Title

Visual Analytics Techniques for Investigating Large-Scale HPC Profiles and Trace Data

Author

Padmanaban Kesavan, Suraj

Publication Date

2023

Peer reviewed|Thesis/dissertation

Visual Analytics Techniques for Investigating Large-Scale HPC Profiles and Trace Data

By

Suraj Padmanaban Kesavan

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Kwan-Liu Ma, Chair

Harsh Bhatia

Jason Lowe-Power

Committee in Charge

2023

To my family and friends who always push me to give my absolute best.

Contents

Abstract

Visual Analytics Techniques for Investigating Large-Scale HPC Profiles and Trace Data

Performance visualization is an emerging field that adapts to the growing ecosystem of High-Performance Computing (HPC). With the continued growth in scale and complexity of HPC systems, code developers face the challenge of optimizing performance, requiring a detailed understanding of runtime behaviors of the system and the ability to identify and address performance bottlenecks. To meet this challenge, various tools have been developed to capture performance behaviors and conduct performance analysis, but the complexity and scale of the resulting data have made visualization systems essential for revealing and understanding key patterns that expose performance bottlenecks. The dissertation presents novel visual analytic methods that address the four key data challenges associated with performance analysis: *Attribution*, *Scalability*, *Correlation*, and *Velocity*. My contributions are driven by domain experts' requirements to facilitate scalable interactive analysis of large-scale performance profiles and traces. Specifically, the dissertation addresses the challenges of analyzing and correlating collected performance data across three key domains of supercomputing — *Hardware*, *Application*, and *Communication* (HAC). Overall, the dissertation addresses the challenges of analyzing and correlating collected performance data in HAC domains, leading to a set of tools that are expected to greatly enhance HPC experts' ability to understand and optimize the performance of demanding applications through interactive and scalable visual analytics.

Acknowledgments

I would like to express my deepest gratitude to my dissertation advisor, Prof. Kwan-Liu Ma, for his guidance, support, and mentorship throughout my Ph.D. journey. His patience, wisdom, and encouragement have been invaluable to me and have helped shape my academic and personal growth. I would also like to thank Dr. Harsh Bhatia, Dr. Olga Pearce, Dr. Abhinav Bhatele and other research scientists from Lawrence Livermore National Laboratory for their collaboration, guidance and support through the research. I would also like to thank the other members of my dissertation committee for their time, expertise, and constructive feedback that greatly improved the quality of this work.

My sincere appreciation goes to my family and friends – Mom, Dad, Advaitaa, Deepika, Ashwin, Pranav, Somya, Karnali, Swathi, Natchan, Marissa, Agrarian coop, and soo many others, for their unwavering love, support, and encouragement, which gave me the strength and motivation to pursue my academic goals. I also want to thank the labmates of VIDI at UC Davis, especially Dr. Jia-Kai Chou, Dr. Takanori Fujiwara, Dr. Oh-Hyun Kwon, Sandra Bae, Dr. Jianping Kelvin Li, Dr. Tyson Neuroth, Dr. Shilpika, Dr. Xiaoyu Zhang, and Keshav Dasu. The discussions with them were always engaging and serendipitously provided me hints that led to new research ideas.

Finally, I thank all the individuals who have contributed to this work in various ways and have made it possible, including my research collaborators, funding agencies, and study participants.

Thank you all for being a part of this journey.

Chapter 1
Introduction

1.1 Performance Analysis and Visualization for HPC

High-Performance Computing (HPC) systems enable scientists to study intricate scientific phenomena using large-scale simulations and advance the frontiers of human understanding in multiple disciplines: from weather forecasting [1] and earthquake modeling [2] to molecular dynamics [3] and drug design [4]. Prominent examples of such systems include the leadership class supercomputers deployed by the U.S. Department of Energy, such as Oak Ridge National Laboratory's Summit, Lawrence Livermore National Laboratory's Sierra, and El Capitan, which is the first Exascale machine (i.e., capable of performing quintillion (10^{18}) floating point operations per second) planned for 2023 [5]. The simulation codes are executed in parallel on large-scale computing resources that distribute the compute over thousands of distributed nodes with multiple processors for efficient computation. To maximize the amount of science per dollar and insights per Watt, computational scientists and HPC code developers continuously strive towards optimizing the performance of their simulation codes. Therefore, achieving efficiency while performing extreme-scale computations becomes crucial to meeting optimal performance.

In pursuit of improved computational performance, code developers conduct *performance analysis* to achieve optimal performance while consuming the least amount of resources. Since performance issues and bottlenecks can occur among any of the domains and at different points of application execution, code developers require a complete picture to break down *"where"* and *"why"* performance might struggle. Therefore, programmers are constantly required to measure, analyze, and understand the behavior of their applications across (a) *Hardware*, (b) *Application*, and (c) *Communication* (HAC) domains with great detail [6]. The hardware domain consists of network nodes and physical

