Visual Analytics Challenges in Analyzing Calling Context Trees

Alexandre Bergel[1], Abhinav Bhatele[2], David Boehme[2], Patrick Gralka[3], Kevin Griffin[2], Marc-André Hermanns[5], Dušan Okanović[4], Olga Pearce[2], Tom Vierjahn[6,7]

[1]Department of Computer Science, University of Chile, Santiago, Chile, abergel@dcc.uchile.cl
[2]Lawrence Livermore National Laboratory, Livermore, California, USA, {bhatele, boehme3, griffin28, olga}@llnl.gov
[3]Visualization Research Center, University of Stuttgart, Stuttgart, Germany, patrick.gralka@visus.uni-stuttgart.de
[4]Institute of Software Technology, University of Stuttgart, Stuttgart, Germany, dusan.okanovic@iste.uni-stuttgart.de
[5]JARA-HPC & Jülich Supercomputing Center, Jülich, Germany, m.a.hermanns@fz-juelich.de
[6]JARA-HPC & Visual Computing Institute, RWTH Aachen University, Aachen, Germany, tom.vierjahn@acm.org
[7]Westphalian University of Applied Sciences, Bocholt, Germany

***Abstract*—Performance analysis is an integral part of developing and optimizing parallel applications for high-performance computing (HPC) platforms. Hierarchical data from different sources is typically available to identify performance issues or anomalies. Some hierarchical data such as the calling context can be very large in terms of breadth and depth of the hierarchy. Classic tree visualizations quickly reach their limits in analyzing such hierarchies with the abundance of information to display. In this position paper, we identify the challenges commonly faced by the HPC community in visualizing hierarchical performance data, with a focus on calling context trees. Furthermore, we motivate and lay out the bases of a visualization that addresses some of these challenges.**

I. Introduction

The process of optimizing performance of parallel applications is an integral part of a successful software strategy in high-performance computing (HPC). However, the process of identifying performance bottlenecks and understanding behavioral phenomena in parallel applications is complex and tends to involve a problem-specific set of tools and visualizations. It is usually possible to identify some of the performance bottlenecks by solely examining metrics from a single application execution. However, properly identifying and finding complex bottlenecks depends on the ability to compare measurements from different executions, comparing different software versions or runtime configurations. Because the investigative process requires advanced, domain-specific knowledge, determining the right visualization to reveal relevant behavior to the experts is challenging.

Performance profile data can take various forms, as discussed in more detail in Section II. Often, the context for such data follows a hierarchical structure in potentially multiple dimensions (e.g., the Cube data model [1]). The calling context is often of particular interest to the performance expert, as it describes the structure of the application, and identifying a performance phenomenon in the context of the software helps in understanding it. A calling context tree (CCT) [2] is a compact summarization of the relationships between caller-callee entities in an application. Manipulating and reasoning upon the CCT is central to numerous performance engineering activities. As such, most code execution profilers produce performance reports anchored on a CCT or similar construct [3], [4], [5], [6], [7], [8].

High-performance computing applications can produce a large number of CCTs in multiple dimensions: (1) parallelism, (2) time, and (3) commit history. Evolution in terms of parallelism is seen when performance measurements across processing elements (processes, threads, etc.) or across different execution scales (number of nodes, cores, etc.) are compared. In this case, the CCT may contain an additional dimension with values for each processing unit, or each processing unit is associated with its own CCT. Evolution in terms of the time dimension is often reflected by tracking time steps or iterations in the application [9]. Evolution in terms of commit history involves comparing the performance of different versions of the code.

Visualizing CCTs is crucial for understanding the performance of HPC simulation codes, but due to the large scales in each of the dimensions mentioned above as well as the potentially large size of the CCT itself, visualization of the relevant data is challenging.

This paper describes a flexible framework to visualize HPC-related benchmark execution profile. The key aspects of our framework is to express the transition between various complementary interactive visualizations. The combination of these visualizations describes a flow that covers the CCT trees, the function graph, processes / computational units, and the metric list. Transitions between views is enabled by simply selected using the mouse some relevant visual elements.

In this paper, we outline the types of data that are stored using CCTs (Section II), the related work (Section III), and typical user operations on the data (Section IV). Subsequently, we propose a prototype for a CCT visualization framework that allows HPC experts to quickly key in on the underlying cause of performance issues at scale (Section V). Our conclusion and future close the papers (Section VI).

II. HPC Domain Data

Several kinds of performance data are collected in HPC with different purposes. Measurements of a single execution of one application can be recorded for analyzing the performance of this single execution. The measurement of two or more executions (on different process counts, on different architectures, etc.) can be recorded for doing performance comparisons of these executions. For regression analyses, we may compare historical performance data of a single application or an entire system (all applications executed during a period of time) over time.

The HPC community is interested in several sub-components of the hardware and software stack that contribute to execution time and/or energy consumption. Execution time of an HPC application may depend on:

- Time spent in serial computation
- Data movement in the memory hierarchy
- Communication on the network
- Input/output to the filesystem
- Overlap between different application phases/components
- Sharing of the network and I/O resources by multiple jobs/applications

Since there are many potential sources of performance degradation, and it is difficult to attribute performance characteristics to the components listed above, there is not a definitive guide on performance engineering in HPC. Instead, whether comparing datasets from different executions or searching for bottlenecks in a single execution, the performance analysis process depends on advanced, domain-specific knowledge of the performance-engineering experts, and is usually tool-specific, hardware-specific, and/or problem specific.

