Visualizing Hierarchical Performance Profiles of Parallel Codes using CALLFLOW

Huu Tan Nguyen, Abhinav Bhatele, Nikhil Jain, Suraj Kesavan, Harsh Bhatia, Todd Gamblin, Kwan-Liu Ma, and Peer-Timo Bremer

Abstract—Calling context trees (CCTs) couple performance metrics with call paths, helping understand the execution and performance of parallel programs. To identify performance bottlenecks, programmers and performance analysts visually explore CCTs to form and validate hypotheses regarding degraded performance. However, due to the complexity of parallel programs, existing visual representations do not scale to applications running on a large number of processors. We present CALLFLOW, an interactive visual analysis tool that provides a high-level overview of CCTs together with semantic refinement operations to progressively explore the CCTs. Using a flow-based metaphor, we visualize a CCT by treating execution time as a resource spent during a call chain, and demonstrate the effectiveness of our design with case studies on large-scale, production simulation codes.

Index Terms—Performance analysis, software visualization, visual analytics, hierarchical data, coordinated and multiple views

1 INTRODUCTION

COMPUTATIONAL science and engineering codes are widely used to gain a better understanding of scientific phenomena. These simulation codes are executed in parallel on large supercomputers with tens of thousands of processors. To achieve faster scientific breakthroughs via high throughput of supercomputers, computational scientists look to optimize the performance of simulation codes by profiling and improving the execution times of different regions in the code.

As a result, domain experts are interested in identifying functions or code regions that are responsible for significant fractions of the overall execution time, e.g., using *gprof* [1], as well as a *calling context* for each function invocation (obtained by walking up the call stack from the function), e.g., using *HPCToolkit* [2]. Combining the calling contexts of different *call sites* (functions) into a single hierarchy, a *calling context tree (CCT)* is obtained. Call sites form the nodes of the CCT, which is rooted typically at the program `main`, where the execution starts, and the path from the root to a particular node provides the node's calling context. A majority of existing tools generate CCTs for the whole application, including the libraries it depends upon.

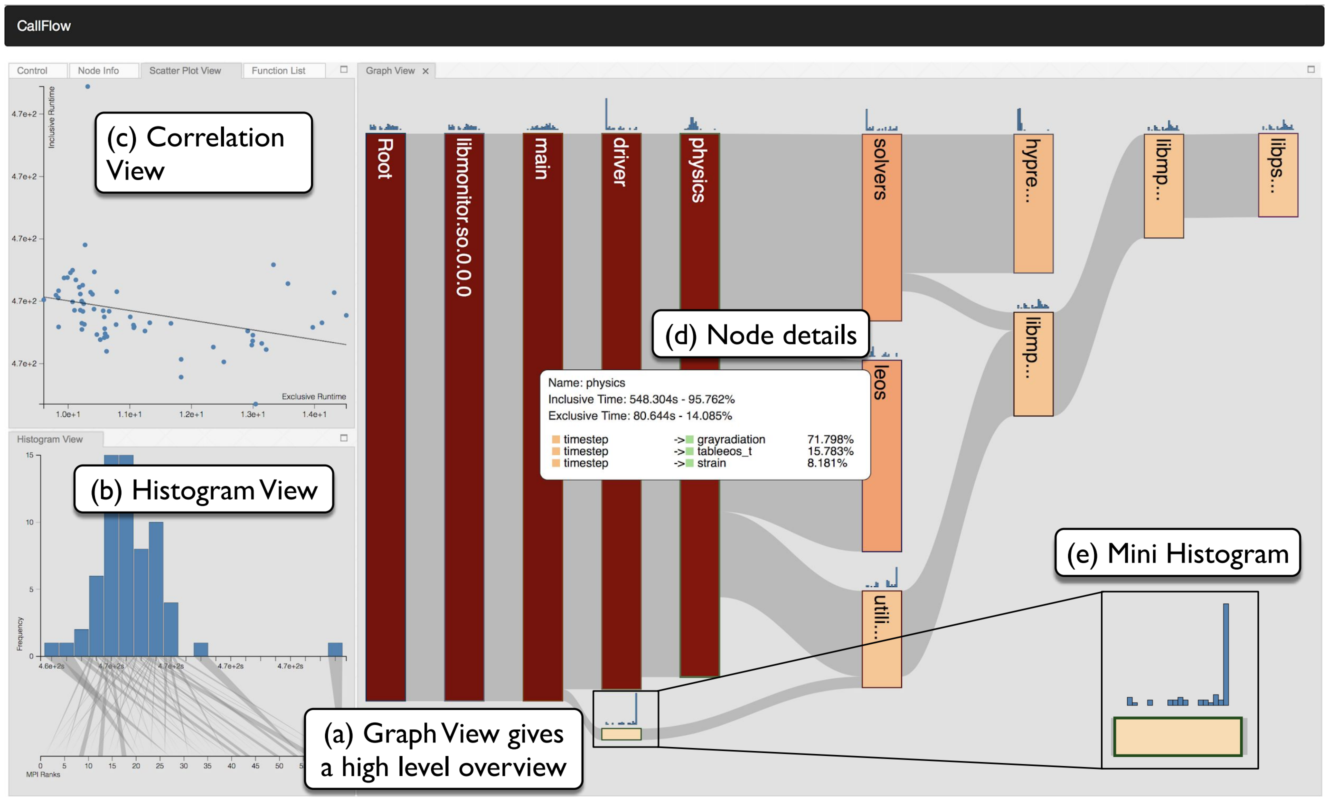
Typically, only a small fraction of the nodes in a CCT is of interest, but such nodes can be buried deep in the tree, and identifying them could be challenging. Furthermore, working directly off a given CCT presents limitations in scalability, and is tightly bound to the hierarchy of the nodes. Instead, it can be more informative to utilize functional affinities between nodes (both among siblings and across levels), which can have a deeper semantic meaning when analyzing CCTs. For example, different call sites that are part of the same code modules, library interfaces, and application function names may be grouped together for a more-effective and easy-to-navigate visualization that still provides desired insights to the user. Transformation of a CCT based on such semantic information leads to the notion of a more generic structure, called a *call graph* [3], [4], [5].

Effective and interactive exploration of call graphs remains a challenge as domain experts seek easy-to-use visualization tools to understand the profiles of large-scale parallel programs. In particular, although many specific and well-defined queries, e.g., extracting hot paths, may be resolved through automated analysis, domain experts often look forward to developing new hypotheses using visual analytics tools combined with human intuition. Most visualization tools currently available operate on CCTs using tree-based metaphors, such as expandable tree layouts used for navigating file systems [2], [6], [7], [8], treemaps [9], or icicle plots [10]. Although familiar to most users, expandable tree layouts do not scale with the size and depth of CCTs, whereas other layouts also use a lot of screen space, under-emphasize leaf nodes, or make comparisons across subtrees difficult. Despite their limitations, domain experts still consider tree-based visualization to be intuitive and well-suited for analysis as it maintains the central notion of hierarchy in the code structure. Nevertheless, the need for an interactive visualization tool that preserves users' intuition, and yet can support a new set of sophisticated queries to explore large-scale CCTs remains a challenge.

Contributions. In this paper, we introduce a new visual analytic tool for interactive exploration of call graphs. Our specific contributions are as follows.

- We present the generic notion of *super graphs*, which can

- *H. T. Nguyen, S. Kesavan, and K.-L. Ma are with the Department of Computer Science, University of California, Davis, CA 95616.*
  E-mail: {htpnguyen, spkesavan}@ucdavis.edu, {ma}@cs.ucdavis.edu.
- *A. Bhatele is with the Department of Computer Science, University of Maryland, College Park, MD 20742.*
  E-mail: bhatele@cs.umd.edu
- *N. Jain is with NVIDIA, Inc.*
  E-mail: nikhijain@nvidia.com
- *H. Bhatia, T. Gamblin, and P.-T. Bremer are with the Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551.*
  Email: {hbhatia, tgamblin, ptbremer}@llnl.gov



Name: physics
Inclusive Time: 548.304s - 95.762%
Exclusive Time: 80.644s - 14.085%

timestep	->	grayradiation	71.798%
timestep	->	tableeos_t	15.783%
timestep	->	strain	8.181%

Fig. 1: CALLFLOW presents dynamically interlinked visualizations to explore calling contexts of large-scale parallel applications. (a) The graph view visualizes the call graph using tailored Sankey diagrams at the desired level of detail. (b) The histogram view enables identifying runtime variations across processes, using histograms and shadow lines, which map histogram bins to process ids. (c) The correlation view allows finding correlation between two attributes of interest. (d) is the tooltip that gives additional information when hovering over a node in the graph view, and (e) gives a closeup of a node with a mini histogram, assisting a quick determination of variability across processes.

be used to represent sampled profiles at user-controllable levels of detail, including but not limited to CCTs and call graphs. We describe new abstractions of the data using semantic *filtering*, *aggregation*, and *splitting* operations.

- We use a flow-based metaphor to visualize super graphs using Sankey diagrams. Instead of using the traditional top-down layout of trees, which emphasizes the levels in the hierarchy, we use execution time as the resource spent to encode the program execution along a given call stack.
- We present the realization of our visual encoding as an open-source[1] visualization tool, CALLFLOW (see Fig. 1). Our tool enables interactive exploration of large-scale CCTs through focus+context visualization by expanding or contracting a super graph where desired.
- We discuss our design process, including data and task abstractions, which are relevant to visualization researchers working in similar domains. Through two case studies using large-scale, parallel, production simulation codes on leadership-class computing machines, we evaluate the design and utility of CALLFLOW, and report on a success story of how visualization research can be leveraged to support crucial inquiries in other fields.

2 PERFORMANCE PROFILES

Understanding performance profiles of large-scale, parallel codes is essential to maximize the output of software-hardware investments. Although we can instrument and gather a variety of performance data on parallel machines, this paper focuses on sampled profiles.