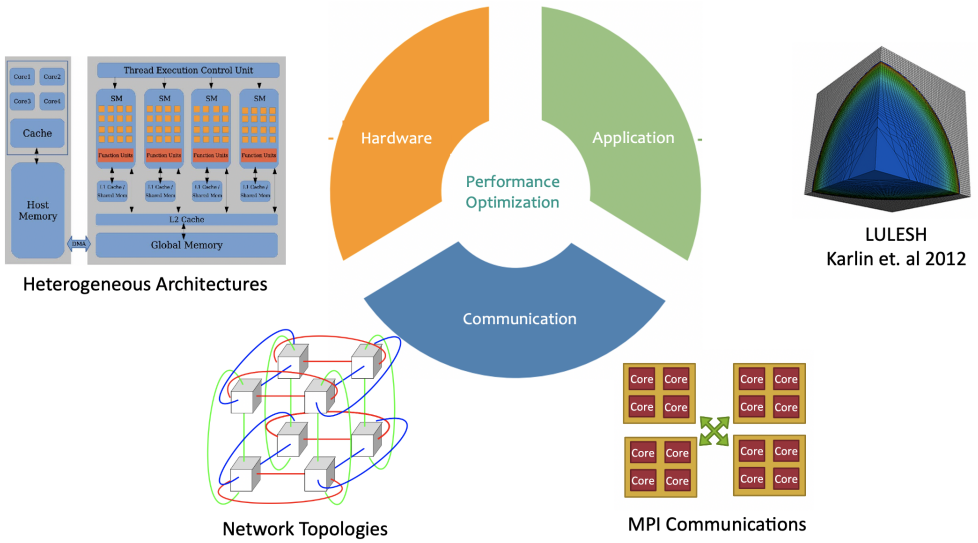


Figure 1.1: Performance data collected across the Hardware, Application and Communication (HAC) domains play a critical role in the task of performance optimization.

links between them, the application domain represents the physical or simulated system designed to solve the underlying problem, and the communication domain represents a communication network, which captures the communication patterns of the application. As hardware and communication networks evolve and the application's usage patterns change, developers must adapt their code to ensure an optimal performance. By conducting performance analysis routinely, developers can ensure that their code is running efficiently and providing the best possible user experience.

Multiple tools have been designed to capture performance data across the HAC domains. A typical HPC performance analysis workflow comprises of three stages. First, domain experts run the application along with performance monitoring tools such as HPCToolkit [7] and Tau [8] to collect performance metrics. Next, considerable efforts are put into data analysis to understand the underlying performance behaviors. Finally, after studying the behaviors, one might be able to tune or tweak certain application parameters for better performance based on the analysis results. From the hardware perspective, supercomputers are upgraded with newer hardware approximately every few years, requiring developers to port their application to utilize the recent updates. Second, for each application, the software and hardware technologies continue to evolve through

the history of an application with new technological advancements. Finally, at exascale the quantity and complexity of performance data grow by orders of magnitude since large number of compute nodes work in a parallel distributed fashion, making traditional and manual performance analysis infeasible.

Performance visualization is an effective medium for application developers to assess the performance of the software and understand the potential causes and severity of found bottlenecks [9]. By observing key patterns and behaviors, code developers and HPC analysts can improve their application's time and energy efficiency. While complex visualizations can elucidate novel or interesting performance data, it is important to keep in mind that the visualization has to aid the developer in accomplishing some set of performance optimization goals. Domain experts must also incorporate different kinds of analyses for each domain to derive fruitful insights. For example, one would be interested in analyzing compute node performance in the hardware domain, finding the expensive code regions from the application domain, and detecting network congestions from the communication domain. *Visual analytics* further helps bridge the gap between user knowledge and insights various statistical metrics can provide. As a result, visual analytic frameworks are favored as they can drive performance analysis by facilitating decisions to help develop novel optimization strategies.

This dissertation focuses on the development of three visual analytic frameworks that enable domain experts to study and analyze performance data collected from large-scale supercomputing applications. The frameworks incorporate *automated data analysis* with *interactive visual analytics*. To study the performance data collected from the HAC domains, *sampled profiles* and *event traces* are extensively studied to detect performance bottlenecks and enable domain experts to realize avenues for further performance improvements in their code and system design.

1.2 Challenges to Performance Visualization

Performance visualization is an essential tool for exploring and communicating trends in the data, and its utility has well served the HPC community. As HPC technologies get

constantly upgraded over time, performance visualization is also required to keep evolving to meet the demands. Due to increasing data sizes and the emergence of Big Data, the need for massive parallelization is a primarily driving a number of visualization research challenges. With growing demands, it stresses the need for visualization to represent, process, and interact with data in efficient ways. This places great importance on efficient data representation, processing, and interaction in visualization. In this work, I have identified four key data challenges: (a) *Attribution*, (b) *Scalability*, (c) *Correlation*, and (d) *Velocity* which form the backbone of this dissertation research.

1.2.1 Attribution

Attribution is a process by which one can explain the causes of behaviors and events. Pinpointing execution behaviors is a critical task when it comes to performance analysis. Experts in HPC are not strangers to performance visualization as it has been an integral component of their workflow. However, attribution of data to relate performance behaviors has been a challenging task due to the lack of a design process during development. To address this increasing concern, all techniques introduced in this work have been designed and developed using *information visualization*.

"Information visualization is a process of representing data in a visual and meaningful way so that users to make hypotheses, look for patterns and exceptions, and then refine their hypothesis." To improve the attribution of performance behaviors, I collaborate with domain experts from HPC, which include software developers as well as performance analysts who assist computational scientists in optimizing their parallel computations. Through an interactive collaboration with domain experts, including interviews and discussions, I characterize the domain problem, identify the primary requirements, formulate actionable tasks, and finally, design suitable interactive visualization techniques. Through this iterative process, I was able to support HPC-critical tasks such as exploratory (see Chapter 3), comparative (see Chapter 4 and Chapter 5), post-hoc (see Chapter 5 and Chapter 7), and in-situ (see Chapter 6) analysis.

