Scalable Comparative Visualization of Ensembles of Call Graphs using CallFlow

Suraj P. Kesavan*, Harsh Bhatia†, Abhinav Bhatele‡, Stephanie Brink†, Olga Pearce†,
Todd Gamblin†, Peer-Timo Bremer†, Kwan-Liu Ma*

*Department of Computer Science, University of California, Davis, CA 95616 USA
†Lawrence Livermore National Laboratory, Livermore, CA 94551 USA
‡Department of Computer Science, University of Maryland, College Park, MD 20742 USA

I. Introduction

Large-scale parallel applications require significant optimization efforts to understand how different application parameters and/or initial conditions may affect the performance. These efforts involve detecting performance bottlenecks and performing appropriate code fixes. In the process, experts conduct a variety of test runs by varying multiple configurations to identify optimal execution parameters and configurations, resulting in large ensembles of performance profiles. Considerable development time is spent comparing the performance of multiple executions in pursuit of optimal performance.

Profiling tools [1], [2] record the application's *calling context tree (CCT)* (*i.e.,* the prefix tree of all dynamic call paths) and the associated performance metrics (*e.g.,* execution time and memory usage). Visualization of the performance metrics along with the CCT helps gain insights into the execution of an application [3]. However, existing tools either lack support for large ensembles of profiles or only provide primitive functionalities, *e.g.,* juxtaposed comparison [4], making an effective comparison of hundreds of profiles almost infeasible.

Contributions. We present an interactive, visual analytics tool to enable scalable performance analysis on large ensembles of profiles and demonstrate its efficacy using case studies from real and proxy applications. The presented tool, *CallFlow v1.1*, is released open-source under the MIT license.

II. Management of Large Ensembles of CCTs

Sampled profiles, such as those generated by HPCToolkit [1], Caliper [2], and gprof [5], contain a variety of performance data. In general, two kinds of information are recorded: contextual information, (*e.g.,* the call path, the process ID), and performance metrics (*e.g.,* execution time, memory usage). Usually, two time metrics are recorded: *exclusive runtime* (*i.e.,* the time consumed by a function excluding its callees) and *inclusive runtime* (*i.e.,* the recursively accumulated time consumed by a function and all its callees). Hatchet [6], a Python-based profile analysis tool, can be used to convert sampled profiles into *GraphFrames*. A GraphFrame ($\mathbf{G}$) consists of two data structures: a *graph* ($\mathcal{G}$) that represents the CCT or call graph, and a Pandas [7] *DataFrame* ($\mathcal{D}$) that stores the associated performance metrics.

Given a collection of graph frames, we construct a single *ensemble GraphFrame*, $\mathbf{G}^{\mathrm{sg_E}}$, through unification [8].

Unify DataFrames. First, we perform a *unify* operation to concatenate $\mathcal{D}_i$'s into an *ensemble dataframe*, ($\mathcal{D}_\mathrm{E}$). For large ensembles, $\mathcal{D}_\mathrm{E}$ can become prohibitively large, incurring large cost for data operations. To alleviate the computational costs, we abstract the information stored in $\mathcal{D}_\mathrm{E}$ with respect to *name* and *module*. The former represents the data with respect to the call site's names, whereas the *module level* groups the call sites based on the module/library they belong to.