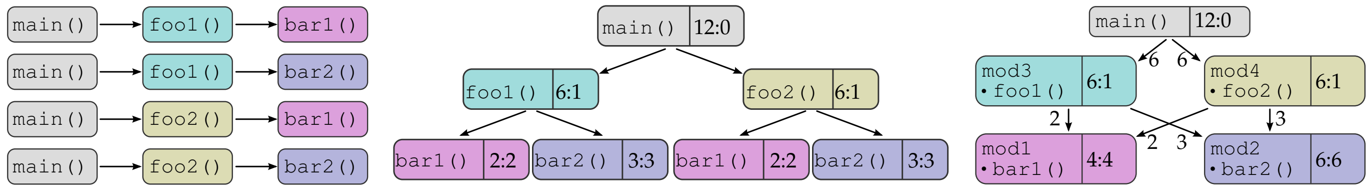
Sampled profiles are collected by forcing an interrupt in the program every nth instruction. At each interrupt, a sample is collected, which contains two types of information: *contextual information*, i.e., the current line of code, file name, the call path, the process ID, etc.; and *performance metrics*, such as the number of floating point operations or branch misses occurred since the last sample. Statistically, the number of samples that fall within a given function represents a good estimate of the time spent in the function. Sampled profiles have been employed widely for performance analysis as they produce reliable data with small overhead, which depends only upon the sampling frequency, and not on the complexity of the call path. Table 1

```
// example code
int bar1(){/*...*/}
int bar2(){/*...*/}
int foo1()
{ bar1(); bar2(); }
int foo2()
{ bar1(); bar2(); }
int main()
{ foo1(); foo2(); }
```

name	inclusive time	exclusive time	% time
bar1	4	4	33.34
bar2	6	6	50.00
foo1	6	1	8.33
foo2	6	1	8.33
main	12	0	0.00

TABLE 1: An example program with its flat profile. The profile contains the *inclusive* (cumulative) time, the *exclusive* (self) time, and the percentage of time a function uses.

1. Released under MIT license. https://github.com/LLNL/Callflow



(a) Call paths (b) Call tree (c) Call graph

Fig. 2: The call paths (a), call tree (b), and call graph (c) of the example program in Table 1. In (a), each call path shows invocation instances of `bar1()` and `bar2()`, rooted at `main()`. Each node in the corresponding call tree contains three pieces of information: node name, inclusive metric, and exclusive metric (denoted as inclusive : exclusive). Since the call graph is constructed from the call tree, the original inclusive cost information can be retained through edge weights.

exemplifies a simple sampled profile. The collected samples can be aggregated in different ways to simplify the analysis.

Calling Context Trees. When the samples are aggregated by unique call paths, it results in a *calling context tree (CCT)* [11], [12]. Each unique invocation of a function (by call path) becomes a node in the CCT, and the path from a given node to the root of the tree represents a distinct *calling context*. Metrics on each node can be *inclusive* or *exclusive*—the former represents the metric values that can be attributed to the body of a given function (including any function calls in the body), and the latter represents the difference between the inclusive value and the values that can be attributed to its child nodes. Fig. 2(a) shows the call paths of the program in Table 1, and Fig. 2(b) shows the corresponding CCT.

However, when using performance tools such as HPCToolkit [2], a node is not limited to function invocations only but can also represent loops, statements, and other code structures. Moreover, for parallel programs, one can aggregate samples across all threads or processes to create a global CCT or keep a forest of CCTs in which each node carries separate information for each thread/process. Therefore, despite being very informative, CCTs pose practical analysis challenges. Modern applications are typically built on top of rich frameworks and libraries that provide many layers of abstraction, increasing the depth of call paths, leading to very large CCTs. Such large-scale trees are often hard to decipher, potentially leading to oversight. Although one may want to reduce the size and/or depth of the CCT by discarding the least important nodes, e.g., functions with negligible timings, a simple filtering of such nodes could change the topology of the CCT, and standard approaches for analysis may not be applied.

Call Graphs. Calling context trees can be aggregated in different ways to provide information in a more concise manner. *Call graphs* [1] are created by merging CCT nodes with the same name (function name), e.g., see Fig. 2(c).

By aggregating CCTs across function names, call graphs can significantly reduce the scale and complexity of the data. Nevertheless, despite the simplification, call graphs of large, parallel programs can retain additional complexity that often prohibits the experts from unraveling the underlying profiles. In favor of easy visual exploration, we introduce the generic notion of *super graphs*, which allow aggregating raw profiles at appropriate and controllable levels of abstraction. Super graphs will be introduced in detail in Section 6.

3 Related Work

Most analysis tools visualize CCTs as expandable trees [2], [6], [7], [8], using which the user can show and hide nodes as well as sort by attributes (see e.g., Fig. 3). Although useful in some cases, such visualizations do not provide a clear understanding of the code structure, and suffer from scalability issues.

Node-link layouts [14], [15], [16], [17], [18], [19], [20], [21] are a popular approach for tree visualization, although dense matrix-based representations perform better for large-scale trees [22], [23]. Node-link layouts represent entities as nodes, and relationships as edges. In the case of a CCT, the entities are the frames in the call stack and edges represent the caller-callee relationship. Various types of information can be shown on the node, e.g., time can be encoded as the color of the node. Several techniques have been proposed to extend node-link layouts by encoding additional information. For example, DeRose et al. [24] embedded a histogram onto the node to show imbalances between processes, and Nguyen et al. [25] encode runtime variation among processes to indicate the anomalies. Bohnet and Döllner [26] identify and visualize features in the data. For large-scale parallel applications with hundreds to thousands of function calls, visualization of all nodes using node-link layouts becomes intractable. More recently, Xie et al. [27] employ a node-link layout to represent the learned structural features of the CCT computed using an anomaly behavior detection model. Burch et al. [28] use timeline- and pixel-based aggregations to visualize dynamic graphs.

Many space-filling visualization approaches have been used to visualize large-scale hierarchical data. Treemaps [29], [30] have been effectively used to visualize hierarchical data by partitioning the screen space into bounding boxes that represent the tree structure.

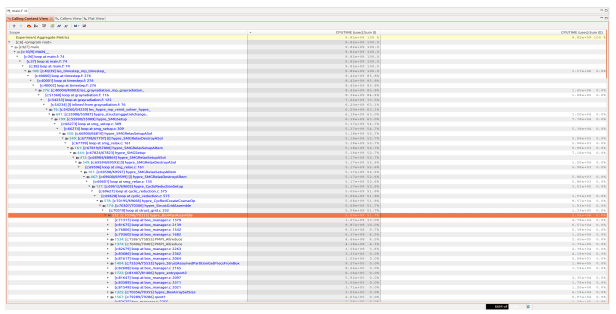


Fig. 3: The CCT of Miranda [13] visualized as an expandable tree using HPCToolkit [2]. Each node of the tree is shown, but the overall structure of the application remains hidden.

However, treemaps under-emphasize leaf nodes and make comparisons between subtrees difficult. Radio plots [31], where nodes are arcs stacked radially outward along the depth of the tree, are also a candidate for tree visualization, but also suffer from scalability issues. Since aggregation of nodes of a CCT into a call graph (or more generally, a *super graph*) can introduce nodes having multiple call paths or nodes with multiple parents, hierarchical space-filling visualization layouts are not well suited.

On the other hand, although node-link layouts are effective to present connectivity, even for complex graphs [32], [33], [34], the edges of standard node-link layouts usually represent only connectivity. Since our domain problem requires us to also encode the flow of resources, most notably time spent, we employ a modified node-link layout. A *flow diagram* uses a flow-based metaphor that represents how energy is transferred from one entity is transferred to another. It is commonly used in engineering and science in the form of a *Sankey diagram* [35], [36], [37]. A Sankey diagram uses a weighted, directed graph, where the width of each link represents the amount of energy entering and leaving an entity in the system. Sankey diagrams are not limited to visualizing energy flow; other works have extended the diagram to show the flow of time. Ogawa et al. [38] represent the number of people participating in the mailing list of open-source software projects each month using a Sankey diagram. Wongsuphasawat et al. [39] and Wang et al. [40] aggregate similar temporal events of patients' diseases and symptoms. In this work, we use Sankey diagrams to visualize performance profiles by representing time spent along the execution path.

4 Design Methodology

Widely anticipated challenges in the design process for domain-specific interactive visualization tools include the gaps between the understanding and expectations of visualization scientists and domain experts and insufficient evaluation strategies [41], [42], [43]. The visualization community has developed formal guidelines [43], [44], [45], [46] for an effective design process. A common thread among such methodologies is to have a verifiable approach that translates domain knowledge and vocabulary into visualization terminology, and in consultation with the domain experts, evaluate various visualization choices with respect to their suitability to the application at hand [47], [48]. For example, Sedlmair et al. [46] present a nine-stage framework encompassing the analysis of some specific real-world problem faced by domain experts, design of a visualization system to support a solution, validation of the design, and reflection about the lessons learned. In this work, we chose the four-phase nested model proposed by Munzner [45] for the design of CallFlow, as this chosen model provides a clear balance between flexibility and specificity of the process. This paper describes the first three phases of our methodology (Sections 5, 6, and 7). Due to space limitations, we cannot provide the algorithmic and implementation details of the tool.

"Information visualization is usually part of some creative activity that requires users to make hypotheses, look for patterns and exceptions, and then refine their hypothesis." [47] This is also the case with the domain experts collaborating with us, which include software developers as well as performance analysts who assist computational scientists optimize their codes. Through an interactive collaboration with domain experts, including interviews and discussions over a period of several months, we identified the primary challenges faced by them in the exploration of performance profiles of large-scale parallel codes. CallFlow was developed in an iterative manner, with our collaborators having access to the evolving prototypes, allowing us to refine the CallFlow to best assist the experts in their inquiries.

An increasing concern among different visualization techniques introduced for software visualization is the lack of concrete evaluation [49]. To this end, several design decisions were made in consultation with the experts, to resolve the tradeoff between the simplicity of the tool and the types of queries supported. Section 6 also provides validation of some of our design choices. Finally, we describe two case studies on real data gathered from scientific applications running on leadership-class supercomputers, and evaluate CallFlow's effectiveness in helping the user explore parallel application codes.