1.2.2 Scalability

With computational abilities reaching exascale computing, increasing effort is required to understand how various software is run on such machines and optimize their performance. As the scale of computing resources continues to grow exponentially, and along with it, the scale of the collected performance data, it is becoming increasingly critical to create highly scalable performance visualizations. Scalability for performance analysis can be dissected into two levels: *instance-level* and *ensemble-level*. At the instance level, performance behaviors must be studied across domains for a single execution. With multiple recorded metrics and increasing available compute nodes, the scale and complexity of collected data can over-power even the simplest of analyses. At the ensemble level, HPC developers often explore various execution parameters, such as hardware configurations, system software choices, and application parameters, and are interested in detecting and understanding bottlenecks across different executions.

Exploration of large-scale performance data requires *information preprocessing* in order to reduce the collected data size to processible levels. Suitable metrics must be collected to obtain similarities and dissimilarities to reduce the visual load on the end user. Furthermore, the degree of abstraction has to be controlled interactively in order to browse information space at arbitrary levels of detail. To manage scalability at both instance and ensemble levels, I design techniques and algorithms that simplify large unstructured sampled profiles and event traces by highlighting the regions of interest to the HPC community. Additionally, the developed frameworks provide support for sufficient *querying and aggregating* capabilities to enable interactive sifting through the massive scale data and provide an appropriate mapping between data and frame of reference.

1.2.3 Correlation

Analyzing behaviors and optimizing the performance of HPC applications requires the collection of a multitude of performance metrics across HAC domains. As a result, HPC datasets are a conglomeration of different quantities (*e.g.,* metrics, attributes, metadata), and experts are particularly interested in the correlation between any two quantities or variables. The correlation of hardware, communication, and application domains

allows application developers to understand how structures in their simulated domains correspond to real performance problems, and it also gives systems engineers a new, more intuitive perspective on how simulations map to physical hardware.

Although it is practically impossible to obtain complete knowledge of how all of the different quantities are related, it is necessary to have strategies to visualize multiple quantities simultaneously or to visualize the correlations between them. To this end, I integrate visualizations that present basic statistical measures like *standard deviation*, *covariance*, *correlation coefficient*, and *causality* during analysis.

1.2.4 Velocity

An ongoing trend in HPC is an exponential increase in the computational throughput of the machine, but a comparatively much smaller increase in the bandwidth to the disk storage system. This has led to a very large disparity between the computation bandwidth and storage bandwidth even today. For example, in *Titan supercomputer* at the Oak Ridge Leadership Class Facility, there are five orders of magnitude difference between the aggregate computational bandwidth and the peak bandwidth to the parallel file system. Similar trends continue with the increasing influence of heterogenous architectures such as GPU-accelerated systems to support the demanding needs of AI-powered technologies. A widely employed solution is to translate the data analysis workflow from *post-hoc* (where data collection precedes the analysis) into *in situ* (where data collection supplements analysis) by using streaming data techniques. With in situ workflow, the collected performance data is made available for analysis as the data is recorded, whereby the analyst gets instant feedback on the experiments.

As in situ analysis workflow is an iterative process where frequent data updates and immediate analysis plays an integral part in deriving context-relevant knowledge through the observations made. Therefore, it is important to visualize key patterns and highlight changes from the temporal aspect as well. To meet this end, I incorporate progressive visualization techniques that provide *situational awareness* and *real-time monitoring* for analyzing network event traces from large-scale simulations (see Chapter 6). Overall, these techniques simplify the labor-intensive process within the core activities of in

situ analysis (i.e., data selection, preprocessing, analysis, and visualization) by allowing experts to realize desired or undesired performance behaviors immediately. I also develop interactive dashboards that summarize the execution timeline of deep-learning applications. These timelines function as a real-time monitoring tool and connect the HAC domains directly to the application execution.

1.3 Content Overview

This dissertation is organized into a 7 chapters covering the topic of performance visualization for HAC domains.

In Chapter 2, I provide background on HPC performance data collected from the HAC domains. From the Hardware domain, I discuss event traces collected from heterogeneous CPU-GPU architectures that provide real-time monitoring of device utilization and performance. From the Application domain, I cover the various representations of sampled profiles, namely flat profiles, Calling Context Trees (CCTs), and Call Graphs. From the Communication domain, I discuss how network traces are critical for parallel distributed computing and provide background on communication data collected from Parallel Discrete Event Simulations (PDES).

In Chapter 3, I introduce CallFlow, an interactive visual analytic tool that utilizes semantic operations to explore the CCTs, representing the caller-callee relationships. Modern scientific applications are built on top of rich frameworks and libraries that add many layers of abstraction, increasing the depth of call sites. Large CCTs end up having complicated caller-callee relationships, which makes identifying performance bottlenecks challenging. Therefore, I introduce *super graphs* that can aggregate and split a CCT at user-controllable levels of abstraction to gain insights into expensive call sites.

In Chapter 4, I extend CallFlow's functionality to handle an ensemble of CCTs and incorporate comparative analysis to study performance variability across multiple executions. Large-scale parallel applications require experts to conduct a variety of test runs by varying multiple configurations to identify optimal performance, resulting in large ensembles of performance profiles. To enable comparative analysis among the ensemble

members, I introduce *ensemble super graphs* that aggregate call paths across multiple CCTs and encode the differences between the CCTs effectively. By characterizing callpath profiles, CallFlow bridges various gaps by elucidating data with interactive and scalable visualizations that aid developers in accomplishing optimization goals.