III. State of the Art

Current production performance tools typically use straightforward means to represent CCTs. Cube and hpcviewer, the performance data browsers for Score-P [3] and HPCToolkit [10], respectively, use text-based tree views or tree tables similar to those often found in file browsers. Figure 1 shows an example of the Cube callpath display. Data spanning multiple CCTs (e.g., multiple processes in a parallel application) is shown on a unified tree using aggregate values. These tree views are easy to implement for tool developers and easy to interpret for users. VIPACT shows a hybrid tree and flat profile view with "halo nodes" showing the distribution of runtimes across processes [11].

Despite its simplicity, there are drawbacks to the traditional tree view presentation:

- All tree nodes have the same size, making it difficult to visually distinguish interesting nodes in the CCT from uninteresting ones. Coloring helps to some extent, but a large number of uninteresting nodes can clutter the display.
- Related functions or function groups (i.e., common subtrees) are visually separated over possibly large distances.

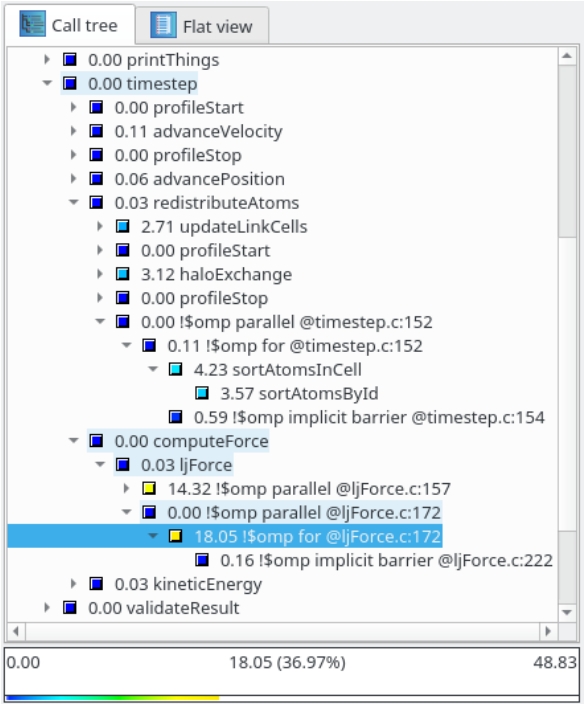


Fig. 1: Traditional callpath display in the Cube profile viewer.

- Typical tree views show only scalar values for tree nodes, limiting the kind of information that can be displayed.
- Laborious interaction: Typically, users must expand each tree branch individually, often requiring lots of clicks to reach nodes of interest. However, both cube and hpcviewer implement a "hotpath" option which automatically expands the most expensive sub-branch.

An important insight here is that a tree view is not necessarily the ideal form to present CCTs. We therefore explore more compact visualizations that better highlight portions of interest.

In the context of memory performance analysis, Gralka et al. [12] presented a tool to visually explore detrimental memory access patterns. Besides a scatterplot where each low-level memory access is depicted as an individual point, they show the call tree as a flame graph [13]. Such a representation of a call tree encodes hierarchy but also duration, and lends it use in trace-based visualizations, such as HPCTraceViewer [14] and Vampir [15]. In principle, a flame graph is a variation of Kruskal's icicle plot [16]. The techniques presented by Trümper et al. [17] and De Pauw et al. [18] use tree visualizations of this type, as well. A similar visualization are indented trees, for instance used by De Pauw et al. [19], [20]. The disadvantage is that these visualizations consume a significant portion of screen space. Another kind of tree visualizations are radial representations, which have been used by Adamoli and Hauswirth [21], and Moret et al. [22] to depict CCTs. In a radial representation, functions with a relative small self-time can degenerate from radial boxes to small lines in the visualization, which we expect to occur quite often in the context of simulation software with an interactive design.

However, we adopted the basic idea of a radial representation to augment the nodes in our graph-based alternative *Function View* with an arc of radial boxes showing the share of execution time with respect to a specific calling function.

IV. Data/Visual Analytics Operations on a CCT

We deem a specific set of user operations necessary that a visual analytics tool should support in order to help users in the analysis of CCTs. An important aspect of CCTs that needs to be managed by user operations is their size and scale. **Filtering** helps to reduce the size of CCTs and helps the user to focus on the interesting parts of the tree. This requires a tool to expose selectable metrics and thresholds, or queries to have the user communicate interesting nodes for subselection to the tool.

In the same context, **grouping** is a helpful operation, as well. Navigating through a large CCT is difficult. If the user could group the nodes in the tree, for instance with respect to their name or load module (library), the tree would resemble something that the user is more familiar with, such as the general architecture of the software. Especially in the context of computational science and engineering codes, we expect to have repeating patterns, for instance from iterative solvers, in a CCT. A visual analytics tool should provide the means to group the nodes of a pattern into a single entity and to unfold them again depending on whether the user wants to gain an overview of the tree or wants to drill down into details. This operation is effective only if analyzed metrics can be aggregated into the grouped entity.

Another set of operations should help the user to cope with multiple trees, e.g., from different runs, or different inputs per run. The basis for most of these operations is a **matching** between the nodes in two or more trees. Most of this matching should be done automatically. However, in very complex trees we expect automatic matching to fail. Unfortunately in such cases, a user-driven matching on the raw data will be often too time-consuming. A user-operated **clustering** can help a user during manual matching to focus on more dissimilar parts in the trees. Adamoli and Hauswirth [21] provide a list of methods and metrics for clustering and comparison of CCTs.

