyen, klma}@ucdavis.edu, †lai.wei@rice.edu, ‡{bhatele, tgamblin, boehme3, schulzm, ptbremer}@llnl.gov

***Abstract*—Profiling tools are indispensable for performance optimization of parallel codes. They allow users to understand where a code spends its time, and to focus optimization efforts on the most time consuming regions. However, two sources of complexity make them difficult to use on large-scale parallel applications. First, the code complexity of parallel applications is increasingly, and identifying the problematic regions in the code is difficult. Traditional profilers show either a flat view or a calling context tree view. Second, most tools average performance data across processes, losing per-process behavior. Diagnosing problems like load imbalance requires profiles from many processes, and manually analyzing profiles from hundreds or thousands of processes is infeasible. We introduce VIPACT, a visualization tool to identify different behaviors in multiple processes. VIPACT introduces "halo nodes" that concisely encode the distributions of runtimes from all processes, and a hybrid tree view that combines the advantages of calling context trees with those of flat profiles. We combine these with approaches such as ring charts, as well as the filtering and subselection of nodes, and we apply these techniques to a production multi-physics code.**

I. Introduction

Parallel simulation codes are expected to scale to ever larger processor counts as supercomputers continue to grow, which makes optimizing their performance increasingly challenging. A variety of tools and techniques have been developed to assist programmers in collecting and analyzing performance data. A common technique is to create a flat profile [1], which shows the aggregate runtime, invocation count, etc., associated with each function. This allows users to quickly identify the most heavily exercised portions of a code and to target optimization efforts. However, many functions, especially those in commonly used libraries, may be called throughout the code with different parameters or global state, and only some scenarios may exhibit performance problems. Flat profiles hide this information. To mitigate this, Ammons et al. [2] developed calling context tree (CCT) profiles, in which data is collected with respect to each stack frame. CCTs comprise the merged call-stacks from all functions invoked during the course of program execution, with each frame in the stack represented with a node in the tree, and with common prefixes merged. Typically, each stack in the CCT is rooted at *main*, and nodes are annotated with *inclusive* time (time spent in a node and its descendants) and *exclusive* time (time spent in only that node) [1], [3]. A CCT can differentiate a function called in different ways from different regions, but it also hides the effects of low cost functions called from many places, when the overall runtime of the function is high. Further, when performance data is aggregated across processes, an aggregate CCT or an aggregate flat profile still obscures problems occurring on a small subset of CCTs (one CCT per process), yet directly visualizing thousands of trees is infeasible.

While there exist a number of tools that present both flat profiles and CCTs to help analyze large-scale simulation codes [1] these tools typically show the flat profile and the CCT as two independent views. This requires users to cross-compare information such as global versus per call-stack runtime manually which is tedious and error prone. Further, most existing tools show a single aggregated tree or at most a small number of trees. This makes the task of comparing a large number of trees, which is important to find localized performance problems, difficult. To address these problems, we create VIPACT. VIPACT is a visualization tool that allows users to quickly find performance variations in ensembles of CCTs from many processes. This is a necessary step to identify load imbalance. Similar to the global approach, we map all CCTs onto a joint global tree to create a matching tree topology. However, VIPACT maintains the relevant per-process performance data separately which allows us to encode the distribution of measurements across all trees on each node. In addition, VIPACT provides a new hybrid view which combines the advantages of a flat profile with that of a CCT allowing users to understand both the context a function was called in and its runtime cost.

II. Related Work

The CCT is a hierarchical data structure used to store the calling contexts of an application [2], [4]. For initial explorations of a CCT, visualizations have been effective in conveying the hierarchical structure and accompanying information [4], [5], [6], [7]. The most common representation of a CCT is the node-link layout [8], in which each function invocation is represented as a node and the caller-callee relationship is represented as an edge. Traditionally,

performance data such as CPU runtime is encoded onto the node using shape, color, and/or size of the node [8], [9]. For example, DeRose et al. use bar charts as nodes to show load imbalance [10]. One important disadvantage of node-link layout is the amount of screen space required to present the CCT. However, if properly filtered we have found that node-link diagrams are easy to understand and can effectively convey the hierarchical structure of a CCT.