In Chapter 5, I design a visual analytic framework to support data collection and visual analysis of performance metrics recorded as *event traces* at different time points of code execution. Grouping computing nodes based on the performance behaviors can reveal bottlenecks while studying the network traffic between computing nodes. I integrate automated analysis of multivariate performance metrics with coordinated communication views to reveal temporal behaviors using clustering algorithms and detect change points using time-series analysis.

In Chapter 6, I discuss strategies to reduce the high computational cost of analyzing the entire dataset, I extend the framework to support real-time analysis and monitor streaming data by adopting an in-situ data analysis workflow. Four online algorithms are introduced to present intermediate analysis results to the visualization views, which get updated real-time. These views are designed using progressive visualization techniques to enable different forms of data attribution like active user-engagement, perception, and comprehension of performance behaviors during analysis. By combining the network views with the temporal behavior views, the framework correlates multiple performance metrics and detect key communication patterns in various HPC network topologies.

In Chapter 7, I introduce DMV – Data Movement Visualized, a unified visual analytic framework to track data movement across CPU-GPU heterogeneous interfaces. DMV framework combines the performance data collected from HAC domains to understand performance trade-offs of data movement in large-scale heterogeneous applications, which has become increasingly vital with the expansive usage of GPUs for compute-intensive operations. This chapter explores critical data access patterns and anti-patterns within the CPU-GPU interface and demonstrates the effectiveness of the DMV framework by investigating the GEMM kernel under memory allocation strategies and the training of the GPT-2 model.

Chapter 2
Background

Understanding the performance of large-scale computations generally relies on two types of data: the application context in which these measurements were taken as *profiles*, e.g., execution runtime, on what processor, or at what callpaths; and measurements that reflect the performance of the hardware and communication domains as *event traces*, e.g., FLOP counts, cache misses, or network stalls.

2.1 Application domain - Sampled Profiles

Diagnosing the true causes of performance bottlenecks typically requires a simultaneous understanding of both performance metrics and the region of code attributed to performance. Knowing the source of the performance problem lets the code developers alter the application's high-level algorithms and tweak them for optimized performance.

Sampled profiles are time-based profiles that estimate the percentage of time an application spends in a particular section of code by interrupting the execution at regular time intervals. There exist various profiling tools [7,8,10,11] that record the application's *calling context* (*i.e.,* the call path from each profiled function to program main) and the associated performance metrics (*e.g.,* execution time and memory usage). Performance counters in these tools collect data on the fly as the application executes, allowing full-speed data collection and avoiding large profiling overheads. Statistically, the number of samples that fall within a given function represents a reasonable estimate of a code region's runtime. Various types of counters have been employed to predict the power, performance, and energy efficiency of computing systems [12].

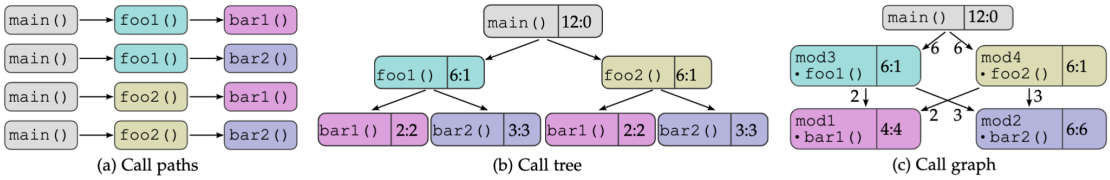


Figure 2.1: The call paths (a), call tree (b), and call graph (c) of the example program. In (a), each call path shows invocation instances of `bar1()` and `bar2()`, rooted at `main()`. Each node in the corresponding call tree contains three pieces of information: node name, inclusive metric, and exclusive 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.

2.1.1 CCTs and Call Graphs

Figure 2.1 (a) shows a sample of a profile comprising five call sites. In general, two kinds of information are useful for detecting bottlenecks: contextual information (*e.g.,* the call path, the process ID), and performance metrics (*e.g.,* execution time, memory usage). And commonly, two types of metrics are employed during analysis: *exclusive metrics* – the metrics belonging to a call site, and *inclusive runtime* – the recursively accumulated metric consumed by a call site and all its callees. A large-scale application would have thousands of call sites, making performance analysis challenging and time-consuming.

Flat profiles. The collected samples can be aggregated in various ways to simplify the analysis. For example, aggregating by lines in the code across call paths creates a *flat profile*. A flat profile can provide a good overview of the code regions that consume the most resources (*e.g.,* time) as they are usually displayed as sorted lists. However, they lack calling contexts that inform the control flow of general-purpose functions, such as communication or numerical routines.

Calling Context Trees. When the samples are aggregated by unique call paths, it results in a *calling context tree (CCT)* [13, 14]. 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*. Figure 2.1(a) shows the call paths of the program, and Figure 2.1(b) shows the corresponding CCT. Additionally, a node in a CCT is not limited to function invocations

only but can also represent loops, statements, and other code structures.