Unify CCTs. Next, *unify* is performed on the $\mathcal{G}_i$'s to construct an *ensemble CCT*, ($\mathcal{G}^{\mathrm{cct_E}}$). The unify operation merges the calling contexts that share the same caller-callee relationship across $\mathcal{G}_i$'s. A call site from $\mathcal{G}_i$ is considered equivalent to a call site from $\mathcal{G}_j$ if they have the same calling context across the two runs. The corresponding performance metrics are aggregated and stored as vectors. The unification enforces graph equivalence by assigning a null value ($\varnothing$) to the missing nodes in any $\mathcal{G}_i$. Next, $\mathcal{G}^{\mathrm{cct_E}}$ is converted into an *ensemble call graph*, ($\mathcal{G}^{\mathrm{cg_E}}$), by merging the call sites with the same function name. The associated vectors are element-wise added to summarize the performance runtime on each call site. Finally, the $\mathcal{G}^{\mathrm{cg_E}}$ is aggregated into a module-level super graph, *ensemble super graph*, $\mathcal{G}^{\mathrm{sg_E}}$, through user-defined filtering and/or grouping operations. $\mathcal{G}^{\mathrm{sg_E}}$ captures all equivalence relationships among call graphs using nodes, and the edges store the subtle differences in the performance and calling structure. The final construct, the *ensemble GraphFrame* ($\mathbf{G}^{\mathrm{sg_E}}$), is then created by combining $\mathcal{D}_\mathrm{E}$ with $\mathcal{G}^{\mathrm{sg_E}}$, and allows exploring the ensemble interactively using our visual encoding.

III. Visual Exploration of Ensemble Super Graphs

The visual interface of CallFlow v1.1 offers several interactive, inter-linked views that together provide a holistic exploration of an ensemble super graph.

Ensemble Super Graph View encodes an ensemble super graphs using Sankey layout [4], where the resource expenditure (*e.g.,* time spent) is visualized along the depth of the super graph. Each supernode (*i.e.,* a node in $\mathcal{G}^{\mathrm{sg_E}}$) is

e-mail: spkesavan@ucdavis.edu

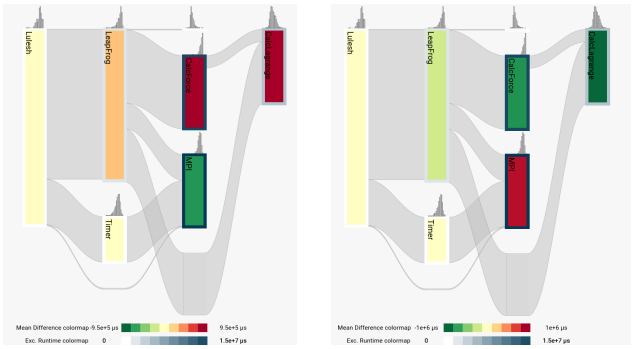

Fig. 1: Diff View: Pairwise differences (subtract operation) of the runtimes between 2 selected runs (i.e., 64-cores vs 27-cores (left), 216-cores vs 125-cores (right)) are colored using a green-red color map.

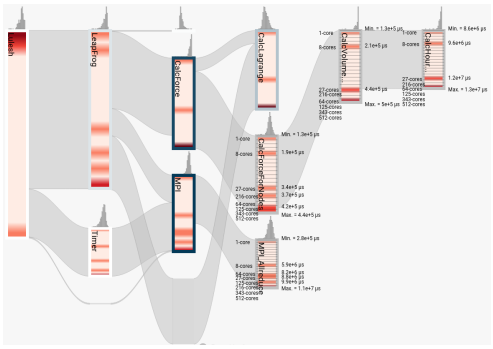

Fig. 2: Ensemble View: Out-of-order runtimes of the libraries, MPI and CalcForce, are highlighted using *text-guides* to study performance variability.

visualized using *ensemble gradients*, which correspond to the distribution of the chosen metric (inclusive or exclusive) mapped vertically along the height of the supernode using a white-red colormap. Using ensemble gradients, users can determine the performance distribution (faster at the top and slower at the bottom of the supernode) as well as identify individual executions, using the *text guides* and *target guides*. To improve flexibility, factors that affect gradient calculation (*e.g.,* number of bins, performance metric) can be modified using the settings. Finally, the color mapped borders directs user's attention towards the resource-intensive call sites.

Diff View enables the user to compare any two executions by visualizing the differences between the mean runtime across supernodes. The resulting super graph is colored with a green-red colormap where hues of red highlight the performance slowdown and hues of green highlight performance speedup.