In general, an important hint towards interesting behavior that a user wants to analyze are the differences in trees according to one metric or a set of metrics, or topological changes. Thus, a visual analytics tool should support the **union** or **subtraction** of two or more matched CCTs and highlighting according to detected similarities or differences.

For different optimization strategies (performance, throughput, power consumption, etc.) the user will need the capability to **overlay** different metrics on the nodes of a CCT. For example, after matching or subtracting CCTs, being able to select and view different metrics on the highlighted node(s) is crucial for understanding the performance of HPC simulations.

V. Prototype of a Flow-Based Visualization Framework

This section outlines a framework for visualizing the execution of HPC applications. The framework allows for several visualizations to be hooked together and is able to support a navigation flow between them. In section IV, we described a set of user operations a comprehensive framework should support. Note that, given the outlined framework is a prototype, not every aspect of these operations is included in the following framework. However, we encourage the reader to extend our framework to a tool ready for production use.

A. Flow-based Navigation

Figure 2 gives a high level representation of the flow supported by our framework. The flow is modeled as a directed acyclic graph of four nodes. Each node represents a family of views on the data commonly considered when dealing with performance assessment and performance correction activities. In particular, we support the following views.

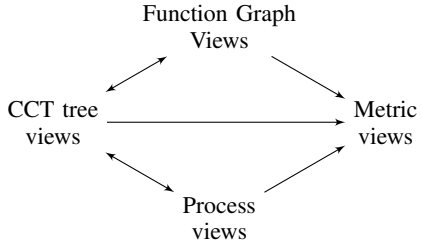


Fig. 2: Flow supported by our framework

CCT tree views: A CCT is a considered as a standard and intuitive representation of a program execution. This node describes visualizations of a CCT. A CCT may be large, which may turn a simple visualization ineffective. In particular, these views may filter out irrelevant parts of the CCT (*e.g.,* the use of a particular library or architectural layer), and fine-tune the visualization (*e.g.,* by using a particular or customized tree layout, user-defined color mapping to highlight some properties of each node). Therefore, the CCT tree view supports the *filtering* and *grouping* operations mentioned in Section IV. An example of a CCT tree view is given in Figure 4.

Function graph views: CCT may be verbose, particularly in presence of loops. Considering the graph of function calls may be relevant for some activities (*e.g.,* debugging, code maintenance, code understanding). Being able to visualize functions calls complements the *CCT tree views*. Such views have to consider the fact that function calls may form graphs, possible with cycle. The view should therefore accommodate such characteristics. Three examples of function graph views are given along the paper (Figure 5, Figure 6, and Figure 9).

Process views: HPC applications typically run over a large number of execution units, typically CPU and GPU cores. Focusing on the execution units is relevant to characterize the use of the available resources. A process view may support a particular HPC-related activity. For example, measuring the load balancing across the execution units or identifying underused units. This can be seen as a *clustering* to gain an overview as described in Section IV. An example of a process view is given in Figure 7.

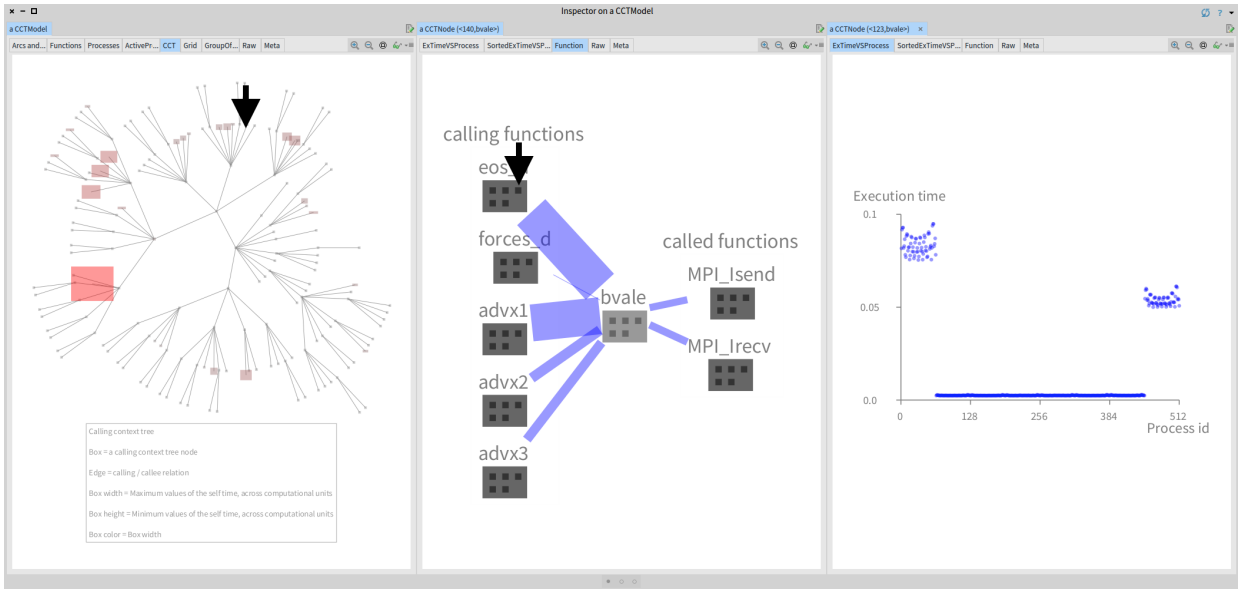


Fig. 3: Flow example for an execution of the ZeusMP/2 benchmark. Selected elements are indicated with a thick black arrow.