Call Graphs. Calling context trees can be aggregated in different ways to provide information more concisely. *Call graphs* [15] are created by merging CCT nodes with the same name (function name), *e.g.,* see Figure 2.1(c). By aggregating CCTs across function names, call graphs can significantly reduce the data's scale and complexity. Nevertheless, despite the simplification, call graphs of large, parallel programs can retain additional complexity that often prohibits the experts from unraveling the profiles.

For large-scale parallel applications, an analyst might aggregate samples across all threads or processes to create a global CCT or keep a forest of CCTs. Each node carries separate information for each thread/process. Therefore, despite being very informative, CCTs and call graphs pose practical analysis challenges. Modern applications are typically built on top of rich frameworks and libraries that provide many layers of abstraction, increasing call paths' depth, 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, simple filtering of such nodes could change the topology of the CCT, and standard approaches for analysis may not be efficient.

2.1.2 Visualization of CCTs and Call Graphs

Most analysis tools visualize CCTs as expandable trees [7, 8, 16, 17], using which the user can show and hide nodes as well as sort by attributes. 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 [18–24] are a popular approach for tree visualization, although dense matrix-based representations perform better for large-scale trees [25, 26]. 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. [27] embedded a histogram

onto the node to show imbalances between processes, and Nguyen et al. [28] encode runtime variation among processes to indicate the anomalies. Bohnet and Döllner [29] 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. [30] employ a node-link layout to represent the learned structural features of the CCT computed using an anomaly behavior detection model. Burch et al. [31] 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 [32, 33] have been effectively used to visualize hierarchical data by partitioning the screen space into bounding boxes that represent the tree structure. However, treemaps under-emphasize leaf nodes and make comparisons between subtrees difficult. Radio plots [34], 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 [35–37], the edges of standard node-link layouts usually represent only connectivity. Since the domain problem requires us to also encode the flow of resources, most notably time spent, employing a node-link visual representation is a premier choice in most performance tools. 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* [38–40]. 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. [41] represent the number of people participating in the mailing list of open-source software projects each month using a Sankey diagram.

Wongsuphasawat et al. [42] and Wang et al. [43] aggregate similar temporal events of patients' diseases and symptoms. To address scalability and interactivity issues, hpcviewer [7] employs additional strategies, such as automatic hot path extraction within a chosen hierarchy, flattening of calling structure, and zooming interactions to expand the relevant portion of the CCT.

2.1.3 Visual comparison of graphs and trees.

Comparison is often a key component of data analysis, especially to form and validate hypotheses [44, 45]. Multiple comparative visualization tools support comparison between two trees (*e.g.,* the original Unix `diff` program, or specialized tools such as TreeJuxtaposer [46], Mizbee [47], and Code flows [48]). The challenges in identifying the differences between two trees were summarized by Graham and Kennedy [49], and later generalized by Gleicher *et al.* [50], who provided a taxonomy of visualization designs to support comparison of two trees. In general, three approaches to comparative designs are *juxtaposition* (showing different objects separately) [51], *superposition* [52] (overlaying multiple objects), and *explicit encoding* [53,54] (using an alternate visual medium to show differences). A number of works have focused in comparing multiple trees, especially for phylogenetic trees [55–57]. These works support various analytical tasks using juxtaposed views combined with user interactions to highlight similarities and differences between two trees. However, these techniques cannot scale to a large number of trees and are designed to address the specific needs for phylogenetic trees. Recently, Barcodetree [58] suggest using a bar-code like visualization for comparing a large number of trees. However, their approach is primarily suited for stable and shallow trees, and cannot be used for call graphs, which are significantly deeper. Focusing on Sankey diagrams, Vosough *et al.* [59] provided a novel approach to visualizing uncertainty in flow diagrams by imposing gradient-based fading on the boundaries of the nodes. In contrast, my design maintains sharp bounds (node edges) but uses the color gradient to highlight the variability.

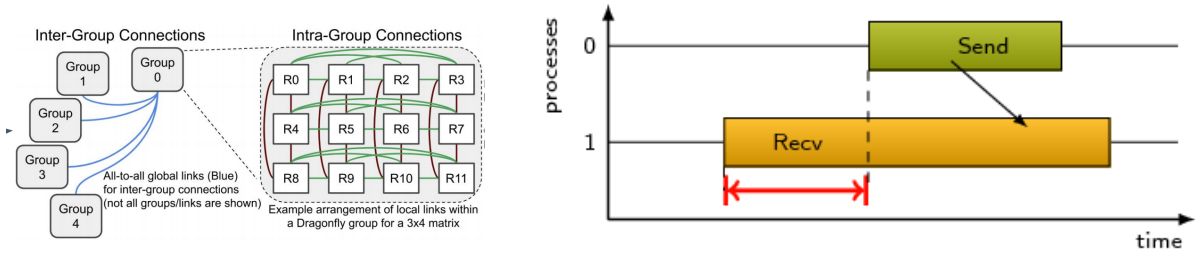
2.1.4 Visual exploration of ensembles of Call Graphs.

Williams *et al.* [60] amplified the importance of comparison, mainly for execution graphs (where each edge represents the dependency between tasks). The authors also proposed a visualization tool that compares only two execution graphs at any time by employing node-encodings to show the differences in the execution graphs. To compare several call graphs, Trevis [34] used a matrix view to visualize the pairwise similarity between graphs using a variety of distance measures. However, not only does the similarity matrix suffer from scalability issues, it often requires the user to perform pairwise comparisons [61] by matching individual runs. Nevertheless, the more-general task of exploring ensembles [62] of call graphs and study performance variability is not supported by existing tools. Instead, experts generally visualize the different call graphs individually to understand the general calling structure and use simple statistical measures to understand the distribution of runtimes.