5 Domain Problem Characterization

The first phase of our design study involved developing knowledge about the domain problem. Over a period of about one year, we conducted several interviews with various HPC experts at LLNL, who are interested in improving the performance of large-scale parallel applications. Our focus was to clearly understand their goals, as well as the current workflow and the limitations therein.

A CCT can contain a host of different information, and has been used for several automated analysis techniques, e.g., extracting hot-paths [2], [50]. However, such automated approaches usually address only a well-defined aspect of a more-general goal that domain experts are interested in: *"finding performance bottlenecks"*. In practice, users often face less well-defined problems, e.g., an application underperforming on a particular input or a new platform, without a clear indication of the root cause of the problem. Through numerous discussions with experts, it became clear that although an automated tool to pinpoint problems would be ideal, past experience has shown that the underlying causes are so case-specific that human intuition and expertise are often key to making progress. Consequently, the overarching goal when designing CallFlow has been to provide a generic way of exploring CCT data to either diagnose the problems directly or identify which existing tools may lead to new insights.

High-level overview of calling contexts. The CCTs constructed directly from sampled profiles contain details up to individual function calls, and therefore, can create tens to hundreds of thousands of nodes. For easy navigation and understanding of data, experts expressed interest in a high-level overview of CCTs with filtered and/or aggregated information. CallFlow develops the notion of super graphs to allow visualization of aggregated calling contexts based on user-defined semantics.

Metrics-based visual profiling. Two types of performance metrics are critical for performance analysis: *inclusive* vs. *exclusive* (see Section 2). Together, these metrics signify the performance of different parts of the code and can offer significant insights into bottlenecks and help address them. For example, if the inclusive time of a given function significantly outweighs its exclusive time, then experts explore the performance of its callees, whereas attention is paid to the function itself if its exclusive time is significant. One of the goals for visual exploration of CCTs is to be able to denote both inclusive and exclusive performance metrics.

Process-based visual profiling. When large-scale applications are deployed on supercomputers, making effective use of the available resources is critical. Although increasing the number of processors typically yields better performance, maximizing the performance requires balancing the load on different processes. A visual representation of the time spent by individual processes in a CCT node can help understand the balance of load [24], as well as in detecting parts of the codes that have high exclusive costs distributed in an inconsistent fashion.

Additionally, the experts expressed interest in finding out whether code slow down is related to IDs of specific processes. For instance, the computation of the physical domain in a simulation, e.g., a volume, is distributed among the processes according to a certain regular pattern, e.g., a row-major order. Knowing which MPI processes are slow vs. fast and identifying any patterns, e.g., every nth process being slow, allows experts to form new hypotheses on potential root causes. Note that the experts expect such patterns hard to generalize, as they could be domain- and data-dependent, that can reorder the processes arbitrarily and thus completely change the observed pattern.

User-driven interactive visual analytics. Given the different types of analysis tasks that experts are interested in, a severe limitation in their current workflow is the lack of a comprehensive tool that allows the desired functionality in an interactive manner. For example, HPCToolkit [2] provides two separate views for top-down (calling context) and bottom-up (callee's context) traversals of a given CCT, each supporting a different type of inquiry. However, switching between views causes additional cognitive load, leading to an analysis that is ineffective at best and incorrect in extreme cases. To easen and accelerate the exploration process, experts expressed interest in an unified visualization with an ability to resolve different types of queries while maintaining the user's focus.

Finally, the experts are also interested in supporting side-by-side comparative analysis to analyze how calling contexts vary at the process level. For such comparative analysis, the goal is to understand two types of differences: 1) comparison of CCTs' contexts to understand hierarchical differences in the caller-callee relationship, and 2) per-node comparison to analyze the differences in execution metrics.

The aforementioned limitations in existing workflows create several gaps in experts' understanding of the performance of large-scale applications, leading to suboptimal use of computing resources. CALLFLOW is designed to fill such gaps by supporting a versatile set of interactive inquires on CCTs. Equipped with this tool, the domain experts can not only explore the sophisticated causes of performance bottlenecks more effectively, but also devise new strategies to overcome them (see Section 8).

6 DATA TYPE ABSTRACTION AND OPERATIONS

Although different profilers can have slightly varied data formats, generally, the input data to CALLFLOW contains two types of information: (1) the hierarchy of function calls in the profile, and (2) the performance metrics associated with the functions therein. Irrespective of the source, the former type of data can be converted into a a CCT or a call graph, with its root at the first call of the application, usually, the function `main` (see e.g., Fig. 2).

Super graphs. As argued earlier, the scale and complexity of CCTs or call graphs poses significant challenges for interactive visual exploration. In this work, we present the generic notion of a *super graph*, which are created by merging nodes of CCTs. Super graphs utilize semantic information to provide a high-level overview of the code. For example, several nodes in a call path often belong to the same library and might have repetitive calls from different code modules to form similar subtrees with different parent nodes. In such cases, visualizing nodes that correspond to modules or libraries are usually more meaningful than function-level nodes. Therefore, grouping function calls by modules provides a semantically meaningful representation of the underlying CCT. Although the notion of semantic representations for call graphs is not new [27], super graphs are introduced as a more-general concept to express CCTs[2] (merging no nodes), call graphs (merging by call paths), module diagrams (merging by load module), and anything in between (e.g., see Fig. 4).

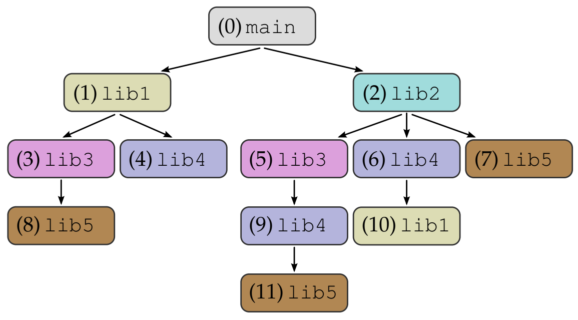
Formally, we denote a super graph as $\mathcal{G}_{\text{cct}}(\mathcal{V}, \mathcal{E})$, where the set of nodes, $\mathcal{V} = \{\mathbf{v}_i\}$, uniquely represent the call sites (functions in the call stack), and the directed edges, $\mathcal{E} = \{\mathbf{e}_{ij}\}$, capture the caller-callee relationship between $\mathbf{v}_i$ and $\mathbf{v}_j$, respectively. Each edge $\mathbf{e}_{ij}$ is associated with a weight w_{ij}, which depends upon the performance metrics of the two nodes (see e.g., Fig. 2(c)). The performance metrics for nodes are stored as rows in pandas [51] dataframes, which allow fast access and operations. In addition to inclusive and exclusive metrics, cache misses, etc., the dataframe also stores meta-attributes of the nodes, such as function name, file name, and location in source code.

Given the scope and requirements for CALLFLOW, the domain-specific goals can be translated into more-specific graph operations: *filtering*, *aggregating*, and *splitting* of nodes.

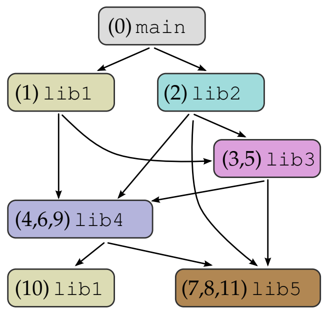
6.1 Filtering of CCT Nodes

The first operation when processing any CCT is typically filtering out nodes unlikely to be of interest to the user. In particular, the nodes towards the bottom of $\mathcal{G}_{\text{cct}}$ typically represent decreasingly smaller portions of the overall run time. Since the goal of CALLFLOW is performance optimization, functions that represent only a tiny fraction of the overall time are not of much interest to the user. Also, each function could potentially be represented thousands

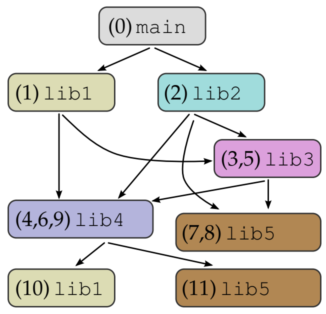
2. Although a CCT is a tree, notating it as a (super) graph allows discussing the various operations of interest more concisely.



(a) A given CCT



(b) Node aggregation



(c) Split by entry function

Fig. 4: Node aggregation and splitting operations. (a) shows the original tree, labeled as "[function_name] `module_name`"; (b) shows aggregation operation where nodes from the same module are merged together. The aggregated super graph contains two *supernodes* corresponding to `lib1` to prevent a cycle; (c) shows a split by entry function operation in which `lib5` supernode is split with respect to its entry functions, (7,8) vs. (11).

of times in the CCT, being called from different contexts. Therefore, filtering $\mathcal{G}_{\text{cct}}$ by removing nodes with small inclusive runtimes could remove a nontrivial portion of the execution. Instead, filtering the nodes based on the total inclusive runtimes across all the instances of the corresponding function is more meaningful. The aggregate information of the removed nodes still remains available as part of the inclusive runtime of their ancestors. The only information that is lost is the ability to differentiate how this filtered runtime is distributed among lower level calls.

From our experiments, we noticed that even a conservative threshold (less than 0.1% of the root's inclusive run time) can reduce the number of nodes in $\mathcal{G}_{\text{cct}}$ drastically (by approximately 70–80%). A majority of nodes are filtered out because most function calls are wrapper functions that are called by a library in the program and do not contribute to effect in performance. Therefore, filtering is key to enable an interactive tool. The output of the filtering is a smaller super graph, $\mathcal{G}_{\text{filt}}$. Filtering removes information from the CCT, and thus, in principle, could impact the downstream analysis. To mitigate the information loss, CALLFLOW supports repopulation of the filtered nodes, if desired. Combined with this fail-safe operation, filtering proves to be a powerful tool for exploration of large CCTs.

6.2 Aggregation of CCT Nodes

Modern software abstractions have numerous intermediate call sites that are not relevant to performance analysis. Such call sites can obscure relevant information by spuriously increasing the height of the tree. For example, common accessor functions in object-oriented languages or template wrappers create additional call sites with insignificant performance metrics. In most cases, these nodes are internal to the tree therefore, cannot be removed without removing the corresponding subtree. Instead, these nodes should be aggregated with respect to higher-level code abstractions, such as libraries, code modules, files, etc., which are often more intuitive to the user.