Metric views: An execution is accompanied with numerous metric reports to characterize, e.g., memory consumption, CPU uses, and cache uses. Numerical reports should be adequately presented using state-of-the-art visual representations. These views relate to the *overlay* operation. However, it requires links to the other views, especially the tree and function views.

This position paper claims that *manipulating these views in an explicit fashion and expressing multiple flexible flows is key to incrementally build flexible and open analyzing HPC tools.*

B. A First Flow Example

We illustrate the use of our framework on the basis of a Cube measurement report of the execution of the ZeusMP/2 benchmark [23] of the SPEC MPI 2007 benchmark suite [24] on 512 processes of the IBM Blue Gene/P supercomputer JUGENE [25], formerly operated by Forschungszentrum Jülich GmbH, Germany.

Figure 3 presents a flow made of the path *CCT Tree view* → *Function view* → *Metric view*. On the left hand side, the figure shows the calling context tree represented as a radial visualization. The shape and the color of each node correspond to some particular metrics. Clicking on the node indicated with the black arrow, in the most-left pane opens the second pane showing a function view. In this new view, clicking on the context indicated with the black arrow opens the third pane, a metric view.

As presented in Figure 4, the CCT view uses a polymetric view [26] in which each box represents a CCT node and an edge represents a calling-callee relation. The height of a box represents the minimum value of the self-time across all the processes. The width of a box, as well as its color, represents the maximum value of the self-time, across all the processes. Size of a node therefore indicates its significance regarding the overall consumption share.

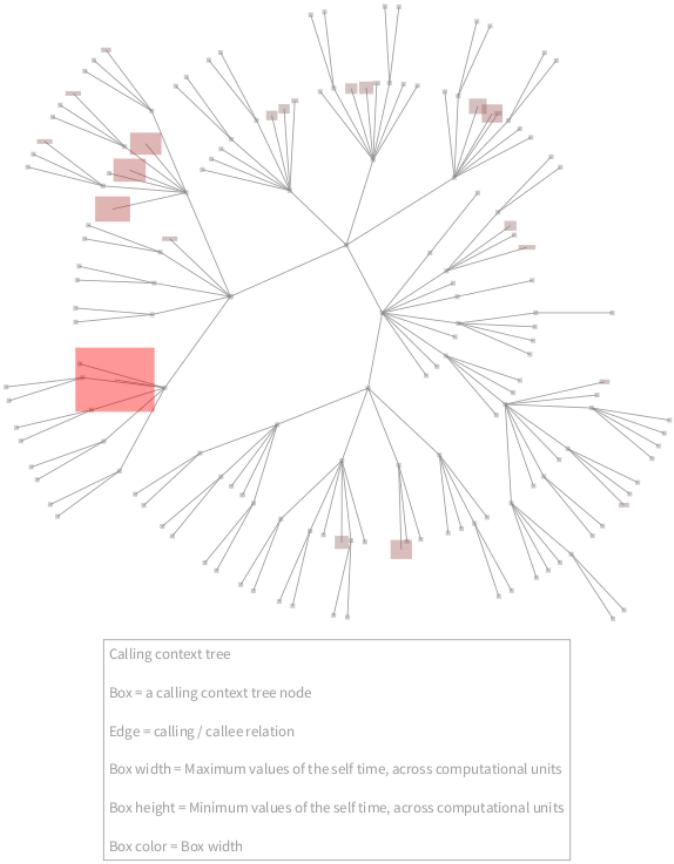
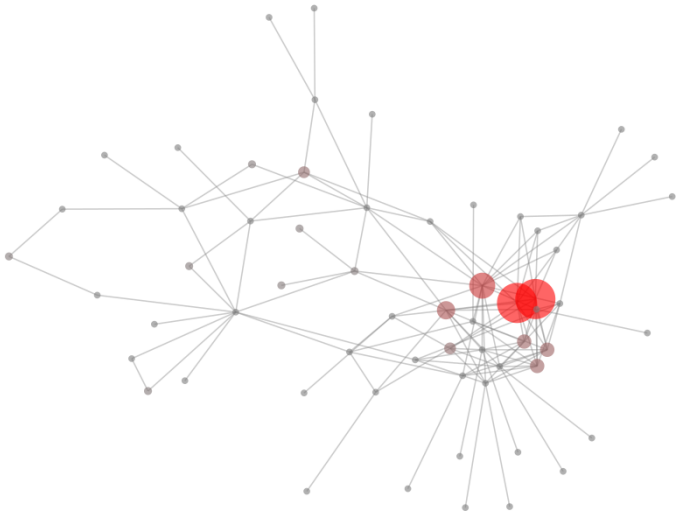


Fig. 4: Example of a CCT view

The overall function call graph is given in Figure 5. Each function is represented as a circle. The size of a circle indicates the number of CCT nodes of that function contained in the CCT



Graph of functions invocations

Circle = a function

Circle size = color = number of CCT node corresponding to the function

Green circle = functions invoked by the selected function

Edge = call between function

Fig. 5: Function view

tree. Edges are not visually directed, however, an interactive tooltip indicates caller and callee functions.