Another common method to show CCTs is using a radial layout. Moret et al. introduce the Calling Context Ring Chart (CCRC) [4], which uses the sunburst diagram [7] as the visual metaphor, which is an intuitive and scalable method to visualize and analyze data encoded at the nodes of a CCT. Each ring is partitioned into segments, where the number of segments is determined by the number of nodes at that particular level of the tree. The area of the segment is set by a metric. However, we found that the hierarchical structure of a CCT is not well represented using CCRC.

Another common visual encoding for trees and hierarchical data in general is Treemap which has been used as a space-filling encoding of CCTs [8]. Treemap is one of the more scalable methods to display hierarchical data. However, because only leaf nodes are directly visible, Treemap is not very effective in analyzing the hierarchical structure of the tree [4].

Various tools exist to present CCTs. For example, HPCToolkit [3] is an open source suite of tools for measuring and presenting performance data. `hpcviewer`, which is a part of HPCToolkit, allows users to analyze both flat profiles and CCTs. However, both views are independent of each other, requiring users to manually draw connections between them. Further, HPCToolkit presents a CCT as an expandable tree, which can take significant effort to browse. Trevis [11] is a tool that shows CCTs using both node-link layout and CCRCs. However, Trevis does not support flat profiles, which may hinder the analysis of some applications. In addition, although Trevis allows users to compare multiple trees, it does so by using a dissimilarity matrix and pairwise comparison, which is ineffective for large-scale applications where hundreds of trees need to be compared. To address issues with these existing tools, we have developed VIPACT, which is capable of showing hundreds of CCTs in a combined view and integrating information from the flat profile in a novel hybrid view.

III. Approach

VIPACT introduces two new concepts for users to quickly find performance variability among CCTs and to identify where the most opportunities for improvements lie. The first is the notion of a halo node, which maps the distribution of performance metrics across per-process CCTs to intuitive glyphs. Second is the Halo-DAG view, which integrates the flat profile into the tree to identify the runtime costs of heavily used functions. In addition, VIPACT includes commonly used tree operations like subtree selection, filtering, and means to browse individual call trees. Combined, these simplify the complexity of multi-process profiles of complex codes.

VIPACT is a web-based application implemented in Javascript and D3 to visualize the data via a browser. We use ColorBrewer [12], a tool for selecting color schemes for color maps. For the backend, a Node.js [13] server is used to perform calculations and sending data.

A. Halo Node

One of the most difficult performance bottlenecks to spot and correct are load imbalances in which a small number of processes may block all others from progressing. One of the best indicators of such problems is performance variability among processes in CCTs. In many applications, each process is ideally expected to perform very similar or even identical computations. Significant variations among processes may indicate, for example, load imbalance, network interference, or excessive system noise.

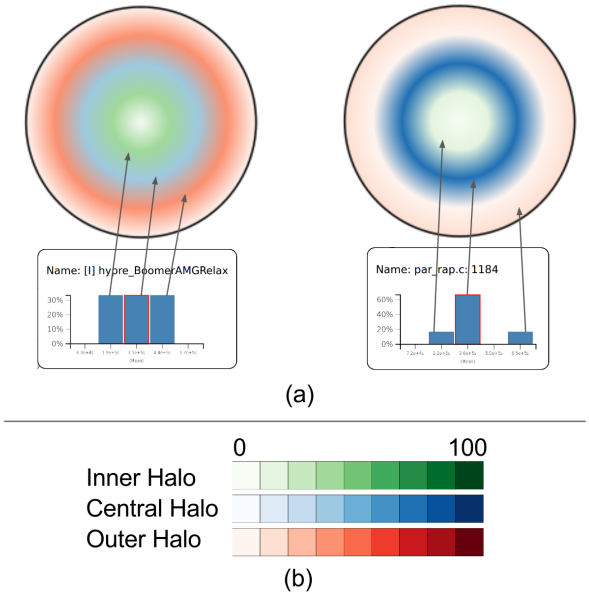

Fig. 1. Halo Node. (a) Two halos with different distributions. The left halo has an even color distribution, indicating work was not evenly distributed. The right halo has the mean as peak, this is typical when work was evenly distributed. (b) The three color scales for the inner, central, and outer halos.