2.2 Communication domain - Network Event Traces

Application domain of large-scale parallel computations is commonly decomposed into sub-problems to run on a cluster of interconnected computing nodes and often require a large amount of communication. Communication protocols such the Message Passing Interface (MPI) [63], is used for coordinate execute thousands of processes (or threads) that exchange intermediate results as the computation progresses. To enable fast inter-process communications, the compute nodes communicate in a distributed fashion via a high-bandwidth low-latency *network topology*, some famous topologies include fat-tree [64, 65], butterfly [66], high-radix dragonfly [67] (see Figure 2.2(a)) and high-dimensional torus [68]. Several research works [69, 70] have demonstrated that communication slowdown can lead to poor scalability for applications.

To analyze the behaviors and performance related to various network topologies, both hardware and communication domains must be studied and correlated to get a complete picture. Perfromance data is collected as *event traces*, which are a collection of time stamped records of various events occuring during the execution of an application [71].



(a) Dragonfly Network topology (b) MPI send vs MPI recv communication bottleneck

Figure 2.2: Example of a communication-related perfromance bottleneck: Process 1 waits in its receive cycle is blocked until the Process 0's send call leading to a performance bottleneck (as denoted by the right arrows).

Correct diagnosis of certain complex performance problems that arise on high end systems requires detailed event traces, which involves writing a time stamped record for each event, into a buffer or file for later analysis. Unfortunately, the collection of event traces presents scalability challenges: the act of measurement perturbs the target application; and the large volume of collected data increases the perturbation, and performance results require a large storage for further analysis. Tremendous amounts of trace data from event traces abstracted from HPC applications are produced at a high frequency, and must be managed and processed efficiently in order to conduct a definitive performance analysis.

Considering the scalability challenges of event traces, one might be tempted to avoid tracing entirely. Profiling, for example, provides summary information and therefore exhibits better scaling behavior. However, profiles are constrained to provide accurate diagnosis of runtime performance issues and not other issues. For example, as shown in Figure 2.2(b), "lateness of senders" is a common performance in a message-passing programs. This situation occurs when the receiving process waits at a blocking receive call waiting because the sending process hasn't yet reached the matching send call. While a profile would indeed to show that excessive time was being spent in receive operations, the data is not sufficient to distinguish between a late sender or some other root cause, such as network contention that caused the message to be received late. In contrast, an event trace captures the relative timing of events, and would show that the send operations started late and caused the receive operations to block. Event tracing is also useful for

showing the causality of events, enlist all interactions between processing elements (PEs), and reveal event patterns that accurately describe the performance problems and locations of possible optimization.

Understanding such detailed workload characteristics from HPC applications results in multiple metrics being recorded to distinguish the different kinds of performance bottlenecks. Although multiple approaches and tools have been developed to characterize event traces [72] and handle scalability [73], they do not handle the *multivariate* nature of the event traces (*i.e.,* a time-series for each recorded metric) obtained from the hardware and communication domains. Although, a number of research work have explored various time-series analysis methods and visualization techniques to allow users to explore both temporal behaviors and network traffics in a dashboard with multiple coordinated views. Simultaneous analysis of more than two variables is crucial for correlating the temporal behaviors and identifying similarities among the computing nodes to communication patterns.

2.2.1 Parallel Discrete Event Simulation (PDES)

To analyze the behaviors at the system and application level data, I collect the communicated data on each computing node using *Parallel Discrete-Event Simulation* (PDES) [74]. PDES is a cost-effective and useful tool for modeling by allowing the HPC developers to answer a variety of *"what-if"*, ranging from studies of complex physical phenomena to the designs of supercomputers. Rensselaer's Optimistic Simulation System (ROSS) [75] is an open-source PDES that simulates various network topologies at a flit-level detail to assist HPC network designers in design space exploration. First, I will provide the background information of PDES and its data characteristics.

To full understand the complexity of PDES, it is important to differentiate the parallel with the sequential discrete simulation. A sequential discrete event simulation model assumes the system being simulated only changes state at discrete points in simulated time upon the occurrence of an *event*. Each event is timestamped to indicate when a particular change occurs in the actual system. Whereas, in a parallel discrete event simulation (PDES) model, the events occur asynchronously at irregular time intervals.

This lack of assumption in the PDES model not only provides flexibility with the timing model but also preserves the fidelity of the simulation experiments. However, PDES is difficult because the sequencing constraints that dictate the order in which computations must be executed relative to each other is quite complex and highly data-dependent. Therefore, a domain expert depends heavily on performance metrics to determine if their simulations are being optimal.

To aid PDES performance analysis, the framework is designed for analyzing HPC applications that collect multivariate time-series and communication network data for analyzing performance from a PDES. The time-series data contains numerous performance metrics for each computing node at a specific sampling rate, which is also used for collecting the communication data among all the computing nodes. For the case studies, I use the ROSS Instrumentation layer [76] to collect time series and communication network data. The instrumentation layer provides three time-sampling modes, which allow users to sample simulation engine and model data (1) immediately after GVT computation, (2) at real-time sampling intervals, or (3) at virtual time sampling intervals. When sampling data from the simulation engine, the user can collect data for numerous metrics at different granularities (PE, KP, and LP).