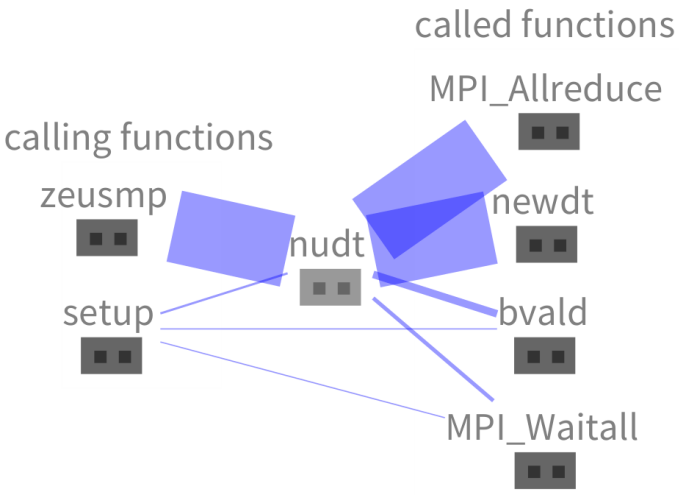
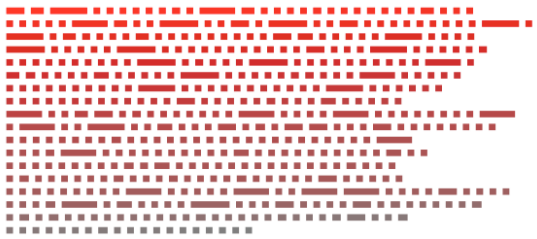


Fig. 6: Function view

Clicking on a CCT node opens a window pane that shows a function graph. Figure 6 illustrates functions, represented as outer boxes. Each function contains the CCT nodes of the encapsulating function. The figure is obtained by clicking on a CCT of the `nudt` function. We see that `nudt` has created two CCT nodes since the box representing `nudt`, at the center, has two small inner boxes. Each of these inner boxes represents a CCT nodes. Callers of the `nudt` are located left of it, and functions called by `nudt` are located on the right hand side. Edges indicate the control flow between the functions. The width of the edges indicates the number of calls.



This visualizations indicates how processing units are used

Box = a processing unit (e.g., a core, a thread)

Box height = the minimum value of consumption

Box width = the maximum value of consumption

Average of consumption (gray = low consumption, and red = high)

Fig. 7: Process view

Figure 7 illustrates a process view. In this visualization, each box represents a process / computational unit. Three metrics are used to represent a process: the width indicates the maximum self-time value across all CCT nodes, the height indicates the minimum value, and the color fading represents the average. The figure clearly shows that the usage of the computation units is not homogeneous. A grid layout is used to order the boxes.

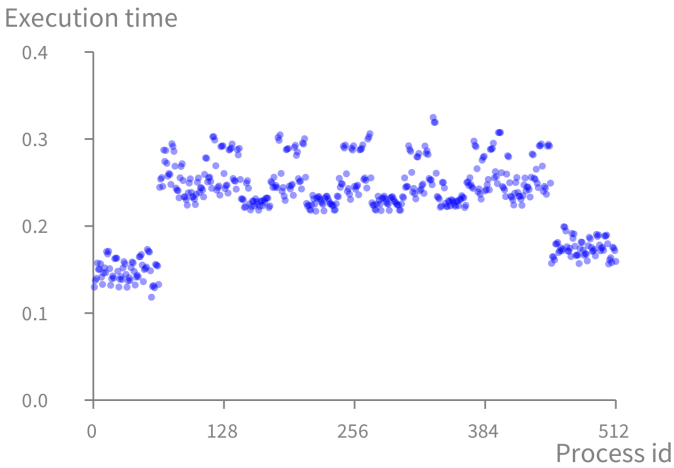


Fig. 8: Metric view

Clicking on a CCT node, either from the *CCT view* or from the second *Function view* opens a *Metric view*. It indicates the execution time of the selected entity across all the processing units. Figure 8 illustrates the execution time of a particular CCT node across all the computational processes.

The example of the flow supports cyclic function call graphs. Although convenient and intuitive, the visualization can be improved in case that the call graph has no cycles, as shown in our alternative function view, described below.

Detail about each visual element is available at all time. First a mouse tooltip indicates the relevant information, including the node name and the relevant metrics. The tooltip appear by moving the mouse above the visual element one wish to have information from. Clicking on a node trigger a new view. The new spanned view may be optionally replaced with a the complete list of associated metrics (accessible from the *Raw* tab, in Figure 3).

C. Alternative Function View

In the case that the function call graph is simpler, we propose an alternative *Function view*, which gives more detail on the causal-effect of self-time. The visualization was generated on the basis of a Cube measurement report [27] of the execution of the Sweep3D benchmark [28] on 294,912 processes of the IBM Blue Gene/P supercomputer JUGENE [25], formerly operated by Forschungszentrum Jülich GmbH, Germany.

Figure 9 represents a function as a circle. The size of the circle represents the self/exclusive execution time. Edges represent calls between functions. In this scenario, the control flow goes from the left to the right of the figure. Functions with a significant amount of exclusive time (*i.e.,* large circle) have their names on it. Less significant functions have no name in the visualization (however the name remains visible via tooltips).

Each large function has an arc around it. Each portion of this arc represents the share of that function's execution for a given calling function. The share is indicated with the size and the color of the arc. In Figure 9, each individual call to a function is drawn separately. However, the calls can be bundled in order to reduce clutter. While, for instance, grouping only the calls that contribute an average share of execution time – i.e., those that behave similarly – the outliers will remain visible and call for attention.