In particular, every node in the CCT belongs to a higher-level abstraction, which can be represented as a *hierarchy map*, μ. Merging the nodes of $\mathcal{G}_{\text{cct}}$ (or $\mathcal{G}_{\text{filt}}$) recursively based on μ until a desired level of abstraction is obtained creates the super graph, $\mathcal{G}_{\text{cfg}}(\mathcal{V}^{\text{s},\mu}, \mathcal{E}^{\text{s},\mu})$. Here, the *supernodes*, $\mathcal{V}^{\text{s},\mu} = \{\mathbf{v}_i^{\text{s}}\}$ are aggregates of the nodes of $\mathcal{G}_{\text{cct}}$ (or $\mathcal{G}_{\text{filt}}$) with respect to μ, and the *superedges* $\mathcal{E}^{\text{s},\mu}$ connect the supernodes. For example, given a hierarchy of a function call, module > library > filename > function, after merging, a module could become a supernode with the remaining hierarchy stored as its subgraph.

To describe the performance metrics for supernodes, we first discuss another crucial data component. The *entry functions* are the functions through which the control enters to a particular module or library. Specifically, given a supernode, $\mathbf{v}^{\text{s}} = \{\mathbf{v}_i\}$, its entry functions, $\mathbf{v}^{\text{s,e}}$ are defined as the nodes whose parents do not belong to the same supernode, i.e., $\mathbf{v}^{\text{s,e}} = \{\mathbf{v}_j\}$ such that $\{\mathbf{v}_j\} \in \mathbf{v}^{\text{s}}$ and $\{\text{parent}(\mathbf{v}_j)\} \notin \mathbf{v}^{\text{s}}$. The exclusive metrics of a supernode, $\mathbf{v}^{\text{s}}$, is the sum of those of all its components nodes $\{\mathbf{v}_i\}$. However, the inclusive metrics of only the entry functions, $\{\mathbf{v}^{\text{s,e}}\}$ are used to represent that of $\mathbf{v}^{\text{s}}$.

We note that although aggregation removes valuable information, such as call paths, from the visualization, the domain experts value the ability to quickly detect bottleneck functions over finding the 1-to-1 caller-callee relationship.

Aggregation of nodes could introduce cycles in the super graph when a function belonging to a library is called multiple times along a call stack. However, cycles break the common understanding of control flow, and are typically considered artifacts of specific implementation patterns, most notably callback functions. Furthermore, cycles would significantly complicate the visualization. Consequently, we prevent cycles from forming during the aggregation, and instead preserve multiple nodes from the same namespace level. For example, when merging the nodes by the libraries they belong to (see Fig. 4(b)), the supernodes `lib1` and `lib2` create a cycle because they call functions in each other. For such cases, duplicate supernodes can be created for some of the labels, e.g., `lib1`. Independent of the hierarchy based on which the nodes are merged, the metrics for the supernodes can be aggregated, and the edges preserved. The resulting super graph would represent the control flow at the selected level of detail with nodes indicating concepts like modules and edges indicating the calling hierarchy.

Aggregation of nodes can be done easily and interactively using standard data structures. The one potential pitfall of these operations is that preventing cycles may result in multiple merged nodes with the same name label, i.e., from the same module/library, which could be counter-intuitive to the user. However, experts anticipate that most such cases arise due to the callback architecture used for performance introspection, as CCTs are typically recorded using default callback interfaces. In a callback pattern, one of the two nodes is associated with setting the callback, and typically does not contribute meaningful runtime. As a result, we employ a layout that does not support cycles, since their downsides outweigh the benefits.

6.3 Splitting of CCT Nodes

The super graph at a given level of refinement may be too coarse to diagnose many performance problems. Therefore, the users are interested in resolving additional details upon request. To this end, CALLFLOW supports splitting of a chosen supernode into two or more (super)nodes, and redistribute the original flow. Although there could be several strategies for splitting nodes, each guided by its own application-dependent use case, through several discussions with domain experts, two most relevant approaches to the semantics of the analysis were identified.

Split by entry functions is the operation that splits a supernode into "component" (super)nodes based on what entry functions they are called by. As discussed earlier, entry functions are generally the public API functions of a module, or a library which the developers are familiar with. To allow users to know the API calls that consume high resources, CALLFLOW allows the user to select one or multiple entry functions belonging to a supernode, and split it into component (super)nodes such that each component (super)node corresponds to a single entry function. Fig. 4(c) shows the splitting of node `lib5` into two, based on entry functions (7,8) and (11), respectively.

Split by callees. Often, lower-level libraries, e.g., MPI, which are typically called by multiple higher-level modules, consume more time than expected. In such cases, the logical next step is to determine whether the problem exists in all contexts, i.e., in all parent modules, or only in some of them. To support such queries, CALLFLOW allows splitting a (super)node with respect to its parents. This operation allows the user to determine where the costs for a particular (super)node come from and where the cost will propagate to. Additionally, it informs the user about the functions or modules responsible for high exclusive time, if any.

By determining entry functions as part of the aggregation step, both splitting operations are easy to support. Both involve only local changes in the topology of $\mathcal{G}_{\text{cfg}}$ and local updates to the metrics. Other splitting operations could be implemented, e.g., to isolate specific nodes of $\mathcal{G}_{\text{cfg}}$ or to recursively split apart two subtrees. However, the former would require first determining which node to isolate, implying that the source of the problem is known, and visual exploration not needed. The latter is an example of automating certain interactions and, in practice, we have not encountered common enough patterns to justify the additional complexity. Another potential candidate is a split by children; however, such a split may not be possible in most cases, as a single node could call into multiple different libraries, and thus may not be able to split accordingly. In summary, the experts consider the chosen splitting operations sufficiently flexible to support the detail-on-demand exploration of interest.

7 VISUAL DESIGN OF CALLFLOW

CALLFLOW is an interactive tool with three linked views: control flow view, histogram view, and correlation view, (see Fig. 1 for an overview). Together, the three views support the queries of domain experts in an interactive manner.

7.1 Control Flow View

The control flow view presents an overview of the application's control during execution. We visualize $\mathcal{G}_{\text{cfg}}$ (or $\mathcal{G}_{\text{cct}}$ or $\mathcal{G}_{\text{filt}}$) using a flow-based metaphor with Sankey diagrams, where a directed graph is laid out with respect to the amount of resource under consideration. We treat the inclusive metric (usually, the execution time) as the resource being distributed among supernodes. To effectively use the aspect ratio of common visual mediums (e.g., computer monitors), we use a horizontal Sankey layout, where the direction of the graph goes from left to right. Thus, each supernode is encoded as a rectangular bar with its height representing the sum of the inclusive metrics of all its entry functions. A superedge $\{\mathbf{v}_i^{\text{s}}, \mathbf{v}_j^{\text{s}}\}$, represents the flow of inclusive metrics between the two nodes, and the thickness (in vertical direction) of the superedge indicates the inclusive metric consumed by the target node, $\mathbf{v}_j^{\text{s}}$.

By design, this visual encoding captures the direction of the super graph, i.e., the control flow can be traced easily from the root node (left) to leaf nodes (right). Furthermore, our visual encoding not only highlights inclusive metrics directly, but also indicates exclusive metrics easily. In particular, the exclusive metric for a given supernode is the difference in the thickness of incoming and outgoing superedges. The exclusive metric is indicated by empty portions towards the bottom of supernodes, where no outgoing edges exist, e.g., the `physics` module in Fig. 1. However, such differences may be difficult to notice visually when exclusive times for nodes are small. To alleviate this limitation, CALLFLOW can also use color to encode exclusive metrics. Finally, we use constant widths (horizontal spans) for all supernodes to easily compare of the area of the nodes and identify nodes with high costs. Thus, the user can identify nodes with high inclusive and exclusive cost as bars with a large area and dark color.

Although the visual encoding described above captures the control flow of an application, effective exploration still requires tailoring the Sankey layout of the super graph at hand, especially considering the interactive support of splitting and aggregation operation for large-scale super graphs. CALLFLOW's graph layout is based on a key domain-specific insight that neither the depth of a supernode nor the order of sibling supernodes has any inherent significance. Consequently, we can vary the placement of supernodes to make the graph as readable as possible, with the only constraint being that the left-to-right order must preserve the call stack order. The key criterion to optimize when choosing a suitable layout is to minimize the number of edge crossings, because edge crossings create visual clutter and can obscure information.

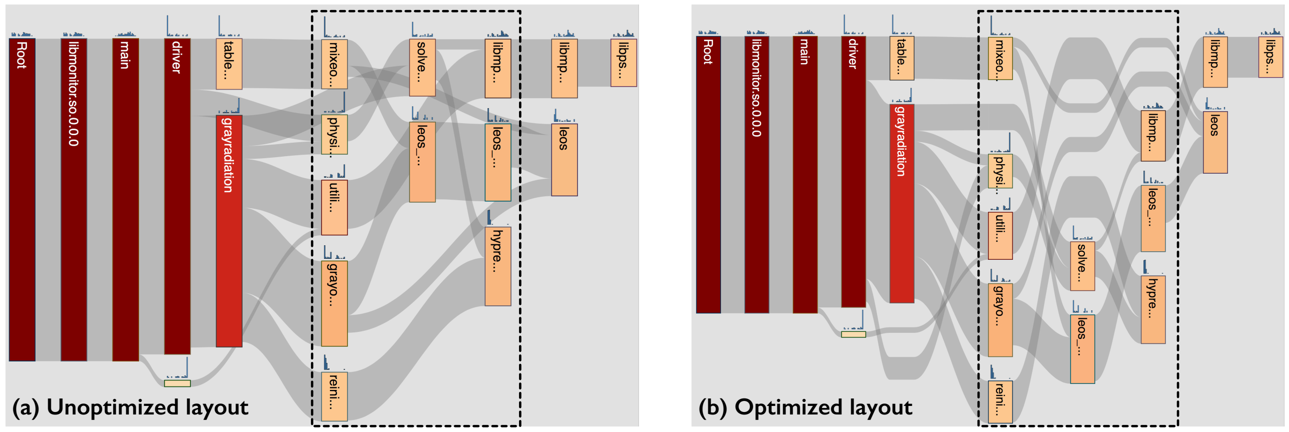


Fig. 5: CALLFLOW uses a modified version of layout optimization presented by Alemasoom et al. [37]. With many overlapping edges in the unoptimized layout, (a) the connectivity is harder to decipher. Using the optimized layout (b), e.g., edges connecting `mixeo` and `leos`, and `util` and `libmpi` can be seen more clearly as compared to (a).