To help users identify potential problems, we map all trees produced from an execution onto a single tree and encode the corresponding data via glyphs for each node. This allows us to scale our visualization to support large-scale parallel applications. Because the trees are generated from the same application, they can easily map onto a global tree structure as done by various tools [3] to create a global CCT. To show variation in runtime of each tree node, we render the nodes as halos. Figure 1 shows two halos along with their respective histograms and the color scale. The histogram is centered at the mean runtime of each node, a value we compute during the aggregation process. The left and right areas of the histogram represent runtimes that are shorter and faster than the mean respectively. In particular, we map the lower third of the histogram to the innermost halo, the middle third,

indicating runtimes close to the mean, to the middle halo and the remaining top third to the outermost halo to emphasize slow processes.

B. Directed Acyclic Graph Integration

Flat profiles allow users to see the total runtime cost associated with each function, across all invocations. This is important for identifying functions that are called from many different sites and consume significant runtime in aggregate. Math functions such as `sqrt` and `exp` often fall into this category. A traditional CCT can hide these functions, as their cumulative runtime may be spread across many different calling contexts (call stacks). Nonetheless, depending on the scenario, a CCT may be the best way to view these functions, especially if only certain call stacks make many or long-running calls to particular functions. To identify potentially problematic functions and their calling contexts, we integrate a Directed Acyclic Graph (DAG) into the node-link view with halo nodes and produce a new representation, which we call the Halo-DAG tree. For each function in the flat profile, we find the corresponding node in the node-link view. If the node is already displayed, we consider its information represented in the view and proceed. If it is not a part of the currently displayed CCT, we find all lowest visible ancestors and create an edge between them and a node at the bottom of the display representing the flat profile information of the function of interest. Figure 2(a) shows a call tree with the integrated DAG. Figure 2(b) shows a close-up of one of the functions in the DAG view. By showing two color schemes, users can see how much runtime that function has relative to other functions and relative to the overall runtime. The links coming out of the function in the DAG view also serve to identify the different contexts from which the function was called.

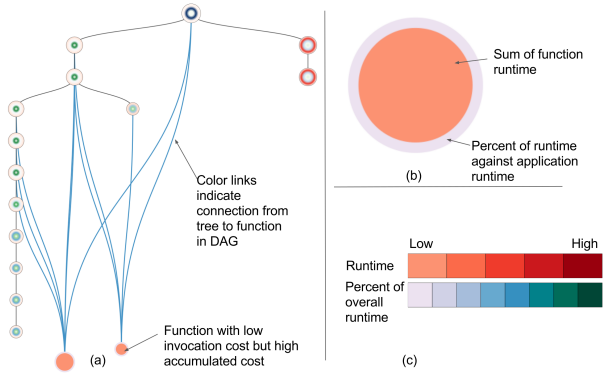

Fig. 2. (a) The node-link diagram with the integrated DAG view. The flat nodes (shown in solid color) are connected to the tree using blue colored links. (b) A close up of a function in the DAG. The center color shows the amount of runtime of that function relative to other functions. The outline shows the percent of function runtime against the overall runtime. (c) The two color scales for the runtime and percentage.

C. Tree Operations

Since a large majority of the nodes contain little or no information of interest, we include both filtering and subtree selection to help reduce the complexity of the tree. Node filtering is used to remove nodes that are below a threshold. This reduces the complexity of the tree and allows users to focus on important parts of the tree. Subtree selection allows users to drill down into the tree and gain additional insight regarding the data without visual cluttering. To help focus the user's attention, our tool uses hot-paths to highlight subtrees where the inclusive runtime is above a threshold value [1]. Together, these operations allow users to focus on sections of interest and drill down into the tree to see more detail.