This *Function view* is more detailed than the previous one (Figure 5). However, its applicability if the data contains cycles or large number of nodes still needs to be evaluated on more data.

D. Data and Visualization Challenges

The typical visual analytics challenges, for instance described by Keim et al. [29], [30], apply as well to the analysis of CCTs. This section revises these challenges with respect to our framework.

Scalability. The **issue of scale** has more than one aspect. First, CCTs themselves can become very large if we analyze complex applications such as parallel simulation software (e.g., SPH [31], [32]). Such large trees, with potentially hundreds of thousands of nodes, require a sensible pre-processing step prior to visualization to prevent visual overload. Second, many issues we want to address require the comparison of CCTs across processes, timesteps, application runs, or input decks, which may involve a large number of trees. Moreover, there is no guarantee that CCTs of different processes, timesteps, or application runs are identical, even for identical inputs. Thus, the trees have to be matched, requiring heuristics or user interaction to deal with structural differences in the trees.

The first aspect remains open, as we assume the CCTs to be readily available for the described protoype for the sake of concisenes of its presentation. To address the second aspect, our framework promotes the uses of relatively simple visualizations for which practitioners can easily jump from one to another.

The flow supported by our framework is based on the "Visual Information Seeking Mantra" formalized by Shneiderman [33]. It consists of supporting a sequential flow of actions: first getting an overview and then zooming and filtering with details on demand. This mantra is a recognized way to design advanced graphical user interfaces.

Indeed, the user can start exploring the visualization from the *CCT view* and drill down into the overall execution information by moving into other views.

The visualization uses visual cues and some elementary interaction to cope with the exploitation of large visualizations. In particular, nodes are translucent to avoid occlusion in presence of overlapping, nodes can be drag-and-dropped, outgoing nodes are highlighted when locating the mouse on a particular nodes, and the view can be zoomed-in and out.

Interaction. The second challenge is to provide **suitable interaction possibilities** to the user, as user feedback is an integral part of each step in the visual analytics model. For the analysis of a CCT, a visualization has to support the user in typical analysis tasks, such as finding outliers, comparing trees, selection and comparison of sub-trees, etc. A thorough user interaction model requires the visualization and ideally the data analytics, as well, to be interactive. This involves a careful trade-off between presenting as much useful information as possible and maintaining interactivity.

Given that the presented visualizations are prototypes, the interaction design is not yet complete. However, we already provide some means to interact with the visualizations. In particular, mouse hovering reveals detail about a particular node. This simple ability therefore removes the need to label each visual element in our visualization. Clicking on an element opens a new pane on the right. Tabs are useful to switch from multiple views within the same class (*e.g.,* as the three function views given in Figure 5, Figure 6, and Figure 9).

User experience. An adequate user experience is key to favor the acceptance of a visualization into the typical workflow of performance engineers.

Currently, our prototype is not empirically validated. We did run pilots on a number of benchmarks, both available from the public domain such as Zeus, and proprietary ones. These pilots were crucial to improve the overall experience of the visualizations. As a future work, we plan to carry out empirical evaluation of the framework. In particular, controlled experiments and case studies are two experimental designs that