Horizontal positioning. Sankey visualizations place nodes in "layers"; the spacing between layers is usually consistent, allowing for an even distribution of horizontal space. Since a CCT is a hierarchical tree, the derived super graphs generally do not contain many nodes in the initial layers, whereas there is also more overlap in the later layers due to the increase in the number of nodes and edges. Therefore, even horizontal spacing leads to ineffective use of space. Since a supernode typically appears in multiple calling contexts, we define its *level* as the maximum depth among all the contexts (paths in the super graph leading back up to the root), and use supernodes at the same levels to create "layers" in the Sankey layout. Once levels have been computed for all supernodes in the super graph, the horizontal position for lth layer is computed as $x_l = \max(\min_x, l^k \cdot \max(n_l, n_{l-1}))$, where, $\min_x$ is the minimum space between adjacent levels, n_l denotes the number of supernodes in level l, and k is a scaling exponent. This approach places the layers with fewer bars closer to each other than the layers with larger node count.

Vertical positioning. To assign vertical position to nodes, we follow the method described by Alemasoom et al. [37]. Similar to their technique, we add dummy nodes and edges to connect two nodes when they are in nonconsecutive levels. These intermediate nodes simplify the layout by providing anchors to long edges, and thus, reduce edge crossings. The height of the dummy node is equal to the flow between the original nodes it connects. The approach computes an optimized layout that minimizes the weighted sum of distances between each two connected nodes in consecutive layers. We impose an additional constraint to this optimization by imposing a minimum vertical gap between two nodes within the same level to allow embedding the histogram for process-specific information. Fig. 5 demonstrates the value of such an optimization.

7.2 Histogram View

To enable process-based visual profiling, CALLFLOW provides statistical visualization using histograms. Although common approaches often display measures such as standard deviation, quartiles, etc., they are useful mostly when the data comes from a known distribution. Since there is no reason to assume that run times of parallel applications would follow a specific distribution, such measures can be misleading. Instead, CALLFLOW uses histograms to show the actual distributions. The histogram view in CALLFLOW shows the sampled distribution of process-based metrics for a selected supernode. However, selecting a supernode to highlight its histogram is tedious, particularly in the exploration phase since it forces the user to select several nodes before identifying the one with interesting variation. CALLFLOW addresses this problem by also showing a *mini histogram* at the top of every bar (supernode) in the control flow view (see Fig. 1). The mini histogram is small enough that it can be placed on every bar without creating much visual clutter, yet big enough that the user can quickly identify which supernode has an interesting distribution. Once an interesting distribution has been identified, the user can select the corresponding supernode to view the larger version of the histogram.

To assist the user explore the connection between slowdowns in MPI ranks and the physical domain, we display the rank-to-bin mapping in the histogram view. There are two ways in which the histogram view indicates this mapping. First, hovering over a bin in the histogram pops up a tooltip informing the user about the ranks in the corresponding bin. The second approach is *shadow lines*, which map the bins in the histogram to the process/rank id laid out on an ordered line at the bottom of the histogram. Fig. 1(b) shows the shadow lines within the histogram view. Although shadow lines can create visual clutter, especially for large processor counts, this is in fact a desired visualization since it indicates that the code behaves normally. Clutter generally appears when bins in the histogram contain a broad range of rank, an indication that the rank id is not correlated to the observed run times. On the other hand, scenarios without clutter indicate that certain run times are correlated to the rank id, which can be a sign of load imbalances and inefficient algorithms.

7.3 Correlation View

Generally, performance bottlenecks can be observed in many metrics. For example, high cache misses lead to higher run times, as more time is spent accessing the memory.

Analyzing many metrics individually can be cumbersome; instead, CALLFLOW leverages the correlation between two metrics to identify performance bottlenecks among different processes using a correlation view (see Fig. 1), where each point in the scatter plot represents a process. If there is a correlation between two metrics, we expect processing elements to form clusters, whose size informs the extent of correlation. We also show the best-fit line to aid the user in observing the trend of the scatter. Hovering over the best-fit line displays useful statistical measures. The correlation aids in choosing the bins in the histogram view that cause load imbalances among processes based on their MPI ranks, and later compare their respective subgraphs.

7.4 User Interactions

The user can interact with CALLFLOW in several ways.

Hovering over a supernode brings a tooltip with information about the corresponding supernode. As shown in Fig. 1(d), the tooltip shows the name of the corresponding module/function, its inclusive and exclusive metrics. Additionally, the calling context of the supernode is shown: the function that calls the highlighted supernode, the entry functions called in the supernode, and the metrics spent in those calls. For each calling function, a small square is shown indicating which node the function belongs to. The tooltip is useful for a quick inquiry into the functions in a module (supernode) that consume most resources, as well as the functions that call those expensive functions.

Selection of a supernode indicates the user's interest in finding more information about the corresponding nodes. All entry functions of the supernode are enumerated, and the histogram and the correlation views are updated to correspond to the selected supernode.

Zooming and panning are key to navigating the super graph smoothly, especially, when it is large. Although zooming out may reduce the size of the rendered node, where possible, the legibility of node labels is maintained by increasing the font size.

Splitting of nodes is an important target operation for CALLFLOW. A split by entry function requires the user to choose a function from the list of entry functions. On the other hand, a split by parent leads to updating the super graph by replacing the selected supernode with its parents. In either case, the new super graph requires recomputation of layout, which although is done in real time, could impose additional cognitive burden on the user.

To mitigate this burden, we use a consistent naming scheme for supernodes to provide a consistent context in the transition. In particular, we concatenate the names of the (original) supernode and the (new) split supernodes separated by a hyphen. Fig. 6 shows an example of a splitting interaction based on two of its entry functions.

CALLFLOW also animates the transitions to make them easy to follow. Before supernodes transition to new locations, the edges are removed to prevent the user from tracking too many elements at once, thus reducing cognitive stress on the user. Furthermore, the user is more interested in how the nodes are split and by how much. Hiding the edges allows the user to concentrate on the nodes. The nodes are then moved to new locations and new ones are added in

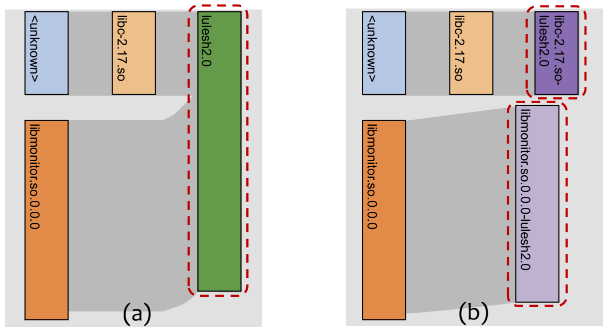


Fig. 6: A node splitting operation is applied on the (a) the green node to create (b) two purple nodes. The split is based on the parents of the original node, and the incoming edges of the new nodes are calculated based on the information from their parent nodes.

the process. The layout minimizes node movements so that unchanged nodes remain as static as possible. After nodes are in the new locations, edges are added back in.

Comparing subgraphs within a super graph is often needed, e.g., to detect load imbalances, where certain processes could remain idle during execution. The user can perform a brushing action on the histogram, and the graph view is split into two, showing the two super graphs (see Fig. 7) associated with the two process groups. The brushed bins constitute the processes that make up the top super graph and the non-brushed bins make up the processes of the bottom super graph. Furthermore, CALLFLOW allows users to color the node based on the difference to detect variations in their exclusive metrics.

8 CASE STUDIES

We present two case studies on understanding profiles of large-scale scientific applications to show the impact of CALLFLOW at Lawrence Livermore National Laboratory. Both these studies were led by our collaborators, who are computational scientists, and work closely with application developers on performance analysis and scaling optimization of scientific codes. These experts have extensive experience in performance optimization of large-scale parallel applications, and have worked with other visualization tools for performance analysis, such as HPCToolkit [2], Scalasca [6], and Vampir [52]. The following describes the studies and summarizes some of the informal feedback provided by the experts.

8.1 Load Balancing of LULESH

LULESH [53] is a proxy application used for modeling the performance of large hydrodynamics simulations. LULESH represents the numerical algorithms, data motion, and programming style typical of scientific applications, and is used for studying the performance of different parallel programming models and architectures. Here, we use CALLFLOW to understand the performance of LULESH when implemented using Adaptive MPI (AMPI) [54] for solving a problem that represents multimaterial systems. AMPI is a paradigm of MPI applications with overdecomposition, i.e., multiple MPI ranks per process instead of the commonly used one rank per process.

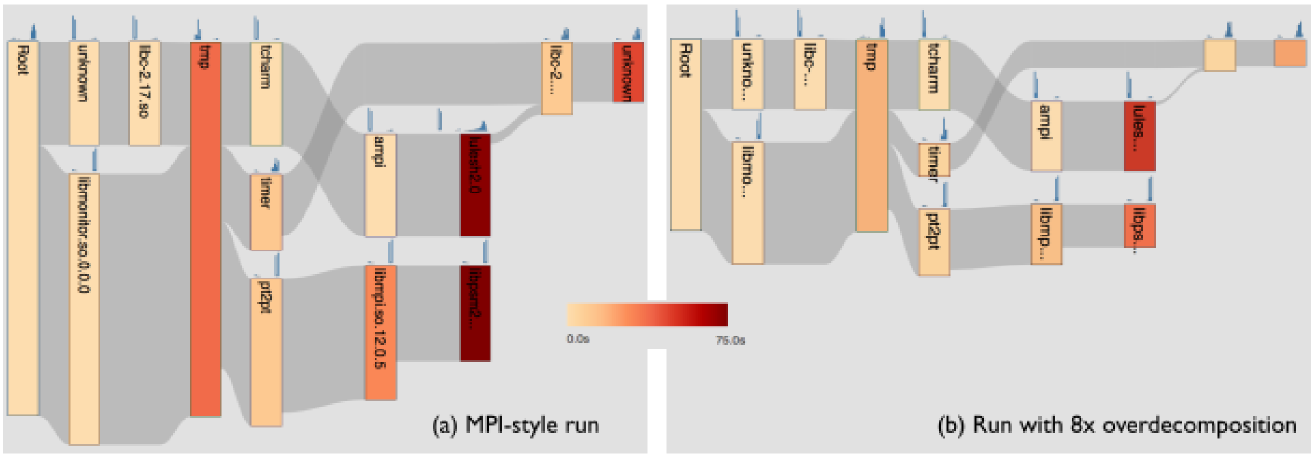


Fig. 7: Understanding the impact of over-decomposition on LULESH. (a) MPI-style execution with 1 MPI rank per process: most time spent in LULESH internals, MPI/`libpsm`, AMPI internals (*tmp* bar), and C-library. (b) With over-decomposition, time spent in most modules reduces, but histograms of several modules show significant load imbalance.