PDES distributes a group of processes, called processing elements (PEs), that run across compute nodes. PEs communicate by exchanging time-stamped event messages that are processed to ensure the correct order of events (*i.e.,* the future event must not affect the past event) [74]. Ideally, these PEs should be distributed across the parallel processor so that (a) all processors remain busy doing useful work all of the time, and (b) inter-processor communications are minimized. In ROSS, where each PE is an MPI process, and message passing between PEs is performed asynchronously. ROSS employs an event rollback mechanism by developing unprocessed computations can be rolled back to their previous state. To track the rollback behavior, ROSS uses the global virtual time (GVT) for tracking the time in PDES, where the GVT is computed as the minimum of all unprocessed events across all PEs in the simulation [77].

Commonly, the metrics collected are either directly related to the *rollback behavior*

(*e.g.,* number of rollbacks) and the *communication among PEs, KPs, and LPs* (*e.g.,* network sends) which has an effect on rollback behavior. The specific metrics used in the visual analysis case studies are:

- **Primary Rollbacks**: The number of rollbacks on a KP caused by receiving an out-of-order event.
- **Secondary Rollbacks**: The number of rollbacks on a KP caused by an anti-message (*i.e.,* a cancellation message).
- **Virtual Time Difference**: The difference in time between a KP's local virtual clock and the current GVT. The value is typically positive, but it can be negative, indicating that the KP has not processed any events since the last GVT.
- **Network Sends**: Number of events sent by LPs over the network.
- **Network Receives**: Number of events received by LPs over the network.

2.2.2 Visualization of Network Traces from PDES

To analyze optimistic PDES, Ross et al. [76] introduced a visual analytics tool specialized for the ROSS optimistic PDES framework [75]. This tool is designed to analyze the simulator performance from multiple aspects, such as communication patterns and correlations between multiple performance metrics. However, only the aggregated values of the selected metrics are visualized for analyzing the change of the performance metrics over time, which is insufficient to depict the temporal behaviors of different entities. Because various temporal aspects affect the rollback behaviors and performance issues, more advanced temporal analysis of performance metrics is required to better understand the performance and behaviors of optimistic PDES.

Several researchers have studied techniques for temporal analysis. With an animation based approach, Sigovan et al. [78] used an animated scatterplot to analyze the temporal patterns in application execution. However, it is difficult to find the patterns of lengthy performance data with analysis methods that rely on animation. The Ravel visualization tool [72] visualizes execution traces and event histories of parallel applications using

logical time instead of physical time. Using logical time allows the application developers to analyze the execution sequence from the program's perspective. Muelder et al. [79] introduced the *behavior lines* for analyzing cloud computing performance. These lines show an overview of the behavioral similarity of multivariate time-varying performance data. Fujiwara et al. [80] designed a visual analytics system that integrated various advanced time-series analysis and unsupervised machine learning methods to overview and analyze the network behaviors of large-scale parallel applications.

However, these approaches only support analysis of either time-series or network data. Effective visual analytics methods and tools are lacking for analyzing both multivariate time-series and communication data as well as exploring their correlations. Instead, adapting the methods to summarize large-scale network and the temporal analysis methods for large-scale parallel applications can facilitate effective analysis and exploration of HPC datasets.

2.2.3 Streaming Data Visualization and Analytics

Dasgupta et al. [81] provided a comprehensive survey of streaming data visualization and its challenges. One of the major challenges with streaming data is to show important changes or meaningful patterns with a low cognitive load since available data is constantly updated [81, 82].

A common approach taken is the simplification of visual results (*e.g.,* aggregation). For example, Xu et al. [83] developed extended Marey's graph for monitoring assembly line performance. While Marey's graph is originally used for train schedules, it suffers from visual cluttering when the data gets updated frequently. To solve this issue, the extended Marey's graph in [83] emphasizes only the abnormal behaviors of the assembly line (*e.g.,* causing delays of assembly processes) by aggregating the other behaviors. Some researchers used dimensionality reduction (DR) methods to summarize changes in streaming multivariate data. For instance, Cheng et al. [84] used the multidimensional-scaling (MDS) for showing an overview of similarities between temporal behaviors in streamed data. In addition to this, they introduced the concept of sliding MDS, which visualizes temporal changes in the similarities between selected points as

line paths. To detect the rating fraud in e-commerce stores, Webga and Lu [85] used singular-value decomposition to show the dissimilarities of the rating streams. However, when streaming data has high volume and velocity (*i.e.,* frequent updates), the algorithms used in the works above could not work well due to the computational cost. To handle high volume and velocity streaming data, the completion time of algorithms used for visual analytics should be constant and/or less than the data-collection rate even when a total number of data points or features are increasing [86].

Recently, progressive visual analytics [87] is being actively studied [87–90] Producing useful results with a latency restriction is a common requirement. In fact, a few studies started to apply progressive visual analytics methods to handle streaming data. To achieve this, a few researchers started to apply incremental algorithms [91] (updating the result only based on new data points) or progressive algorithms [90] (providing the approximated result within a specified time constraint). For example, Pezzotti et al. [92] developed Approximated t-SNE (A-tSNE) as a progressive version of t-SNE [93] which is a commonly-used non-linear DR method. They showed the usage of A-tSNE for streaming data visualization in their case study. The work in [94] enhanced the existing incremental principal component analysis (PCA) [95] for streaming data visualization. In addition to the incremental calculation, their method maximizes the overlap of data points between previous and updated PCA results to keep the user's mental map.