IV. Case Study

In order to show the effectiveness of our tool, we perform a case study on Miranda, a radiation hydrodynamics simulation code developed at Lawrence Livermore National Laboratory (LLNL) [14]. We run Miranda on Cab (an Intel Xeon InfiniBand cluster at LLNL) using 256 MPI processes and use HPCToolkit version 5.4.2 to collect the performance data. We extract the global flat profile and per-thread CCTs from the database generated by `hpcprof-mpi` and feed them into our tool, VIPACT.

For the purpose of this study, we filter the flat profile to obtain the top 0.5% functions with the most runtime. By doing this, we get functions that contribute more than 0.6% to the overall runtime. We also filter the CCT to show only tree nodes with more than 1% inclusive runtime. This allows us to reduce the number of nodes rendered and to simplify analysis. Since Miranda is a highly-optimized, production application, we expect most nodes to be blue. Figure 3(a) shows the Halo-DAG tree and confirms our hypothesis. However, the visual cues provided by the halo nodes allow us to identify four sections of the tree with high variability in runtimes. When looking at the nodes' histograms, we notice that more than 60% of processes spend a significant amount of time in `pmpi_bcast_` as indicated by Figure 3(b). This suggests that there was a load imbalance or network interference issue which led to many processors being stuck in the broadcast.

Next, we look at the contributions of functions in the flat profile shown in the Halo-DAG tree. We find that `_int_malloc`, a memory allocation function, contributes 2.5% to the overall runtime with most branches of the tree invoking this function. Figure 3(a) shows the function with its links highlighted. Since `_int_malloc` is a system call with only 2.5% of the overall runtime, there is no need to optimize this function. However, the number of links coming from `_int_malloc` indicate that there were many calls to this function. Further investigation could determine whether the number of calls to `_int_malloc` can be reduced.

V. Summary and Future Work

In summary, we introduce VIPACT, a tool to analyze performance data in calling context trees of parallel applications. VIPACT introduces the concepts of halo node and integrated tree/DAG view. For future work, we plan to extend the Halo-DAG tree to allow users to select subsets of trees for analysis. This will allow users to focus on processes that are slow. We