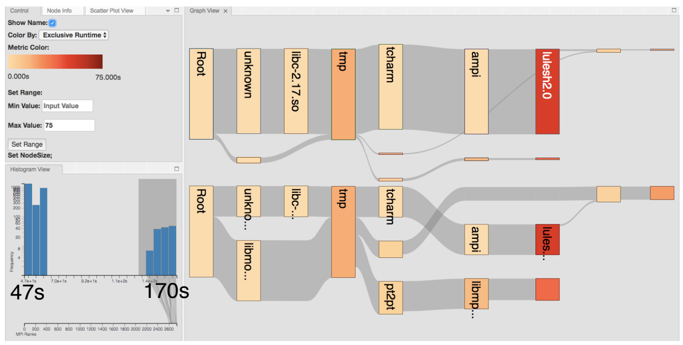

Show Name: ✓
Color By: Exclusive Runtime
Metric Color:
0.000s 75.000s
Set Range:
Min Value: Input Value
Max Value: 75
Set Range
Set NodeSize:

Fig. 8: Analysis of load imbalance in LULESH by dividing the processes based on the time spent in LULESH internals. The splitting shows that processes with light load for LULESH internals (bottom view) spend significant time in MPI/`libpsm`, AMPI, and C-library, while remaining processes (top view) spend minimal time in these modules.

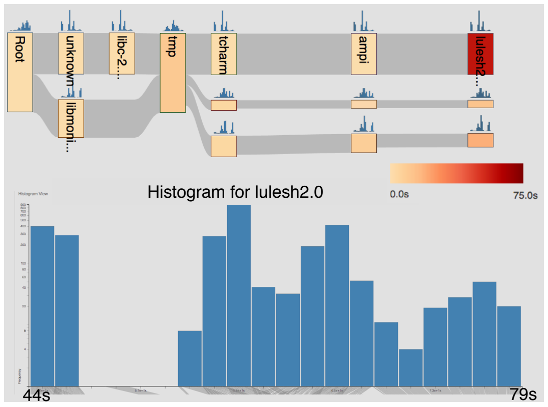


Fig. 9: Impact of load balancing for LULESH (bar heights scaled as Fig. 7): total runtime decreases and time spent in `libc` and MPI/`libpsm` is reduced. The load for LULESH internals is more evenly distributed across processes.

Fig. 7(a) visualizes an execution of the AMPI version of LULESH that emulates the traditional MPI model where one MPI rank is placed on every process in the system. By coloring the nodes based on exclusive runtime, CALLFLOW helps identify the distribution of time among LULESH internals, AMPI framework, and other modules. We find that, on average, LULESH internals and MPI/`libpsm` (the lower-level messaging libraries) account for the majority of runtime, and exhibit load imbalance among processes. Surprisingly, `ampi` and `libc` also show significant runtime.

Significant time spent in MPI/`libpsm` and the load imbalance across processes suggest that AMPI's ability to overdecompose MPI ranks and adaptively overlap computation with communication can improve performance. To test this hypothesis, we run eight logical MPI ranks on every process in the next experiment. This results in a reduction in the execution time by 44% as highlighted by the difference in the height of the root module in Figs. 7(a) and 7(b). The module view eases the task of identifying the code regions that benefit from overdecomposition. Unexpectedly, the time spent in the AMPI runtime decreases despite the fact that an eight-fold overdecomposition leads to eight times more messages and scheduling overhead. Further, most of this improvement appears to come from less time spent driving the communication in MPI/`libpsm`. Since both AMPI and MPI/`libpsm` are large and complex frameworks, their respective nodes in CALLFLOW abstract a large number of CCT nodes across many levels and contexts. Therefore, arriving at these insights from a traditional CCT display would require substantial effort as well as an initial guess to focus on these two components. Instead, CALLFLOW's super graph view immediately highlights the most important differences in the runtimes effectively.

Despite the reduced runtime, the histogram for most modules (see Fig. 7(b)) appear to be heavily skewed. To explore this further, we split the processes based on the time spent in LULESH (see Fig. 8). The split view reveals that only the processes with light load for LULESH internals spend a large amount of time in MPI/`libpsm`, `ampi`, and `libc`. Discovering such a high-level correlation among different modules is difficult using traditional CCT tools. In this case, these results inform the need for load balancing the work done by the LULESH across processes.

Next, we enable load balancing in AMPI, which resulted in 30% better performance. The resulting profile (see

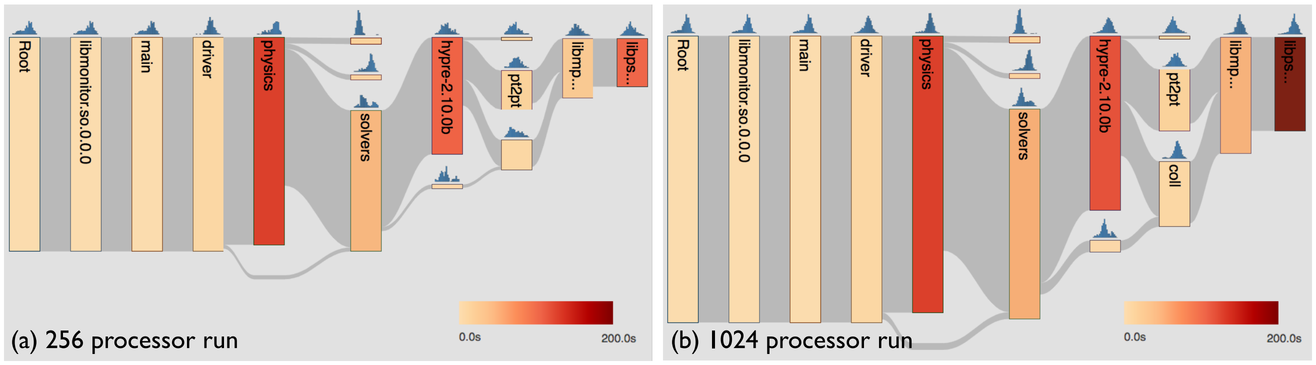


Fig. 10: Comparison of Miranda execution on 256 and 1024 processors; height and color of a bar represent the inclusive and exclusive time spent in the module. These visualizations help identify the modules in which significant time is spent in the execution, and reveal that the increase in the exclusive time spent in the `libpsm` module is the primary reason for lower performance on 1024 processors.

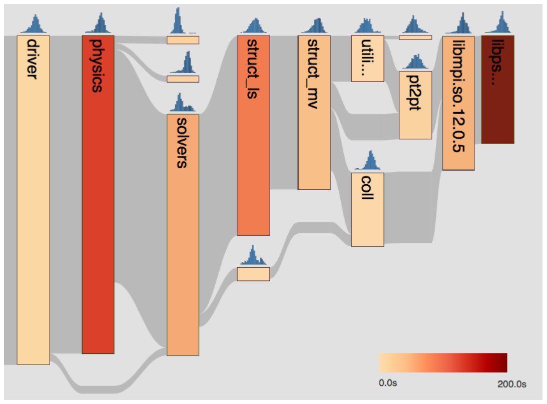


Fig. 11: Splitting `Hypre` allows identifying its components responsible for performance-degrading communication.

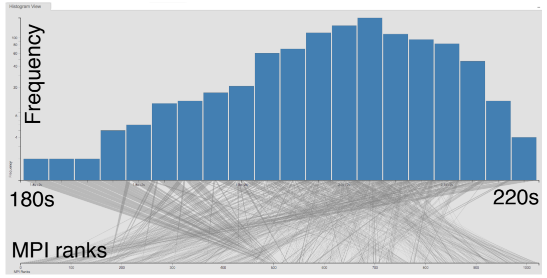


Fig. 12: Histogram for time spent in the `libpsm` module: the distribution is normal and not heavily skewed; the bin-to-MPI rank connections reveal that lower-ranked MPI processes are more likely to have higher `libpsm` time.

Fig. 9), helps understand that the performance benefits are driven by more evenly load for LULESH internals among processes. The view also shows that the time spent in `ampi`, MPI/`libpsm`, and `libc` is reduced significantly.

8.2 Scaling Performance of Miranda

Miranda [13] is a large-scale parallel code that simulates radiation hydrodynamics for direct numerical solution or large-eddy simulation. In order to simulate large-scale scenarios, it is desirable that Miranda exhibits good weak scaling, i.e., execution time should not increase significantly when more processors are used to solve larger problems. However, Miranda developers have observed poor scaling behavior for Miranda. To investigate the causes of degraded performance, we obtained CCT profiles of Miranda at two different process counts: 256 processes and 1024 processes, and noticed that even though the problem size per process is kept fixed, the execution time increases by more than 30%.