Supernode Hierarchy lets the user peek into the module-level hierarchy of a supernode. Icicle plots visualizes the call sites of the selected supernode based on their depth using rectangular bars. Ensemble gradients and color mapped borders encode the ensemble distribution and the runtime distribution for individual call site, respectively.

Metric Correlation compares inclusive and exclusive metrics for a selected supernode within the ensemble using a scatterplot. Hovering over a dot highlights the call site by name so the user can compare the runtime metrics across executions.

Runtime Distribution enables the user to assess the distribution of runtime using histograms for a selected supernode with respect to (1) ensemble members, (2) individual call sites, and (3) MPI ranks. Distributions of the ensemble and a target run may be overlaid for comparison.

Call Site Correspondence shows the performance variation for individual call sites using boxplots. The quartiles of the distribution indicate the spread of runtime and the colored dots correspond to the outliers (above and below 1.5× the IQR).

IV. Analysis of Performance Trends in LULESH

We study weak scaling of LULESH [9], a proxy application, across eight execution parameters: 1, 8, 27, 64, 125, 216, 343, and 512 processes. Here, $\mathbf{G}^{\mathrm{sg_E}}$ contains 33 call sites, which we group into six supernodes.

Pairwise comparison of profiles (Fig. 1). When comparing pairs of profiles within an ensemble, highlighting the faster/slower modules can help identify the modules/libraries that exhibit a curious behavior. The *diff view* highlights not only the modules that are slower but also the relative degree of performance degradation. For example, when scaling from 27 to 64 cores per node, *CalcForce* becomes about 5% more slower than *CalcLagrange*. On the other hand, the *MPI* module takes longer when scaled from 125 to 216 cores.

Identification of slow runs (Fig. 2). Toward the ultimate goal of identifying call sites that exhibit inconsistent and/or unexpected runtime behavior, we explore further. From the ensemble gradient patterns across supernodes, we note that the ensemble exhibits good weak scaling. Next, we toggle the text guides to reveal two cases of out-of-order runtimes (with respect to increasing core count) for *MPI* and *CalcForce* libraries and further refine the ensemble view using the interactive *split graph* operation to reveal the corresponding call sites. The text-guides help identify the culprit call site hierarchies, [*MPI*: *MPI_Allreduce*] and [*CalcForce*: *CalcForceForNodes* → *CalcVolumeForceForElems* → *CalcHourGlassControlForElems*].

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-POST-813361). The UC Davis authors are supported in part by the Department of Energy through grant DE-SC0014917. This work was also supported by funding provided by the University of Maryland College Park Foundation.

REFERENCES

[1] 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.

[2] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance Introspection for HPC Software Stacks," in *Proc. of the Int. Conf. for High Perf. Computing, Networking, Storage and Analysis*, 2016, pp. 550–560.

[3] 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.

[4] H. T. P. Nguyen, A. Bhatele, N. Jain, S. Kesavan, H. Bhatia, T. Gamblin, K. Ma, and P. Bremer, "Visualizing Hierarchical Performance Profiles of Parallel Codes using CallFlow," *IEEE Trans. on Vis. and Comp. Graph.*, 2019.

[5] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[6] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proc. of the Int. Conf. for High Perf. Computing, Networking, Storage and Analysis*, 2019.

[7] W. McKinney *et al.*, "Data structures for statistical computing in Python," in *Proc. of 9th Python in Science Conference (SciPy)*, 2010, pp. 51–56.

[8] S. P. Kesavan, H. Bhatia, A. Bhatele, T. Gamblin, P.-T. Bremer, and K.-L. Ma, "Scalable Comparative Visualization of Ensembles of Call Graphs," *arXiv preprint arXiv:2007.01395*, 2020.

[9] I. Karlin, "LULESH Programming Model and Performance Ports Overview," Lawrence Livermore National Laboratory, Livermore, CA (United States), Tech. Rep., 2012.