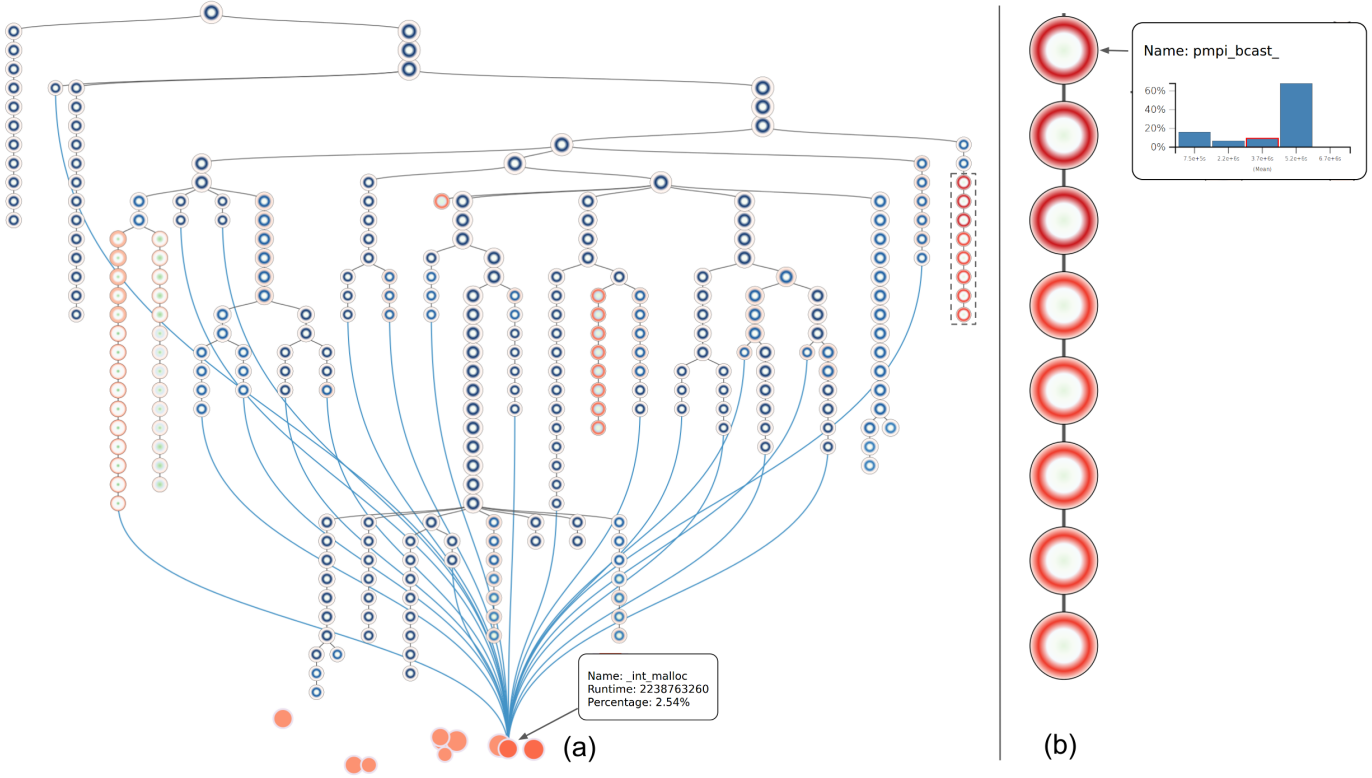

Fig. 3. Halo-DAG tree of Miranda. (a) There are four branches of interest based on the red halos. The histogram indicates that the branch with `pmpi_bcast_` has 60% of processes that are slower than the mean. At the bottom, the function `_int_malloc` was used in almost every branch of the tree and contributed to 2.5% of overall runtime. (b) Highlight the branch of `pmpi_bcast_`.

also plan to extend the view to show the amount of runtime each branch of the tree contributes to the function's runtime.

VI. Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-704659).

References

[1] L. Adhianto, J. Mellor-Crummey, and N. R. Tallent, "Effectively presenting call path profiles of application performance," in *2010 39th International Conference on Parallel Processing Workshops*. IEEE, 2010, pp. 179–188.

[2] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *ACM Sigplan Notices*, vol. 32, no. 5, pp. 85–96, 1997.

[3] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[4] P. Moret, W. Binder, A. Villazón, D. Ansaloni, and A. Heydarnoori, "Visualizing and exploring profiles with calling context ring charts," *Software: Practice and Experience*, vol. 40, no. 9, pp. 825–847, 2010.

[5] I. Herman, G. Melançon, and M. S. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Transactions on visualization and computer graphics*, vol. 6, no. 1, pp. 24–43, 2000.

[6] S. T. Teoh and M. Kwan-Liu, "Rings: A technique for visualizing large hierarchies," in *International Symposium on Graph Drawing*. Springer, 2002, pp. 268–275.

[7] J. Yang, M. O. Ward, and E. A. Rundensteiner, "Interring: An interactive tool for visually navigating and manipulating hierarchical structures," in *Information Visualization, 2002. INFOVIS 2002. IEEE Symposium On*. IEEE, 2002, pp. 77–84.

[8] K. E. Isaacs, A. Giménez, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, B. Hamann, and P.-T. Bremer, "State of the art of performance visualization," *EuroVis 2014*, 2014.

[9] B. Mohr and F. Wolf, "Kojak–a tool set for automatic performance analysis of parallel programs," in *European Conference on Parallel Processing*. Springer, 2003, pp. 1301–1304.

[10] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *European Conference on Parallel Processing*. Springer, 2007, pp. 150–159.

[11] A. Adamoli and M. Hauswirth, "Trevis: A context tree visualization & analysis framework and its use for classifying performance failure reports," in *Proceedings of the 5th international symposium on Software visualization*. ACM, 2010, pp. 73–82.

[12] M. Harrower and C. A. Brewer, "Colorbrewer. org: an online tool for selecting colour schemes for maps," *The Cartographic Journal*, 2013.

[13] S. Tilkov and S. Vinoski, "Node. js: Using javascript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, p. 80, 2010.

[14] W. Cabot, A. Cook, and C. Crabb, "Large-scale simulations with miranda on bluegene/l."