Fig. 10(a) shows the CALLFLOW visualization for a Miranda execution on 256 processes. For performance experts analyzing the behavior of Miranda, who are not familiar with Miranda, the visualization provides a high-level overview of the control flow of Miranda, and the dependencies and relationships between different modules. It is typically difficult to obtain such information from CCTs because hundreds of functions, often with unfamiliar names, are used in production codes. In contrast, it is tractable for performance experts to get familiar with program modules and commonly used external libraries. Further, coloring the modules by exclusive time spent in them helps identify the modules that make up most of the overall execution time. In this case, three modules stand out: `physics` (Miranda's science code), `Hypre` (a linear solvers library), and `libpsm` (the lower-level messaging library underneath MPI). Fig. 10 also shows a juxtaposed comparison of profiles from 256 process and 1024 process executions. To facilitate such a comparison, we use CALLFLOW's feature to rescale the height of the root modules based on their inclusive time (which is also the total execution time) and use a common time range for coloring the two super graphs based on the exclusive time. The contrast in the color of the `libpsm` module in the two figures immediately identifies it as one of the culprits for performance degradation. However, the visualization also reveals that most of the time spent in the `libpsm` library can be traced back to the `Hypre` module, implying that the cause of poor scaling of Miranda is poor scaling of the latter.

To explore the components in `Hypre` that are responsible for the time spent in `libpsm`, the nodes are further split to reveal high-level control flow and inefficiency sources inside the `Hypre` library (see Fig. 11). The module-based split reduces the work that a domain expert would need to do to identify that `utility` and `struct_mv` components of `Hypre` invoke point-to-point communication calls in MPI that result in half of the increased `libpsm` time. Similarly, we find that the collective calls made by the `struct_mv` module causes the remaining performance degradation.

When the time spent in messaging affects the performance, experts tend to analyze the time distribution across processes to find the root causes. With CALLFLOW, such a histogram for `libpsm` (or for any other module) can be obtained simply (see Fig. 12). We observe that for the Miranda execution on 1024 processes, the distribution of time spent in `libpsm` is not heavily skewed and has a narrow time range, suggesting that the increased communication volume is likely the cause of the increased time. If load imbalance or system noise would have been the culprit, a more skewed time distribution would be obtained. Finally, the bin-to-MPI rank connections shown below the histogram reveal that lower-ranked MPI processes are more likely to spend higher time in the `libpsm` module. Such an insight would have been difficult to obtain in traditional tools and can help identify systematic-bias in the code.

9 CONCLUSION

CALLFLOW is a visualization tool for exploring the calling context trees (CCTs) of application codes, particularly useful for large-scale parallel codes. Through an easy-to-understand flow-based visual metaphor in the form of Sankey diagrams, CALLFLOW helps the users identify performance bottlenecks in the code effectively, leading to potential optimizations and improved overall throughput of applications. Catering specifically to the target data, CALLFLOW customizes and enhances the layout of Sankey diagrams, and uses multiple linked views to provide a holistic exploration of CCTs. Through a set of interactive operations on the underlying graph, CALLFLOW provides both a high-level, system-oriented overview of CCTs as well as the ability to drill down to detailed information, making, for the first time, large-scale CCTs accessible and explorable. CALLFLOW has been developed in close collaboration with domain scientists, and has already garnered significant interest in our institute. To expand accessibility, CALLFLOW is publicly released under MIT License and a domain-specific publication establishing its larger impact in the HPC community is forthcoming. In this paper, we demonstrate the effectiveness of CALLFLOW through investigation of production codes, and delivering insights leading to their improvement.

Despite the initial successful use cases and positive feedback from the domain experts, there exist several avenues for further research and development. Since the layout of Sankey diagrams is key to the easy navigation of the visualization, we would like to explore and evaluate other layout optimization techniques. For example, PQR-trees [55], may produce more user-friendly layouts, at an additional but affordable cost. Using constraint programming for graph layouts, as suggested by IPSep-CoLa [56] and Zarate et al. [57] may be another avenue. With respect to stable changes in the layout for animation, the DynaDAG approach [58], appears attractive, although it may require modifications to support large-scale graphs. CALLFLOW currently only loads one dataset a time. However, simulation codes typically run under different conditions and it is necessary to compare the performance under these conditions. We plan to extend CALLFLOW to support multiple datasets and incorporate animation to transition between different data. This would also support streaming data. We would also like to support recursive callings between functions as many applications in HPC uses such techniques. Finally, although the paper concentrates on parallel applications, the techniques described in this paper may also be applied to analyze hierarchical and time sequence data.

10 DISCUSSION

Most profiling visualization tools use UML sequence diagrams, and adopt a node-link visualization layout to visualize the CCT. Although, node-link representation is intuitive, they quickly become difficult to explore when the number of components becomes too large. On the contrary, CALLFLOW transforms the original CCT into a super graph, which reduces the number of nodes to be visualized and presents the control flow of the application as code modules. Since grouping nodes into modules could introduce nodes having multiple call paths or nodes with multiple parents, hierarchical space-filling visualization layouts are not suitable for exploration. Additionally, since the hierarchy is not predetermined but instead emerges dynamically depending on the given data and analysis tasks, static visualization approaches do not provide exhaustive exploration. CALLFLOW supports flexible splitting of modules according to the parent nodes or by the function group (i.e., all collective or all point-to-point messages within the MPI library).

CALLFLOW was designed for a critical, yet a somewhat specific problem of analyzing call graphs. Although the results of this research may not appear directly generalizable, our attempt has been to focus instead on the idea of transferability [46] to other possibly similar domains. The data abstractions presented in Section 6, e.g., the principle of node splitting, may support transferring our specific design and results to other application domains that deal with distributing shared resource across hierarchical entities, e.g., certain types of temporal evolution [59]. Another example is nested schedules of large construction projects, which naturally follow the same pattern of high-level phases, i.e., excavation, framing, etc., and with more detailed breakdowns, each item with corresponding time and money estimates. The key insight from CALLFLOW is that the combination of the hierarchical representation, i.e., trees, with graph metaphors (combining low-level tree nodes) can provide powerful insights. Continuing the construction example, one may consider all jobs requiring specific skills or materials to form a graph indicating possible bottlenecks. Nevertheless, the detailed linked views, i.e., the code view in our application, would be application specific. Similarly, the exact layout of the Sankey diagram could be optimized according to specific needs.

ACKNOWLEDGMENTS

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under contract DE-AC52-07NA27344. The UC Davis researchers are also supported in part by the Department of Energy through grant DE-SC0014917.

REFERENCES

[1] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[2] 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," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, 2010.

[3] K. Ali and O. Lhoták, "Application-only call graph construction," in *European Conf. on Object-Oriented Prog.*, 2012, pp. 688–712.

[4] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 108–124, 1997.

[5] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *ACM Trans. on Programming Languages and Systems (TOPLAS)*, vol. 23, no. 6, pp. 685–746, 2001.

[6] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The SCALASCA performance toolset architecture," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 702–719, 2010.

[7] B. Mohr and F. Wolf, "KOJAK – A tool set for automatic performance analysis of parallel programs," in *Euro-Par 2003 Parallel Processing*, 2003, pp. 1301–1304.

[8] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[9] B. Johnson and B. Shneiderman, "Tree-Maps: A space-filling approach to the visualization of hierarchical information structures," in *Proc. of the 2nd Conf. on Visualization*. IEEE Computer Society Press, 1991, pp. 284–291.

[10] J. B. Kruskal and J. M. Landwehr, "Icicle plots: Better displays for hierarchical clustering," *The American Statistician*, vol. 37, no. 2, pp. 162–168, 1983.

[11] G. Ammons, T. Ball, and J. R. Larus, "Exploiting hardware performance counters with flow and context sensitive profiling," *SIGPLAN Not.*, vol. 32, no. 5, pp. 85–96, 1997.

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

[13] W. H. Cabot, A. W. Cook, P. L. Miller, D. E. Laney, M. C. Miller, and H. R. Childs, "Large-eddy simulation of rayleigh–taylor instability," *Physics of Fluids*, vol. 17, no. 9, p. 091106, 2005.

[14] J. Abello, F. Van Ham, and N. Krishnan, "ASK-GraphView: A large scale graph visualization system," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 12, no. 5, pp. 669–676, 2006.

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

[16] Q. V. Nguyen and M. L. Huang, "A space-optimized tree visualization," in *IEEE Symp. on Info. Vis.*, 2002, pp. 85–92.

[17] C. Plaisant, J. Grosjean, and B. B. Bederson, "Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation," in *Proc. of the IEEE Symp. on Info. Vis.*, 2002, pp. 57–64.

[18] T. Munzner and P. Burchard, "Visualizing the structure of the world wide web in 3d hyperbolic space," in *Proc. of the First Symp. on Virtual Reality Modeling Language*. ACM, 1995, pp. 33–38.

[19] G. G. Robertson, J. D. Mackinlay, and S. K. Card, "Cone trees: Animated 3d visualizations of hierarchical information," in *Proc. of the Conf. on Human Factors in Computing Systems*, 1991, pp. 189–194.

[20] 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 - STARs*, 2014.

[21] T. D. LaToza and B. A. Myers, "Visualizing call graphs," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC).*, 2011.

[22] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "On the readability of graphs using node-link and matrix-based representations: A controlled experiment and statistical analysis," *Info. Vis.*, vol. 4, no. 2, pp. 114–135, 2005.

[23] H. Bhatia, N. Jain, A. Bhatele, Y. Livnat, J. Domke, V. Pascucci, and P.-T. Bremer, "Interactive investigation of traffic congestion on fat-tree networks using TreeScope," *Comp. Graph. Forum*, vol. 37, no. 3, pp. 561–572, 2018.

[24] L. DeRose, B. Homer, and D. Johnson, "Detecting application load imbalance on high end massively parallel systems," in *Euro-Par 2007 Parallel Processing*, 2007, pp. 150–159.

[25] H. T. Nguyen, L. Weit, A. Bhatele, T. Gamblin, D. Boehme, M. Schulz, K.-L. Ma, and P.-T. Bremer, "VIPACT: A visualization interface for analyzing calling context trees," in *Proc. of the 3rd Int. Work. on Visual Perf. Analysis*. IEEE Press, 2016, pp. 25–28.

[26] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems," in *Proceedings of the 2006 ACM symposium on Software visualization*, 2006.

[27] C. Xie, W. Xu, and K. Mueller, "A visual analytics framework for the detection of anomalous call stack trees in high performance computing application," *IEEE Trans. on Vis. and Comp. Graph.*, 2019.