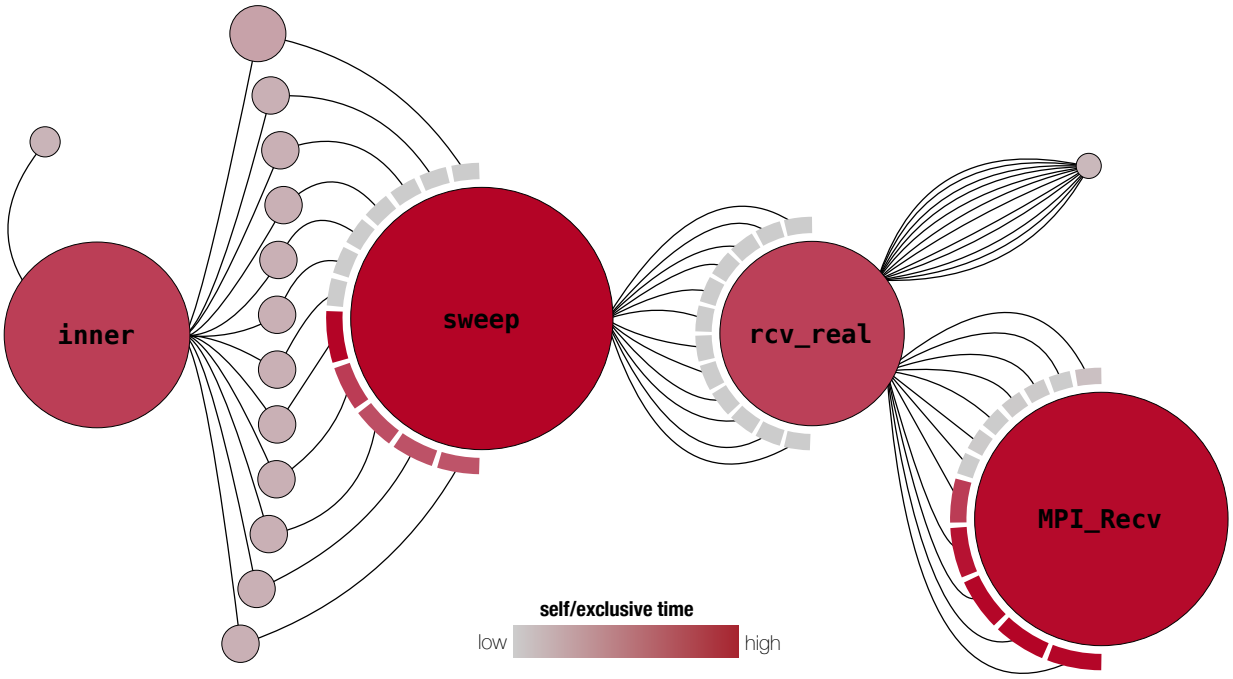


Fig. 9: Second example of the Function view: calls are depicted as a directed acyclic graph with calling-direction from left to right. Currently, each call is depicted separately. However, these can be bundled according to some user-defined statistics, with only the outliers still being drawn individually.

seems adequate to evaluate the experience and performance of performance engineers.

Semantics. Finally, deriving **semantics** and augmenting a visualization with that information to support the user is a challenge, as well. We have to rely on knowledge from performance engineering to find suitable metrics to facilitate automated derivation of semantics from the performance data of simulation runs. A visualization can provide several means to display semantic information. We can utilize color, area, or size, but also additional visualizations that are interactively linked to a CCT.

Our framework is able to integrate new visualization and customization of existing visualizations. The framework is implemented in Roassal [34], an agile framework to build visualizations.

VI. Conclusion

In this paper, we provide an overview of performance data available in HPC, as well as the challenges encountered in their visualization. Particularly, we focus on visualization of CCTs, which are used in performance analysis of parallel codes. Since traditional tree representation is not suitable for large sets of data common in HPC, we propose a flow-based framework for visualizing the execution of HPC applications. The advantage of such an approach is that it connects several visualizations, and facilitates interaction by providing a navigation flow between them. At the time of writing this paper, our visualization framework is a prototype that requires an empirical evaluation. In particular, we envision the framework to enable expressing constructions to handle *scalability* and *interaction*.

Future work will focus on extending the framework with different visualization approaches and connecting them to source code, as well as allowing domain scientists to evaluate the framework. Since CCTs are used not only in HPC performance analysis, but also in software performance engineering [35], we plan to evaluate this framework in both settings. In addition, we plan to carefully evaluate the expressiveness of our approach by conducting empirical evaluations.

Acknowledgment

The ideas presented in this paper originated during the GI-Dagstuhl Seminar 18283, sponsored by the Gesellschaft für Informatik e.V. (GI), where all the authors on this paper were participants. The first author would like to thank LAM Research for its financial support.

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG) in context of SFB 716, project D.3, as well as the Priority Programme "DFG-SPP 1593: Design For Future—Managed Software Evolution" (HO 5721/1-1), and by the Excellence Initiative of the German federal and state governments. 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-756548).

References

[1] Pavel Saviankou, Michael Knobloch, A. Visser, and Bernd Mohr. Cube v4: From performance report explorer to performance analysis tool. *Procedia Computer Science*, 51:1343–1352, June 2015.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, 1997.

[3] Dieter an Mey, Scott Biersdorff, Christian Bischof, Kai Diethelm, Dominic Eschweiler, Michael Gerndt, Andreas Knüpfer, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Christian Rössel, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P: A unified performance measurement system for petascale applications. In *Proc. of the CiHPC: Competence in High Performance Computing, HPC Status Konferenz der Gauß-Allianz e.V., Schwetzingen, Germany, June 2010*, pages 85–97. Gauß-Allianz, Springer, 2012.

[4] Martin Schulz, Jim Galarowicz, and William Hachfeld. Open, speedshop: Open source performance analysis for linux clusters. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[5] Laksono Adhianto, S. Banerjee, Michael W Fagan, Mark Krentel, G. Marin, John M. Mellor-Crummey, and Nathan R. Tallent. HPC-TOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[6] David Böhme, Todd Gamblin, David Beckingsale, Peer-timo Bremer, Alfredo Giménez, Matthew P Legendre, Olga Pearce, Martin Schulz, and Alfredo Gimenez. Caliper: Performance Introspection for HPC Software Stacks. In John West and Cherri M Pancake, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2016, Salt Lake City, UT, USA, November 13-18, 2016*, page 47. ACM, 2016.

[7] Juan-Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–9, September 2013.

[8] Alison Fernandez Blanco, Juan-Pablo Sandoval Alcocer, and Alexandre Bergel. Effective visualization of object allocation sites. In *Proceedings of 6th IEEE Working Conference on Software Visualization (VISSOFT '18)*, 2018.

[9] Zoltán Szebenyi, Brian J. N. Wylie, and Felix Wolf. SCALASCA parallel performance analyses of SPEC MPI2007 applications. In *Proc. of the 1st SPEC International Performance Evaluation Workshop (SIPEW), Darmstadt, Germany*, volume 5119 of *Lecture Notes in Computer Science*, pages 99–123. Springer, June 2008.

[10] 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 http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, 2010.

[11] Huu Tan Nguyen, Abhinav Bhatele, Peer-Timo Bremer, Todd Gamblin, Martin Schulz, Lai Wei, David Boehme, and Kwan-Liu Ma. VIPACT: A visualization interface for analyzing calling context trees. In *Proceedings of the 3rd Workshop on Visual Performance Analysis*, VPA '16, November 2016.