[28] M. Burch, C. Müller, G. Reina, H. Schmauder, M. Greis, and D. Weiskopf, "Visualizing dynamic call graphs," in *Vision, Modeling and Visualization*, M. Goesele, T. Grosch, H. Theisel, K. Toennies, and B. Preim, Eds. The Eurographics Association, 2012.

[29] B. Johnson, "TreeViz: Treemap visualization of hierarchically structured information," in *Proc. of the Conf. on Human Factors in Computing Systems*, 1992, pp. 369–370.

[30] B. Shneiderman and M. Wattenberg, "Ordered treemap layouts," in *Proc. of the IEEE Symp. on Info. Vis.*, 2001, pp. 73–78.

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

[32] G. Sander, "Graph layout through the vcg tool," in *Graph Drawing*. Springer Berlin Heidelberg, 1995, pp. 194–205.

[33] F. Balmas, "Displaying dependence graphs: a hierarchical approach," *J. Softw. Maint. Evol.*, vol. 16, no. 3, pp. 151–185, 2004.

[34] S. Devkota and K. E. Isaacs, "CFGExplorer: Designing a visual control flow analytics system around basic program analysis operations," in *Comp. Graph. Forum*, vol. 37, no. 3, 2018, pp. 453–464.

[35] M. Schmidt, "The sankey diagram in energy and material flow management," *J. Industrial Ecology*, vol. 12, no. 1, pp. 82–94, 2008.

[36] K. Soundararajan, H. K. Ho, and B. Su, "Sankey diagram framework for energy and exergy flows," *Applied Energy*, vol. 136, pp. 1035–1042, 2014.

[37] H. Alemasoom, F. Samavati, J. Brosz, and D. Layzell, "EnergyViz: An interactive system for visualization of energy systems," *Vis. Comput.*, vol. 32, no. 3, pp. 403–413, 2016.

[38] M. Ogawa, K.-L. Ma, C. Bird, P. Devanbu, and A. Gourley, "Visualizing social interaction in open source software projects," in *6th Int. Asia-Pacific Symp. on Visualization*, 2007, pp. 25–32.

[39] K. Wongsuphasawat and D. Gotz, "Outflow: Visualizing patient flow by symptoms and outcome," in *IEEE VisWeek Work. on Visual Analytics in Healthcare*, 2011, pp. 25–28.

[40] C.-F. Wang, J. Li, K.-L. Ma, C.-W. Huang, and Y.-C. Li, "A visual analysis approach to cohort study of electronic patient records," in *IEEE Int. Conf. on Bioinformatics and Biomedicine*, 2014, pp. 521–528.

[41] J. J. V. Wijk, "Bridging the gaps," *IEEE Computer Graphics and Applications*, vol. 26, no. 6, pp. 6–9, Nov 2006.

[42] A. J. Pretorius and J. J. Van Wijk, "What does the user want to see?: What do the data want to be?" *Information Visualization*, vol. 8, no. 3, pp. 153–166, 2009.

[43] M. Brehmer, J. Ng, K. Tate, and T. Munzner, "Matches, mismatches, and methods: Multiple-view workflows for energy portfolio analysis," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 22, no. 1, pp. 449–458, 2016.

[44] R. A. Amar and J. T. Stasko, "Knowledge precepts for design and evaluation of information visualizations," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 11, no. 4, pp. 432–442, 2005.

[45] T. Munzner, "A nested model for visualization design and validation," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 15, no. 6, pp. 921–928, 2009.

[46] M. Sedlmair, M. Meyer, and T. Munzner, "Design study methodology: Reflections from the trenches and the stacks," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 18, no. 12, pp. 2431–2440, 2012.

[47] B. Shneiderman and C. Plaisant, "Strategies for evaluating information visualization tools: Multi-dimensional in-depth long-term case studies," in *Proceedings of the 2006 AVI Workshop on BEyond Time and Errors: Novel Evaluation Methods for Information Visualization*, ser. BELIV '06. New York, NY, USA: ACM, 2006, pp. 1–7.

[48] D. Lloyd and J. Dykes, "Human-centered approaches in geovisualization design: Investigating multiple methods through a long-term case study," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 17, no. 12, pp. 2498–2507, Dec 2011.

[49] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "A systematic literature review of software visualization evaluation," *Journal of Systems and Software*, vol. 144, pp. 165–180, 2018.

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

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

[52] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing*. Springer Berlin Heidelberg, 2008, pp. 139–155.

[53] I. Karlin, A. Bhatele, B. L. Chamberlain, J. Cohen, Z. Devito, M. Gokhale, R. Haque, R. Hornung, J. Keasler, D. Laney, E. Luke, S. Lloyd, J. McGraw, R. Neely, D. Richards, M. Schulz, C. H. Still, F. Wang, and D. Wong, "LULESH programming model and performance ports overview," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-608824, 2012.

[54] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance evaluation of adaptive MPI," in *Proc. of 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2006, pp. 12–21.

[55] J. R. Marchete Filho and C. Silva, "Using PQR-trees for reducing edge crossings in layered directed acyclic graphs," in *Work. of Works in Progress (WIP) in SIBGRAPI*, 2013.

[56] T. Dwyer, Y. Koren, and K. Marriott, "IPSep-CoLa: An incremental procedure for separation constraint layout of graphs," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 12, no. 5, pp. 821–828, 2006.

[57] D. C. Zarate, P. L. Bodic, T. Dwyer, G. Gange, and P. Stuckey, "Optimal sankey diagrams via integer programming," in *IEEE Pacific Vis. Symp.*, April 2018, pp. 135–139.

[58] S. C. North, "Incremental layout in DynaDAG," in *Graph Drawing*. Springer Berlin Heidelberg, 1995, pp. 409–418.

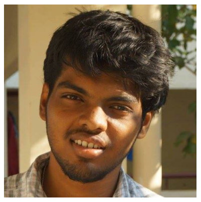
[59] M. Monroe, R. Lan, H. Lee, C. Plaisant, and B. Shneiderman, "Temporal event sequence simplification," *IEEE Trans. on Vis. and Comp. Graph.*, vol. 19, no. 12, pp. 2227–2236, Dec 2013.

Huu Tan Nguyen received his Master's degree in Computer Science at the University of California, Davis in 2017. He obtained his Bachelor's degree in Computer Science and Engineering at the University of California, Davis in 2015. His research interests include information visualization and data analysis. He is currently with Keysight Technologies as a Software Engineer.

Abhinav Bhatele is an assistant professor in the department of computer science at the University of Maryland, College Park. Previously, he was a senior computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research interests are broadly in systems and networks, with a focus on parallel computing and big data analytics. He has published research in programming models and runtimes, network design and simulation, applications of machine learning to parallel systems, and on analyzing, modeling and optimizing the performance of parallel software and systems.

Abhinav received a B.Tech. degree in Computer Science and Engineering from I.I.T. Kanpur, India in May 2005, and M.S. and Ph.D. degrees in Computer Science from the University of Illinois at Urbana-Champaign in 2007 and 2010 respectively. Abhinav was an ACM-IEEE CS George Michael Memorial HPC Fellow in 2009. He has received best paper awards at Euro-Par 2009, IPDPS 2013 and IPDPS 2016. Abhinav was selected as a recipient of the IEEE TCSC Young Achievers in Scalable Computing award in 2014, and the LLNL Early and Mid-Career Recognition award in 2018.

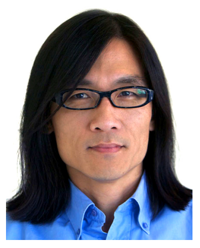
Nikhil Jain is a Sidney Fernbach postdoctoral fellow in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. He works on topics related to parallel computing including networks, scalable application development, parallel algorithms, communication optimization, and interoperation of languages. Nikhil received a Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 2016, and B.Tech. and M.Tech degrees in Computer Science and Engineering from I.I.T. Kanpur, India in May 2009.



Suraj kesavan is a 2nd year graduate student studying computer science at University of California, Davis. He obtained his bachelor's degree from National Institute of Technology, Tiruchirappalli. His current research interests include information visualization and data analytics.

Harsh Bhatia is a Computer Scientist at the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research spans the broad area of visualization and computational topology, with special focus on scientific data. Harsh is also interested in ML-based approaches for scientific applications. Prior to joining LLNL, Harsh earned his Ph.D. from Scientific Computing & Imaging Institute at the University of Utah in 2015, where he worked on the feature extraction for vector fields.

Todd Gamblin is a computer scientist in the Center for Applied Scientific Computing at Lawrence Livermore National Laboratory. His research focuses on scalable tools for measuring, analyzing, and visualizing parallel performance data. For this work, he received an Early Career Research Award from the U.S. Department of Energy in 2014. In addition to his research, Todd leads LLNL's HPC Developer Ecosystem team, and he is the creator of Spack, a popular HPC package management tool. Todd has been at LLNL since 2008. He received Ph.D. and M.S. degrees in Computer Science from the University of North Carolina at Chapel Hill in 2009 and 2005. He received his B.A. in Computer Science and Japanese from Williams College in 2002.



Kwan-Liu Ma , an IEEE Fellow, is a distinguished professor of computer science at the University of California, Davis. His research interests include visualization, computer graphics, high-performance computing, and human-computer interaction. He has served as a papers co-chair for SciVis, InfoVis, EuroVis, PacificVis, and Graph Drawing, as an associate editor of IEEE TVCG (2007-2011) and IEEE CG&A (2007-2013), and as an AEIC of IEEE CG&A (2013-2019). Contact him via email: ma@cs.ucdavis.edu.

Peer-Timo Bremer is a member of technical staff and project leader at the Center for Applied Scientific Computing (CASC) at the Lawrence Livermore National Laboratory (LLNL) and Associated Director for Research at the Center for Extreme Data Management, Analysis, and Visualization at the University of Utah. Prior to his tenure at CASC, he earned a Ph.D. in Computer science at the University of California, Davis in 2004 and a Diploma in Mathematics and Computer Science from the Leibniz University in Hannover, Germany in 2000.