[12] Patrick Gralka, Christoph Schulz, Guido Reina, Daniel Weiskopf, and Thomas Ertl. Visual exploration of memory traces and call stacks. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 54–63, Sept 2017.

[13] Brendan Gregg. The flame graph. *Communications of the ACM*, 59(6):48–57, 2016.

[14] Nathan R. Tallent, John Mellor-Crummey, Michael Franco, Reed Landrum, and Laksono Adhianto. Scalable fine-grained call path tracing. In *Proceedings of the International Conference on Supercomputing*, ICS '11, pages 63–74, New York, NY, USA, 2011. ACM.

[15] Wolfgang E. Nagel, A. Arnold, M. Weber, Hans-Christian Hoppe, and Karl Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[16] J. B. Kruskal and J. M. Landwehr. Icicle Plots: Better displays for hierarchical clustering. *The American Statistician*, 37(2):162–168, 1983.

[17] Jonas Trümper, Alexandru Telea, and Jürgen Döllner. ViewFusion: Correlating structure and activity views for execution traces. In *Proceedings Theory and Practice of Computer Graphics*, pages 45–52, 2012.

[18] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John Vlissides, and Jeaha Yang. *Visualizing the execution of Java programs*, pages 151–162. Springer, Berlin, Heidelberg, 2002.

[19] Wim De Pauw and Stephen Heisig. Visual and algorithmic tooling for system trace analysis: a case study. *ACM SIGOPS Operating Systems Review*, 44(1):97–102, 2010.

[20] Wim De Pauw and Steve Heisig. Zinsight: A visual and analytic environment for exploring large event traces. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS, pages 143–152, New York, NY, USA, 2010. ACM.

[21] Andrea Adamoli and Matthias 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*, SOFTVIS, pages 73–82, New York, NY, USA, 2010. ACM.

[22] Philippe Moret, Walter Binder, Alex Villazón, Danilo Ansaloni, and Abbas Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Software: Practice and Experience*, 40(9):825–847, 2010.

[23] David Böhme, Markus Geimer, Felix Wolf, and Lukas Arnold. Scalasca analysis report for SPEC MPI.2007 benchmark 132.zeump2 on 512 processes in virtual- node mode on Blue Gene/P. Available at: https://doi.org/10.5281/zenodo.1211448, April 2018.

[24] Matthias S. Müller, Matthijs Van Waveren, Ron Lieberman, Brian Whitney, Hideki Saito, Kalyan Kumaran, John Baron, William C. Brantley, Chris Parrott, Tom Elken, Huiyu Feng, and Carl Ponder. SPEC MPI2007–an application benchmark suite for parallel systems using MPI. *Concurrency Computation Practice and Experience*, 22(2):191–205, 2010.

[25] Norbert Attig, Jutta Docter, Wolfgang Frings, Johannes Grotendorst, Inge Gutheil, Florian Janetzko, Olaf Mextorf, Bernd Mohr, Michael Stephan, Klaus Wolkersdorfer, Lothar Wollschlger, Stefan Krieg, Thomas Lippert, and Jeffrey S. Vetter. *Blue Gene/P: JUGENE*, pages 153–188. Computational Science Series. CRC Press, Taylor & Francis Group, Boca Raton, FL, USA, 2013.

[26] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, 2003.

[27] Brian J.N. Wylie, Markus Geimer, Bernd Mohr, David Bhme, Zoltn Szebenyi, and Felix Wolf. Scalasca analysis report of the ASCI Sweep3D benchmark on 294,912 processes in virtual-node mode on IBM Blue Gene/P with manually annotated iterations, August 2018.

[28] Los Alamos National Laboratory. ASCI SWEEP3D v2.2b: 3-dimensional discrete ordinates neutron transport benchmark. Available at: http://wwwc3.lanl.gov/pal/software/sweep3d/, 1995.

[29] Daniel Keim, Florian Mansmann, Jörn Schneidewind, Jim Thomas, and Hartmut Ziegler. *Visual Data Mining: Theory, Techniques and Tools for Visual Analytics*, chapter Visual Analytics: Scope and Challenges, pages 76–90. Springer Berlin Heidelberg, 2008.

[30] Daniel Keim, Gennady Andrienko, Jean-Daniel Fekete, Carsten Görg, Jörn Kohlhammer, and Guy Melançon. *Information Visualization: Human-Centered Issues and Perspectives*, chapter Visual Analytics: Definition, Process, and Challenges, pages 154–175. Springer Berlin Heidelberg, 2008.

[31] AJC Crespo, Benedict Rogers, JM Dominguez, and M Gomez-Gesteira. Simulating More Than 1 Billion SPH Particles Using GPU Hardware Acceleration. *Simulating More Than 1 Billion SPH Particles Using GPU Hardware Acceleration*, pages 249–254, 2013.

[32] Kevin Griffin and Cody Raskin. Scalable Rendering of Large SPH Simulations Using an RK-enhanced Interpolation Scheme on Constrained Datasets. In *Large Data Analysis and Visualization (LDAV), 2016 IEEE 6th Symposium on*, pages 95–96. IEEE.

[33] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *IEEE Visual Languages*, pages 336–343, College Park, Maryland 20742, U.S.A., 1996.

[34] Alexandre Bergel. *Agile Visualization*. LULU Press, 2016.

[35] M. Woodside, G. Franks, and D. C. Petriu. The future of software performance engineering. In *Future of Software Engineering (FOSE '07)*, pages 171–187, 2007.