2.3 Hardware domain - Performance metrics

Heterogeneous computing has been widely adopted to push the frontiers of computational sciences towards exascale computing as we hit the practical limits of Moore's law. In particular, Graphics Processing Units (GPUs) have become a key technology for accelerating the compute performance in supercomputers, including the US Department of Energy's ORNL's Summit, LLNL's Sierra, and the forthcoming exascale systems, LLNL's El Capitan. Since the execution model for GPUs differs from that for conventional processors, applications need to be rewritten to exploit GPU parallelism. In this pursuit, understanding the memory and how the devices are utilized play a critical role in helping

developers assess how well applications offload computation onto GPUs. In this work, I focused on studying the performance of Nvidia GPUs since they are widely employed in around 166 supercomputers in the world.

2.3.1 Data Movement in Heterogeneous Applications

Transfers between the host and device are the slowest link of data movement involved in heterogeneous computing, where the host (*i.e.,* CPU) orchestrates the computation by distributing the computation workload to devices (*i.e.,* CPU or GPU), executing parallel operations. Data movement across devices includes direct transfers to/from the GPU device, launching of GPU kernels, and between libraries supporting computations. With limited CPU–GPU and GPU–GPU bandwidth, careful data management is imperative to avoid unnecessary transfers. The major roadblock in the general adaptation of GPUs is the traditional copy-then-execute programming model as the onus of maintaining complex data structure, data transfer, and explicit data migration falls on the application developers. Handling data movement becomes vital for large-scale scientific applications due to multiple compute-heavy stencil computations and operations like convolution and pooling in deep learning (DL) applications. To achieve good scalability and performance, one must minimize unnecessary data movement. In the absence of appropriate tools to visualize and analyze data movement, developers often strive to realize opportunities for formulating strategies that reduce data movement overheads (e.g., page faults, false sharing, alternating CPU/GPU transfers).

Data locality (*i.e.,* how close data is to where it needs to be processed) across different levels of memory hierarchy has been a significant factor in affecting application performance [96,97]. Cicotti *et al.* [98] perform case studies to quantify the data movement across all the levels of the hardware memory (*e.g.,* node-level, system-level, center-level) and storage system. Their studies show that data movement-related costs can significantly increase as applications abstract heterogeneity. CUMAS [99] offers automatic overlapping of data transfers and kernel executions, but it focuses on scheduling multiple CUDA applications, rather than scheduling a single application's data transfers. dCUDA [100] is a runtime system, similar to MPI+CUDA, that overlaps computation with inter-node

communication in a multi-GPU environment. Ben-Nun *et al.* [101] introduce the *DaCe* framework to enable performance optimization by defining a workflow that enforces a separation between computation and data movement. Ivanov *et al.* [102] further uses the *DaCe* framework to study the effects of data movement when training DL transformers. Several frameworks like Kokkos [103, 104], RAJA [105], UMPIRE [106] have developed programming abstractions that migrate the data between memory spaces. However, abstraction-based frameworks still require significant programmer effort to write code in a manner that can fully exploit and optimize resource utilization.

With the introduction of UVM, developers were able to oversubscribe the GPU memory and make use of automatic page migration which abstracted data movement from the developer. Several research works have studied the effects of on-demand paging in CUDA's UVM [107–112]. Several tools have been developed to overcome the shortcomings of UVM management [106, 113–116]. Ausavarungnirun *et al.* [113] introduce a GPU memory manager, named *Mosaic* that presents a strategy to initiate a bulk transfer of all critical application data to limit the overheads involved with on-demand paging. SwapAdvisor [114], on the other hand, plans for what and when to swap precisely before execution to maximize computation and communication overlap. In contrast, *XUnified* [115] learns a model to guide the optimal use of unified memory of GPUs for various applications at runtime. Although increasing resource allocations (*e.g.,* more GPU devices) and/or enabling efficient memory management strategies can relieve programmers from manual data migration between CPUs and GPUs, validating the performance is still not trivial. Programmers face a multitude of tuning options, like which framework to use and compile *vs.* runtime optimizations to improve performance or/and manage memory effectively. In such scenarios, the lack of a performance visualization tool to explore GPU memory allocation strategies makes it challenging to identify the optimal one.

2.3.2 Data Movement and Memory Visualization

Several visual analytic tools already exist to study the data movement across memory hierarchy [117–119] and compute nodes [120, 121]. Memaxes [118] shows the overview of memory access samples through guided interactions to correlate the performance to code

behaviors. Mu *et al.* [117] propose a visualization to identify which nodes perform the memory accesses and detect an opportunity to reduce remote memory accesses. On the other hand, Sigovan *et al.* [120] map the data traffic across different network topologies for large-scale applications. Caches are widely employed to improve memory management by storing frequently accessed data. Tracking the cache activity and understanding the memory performance helps understand complex performance behaviors like memory fragmentation (*e.g.,* failure at reusing memory). Heapviz [122] uses a node-link diagram to visualize the references between objects to identify the relationship between objects and their memory addresses. Similarly, YACO [123] helps users analyze access patterns using multiple views to display statistics related to cache performance to find data that frequently enter and leave the cache. Moreta et al. [124] allow users to visualize the free memory blocks during runtime by displaying the memory accesses over time. But, these systems either focus on data movement in homogeneous architectures or only consider one of the HAC domains during the performance analysis.

Existing GPU performance analytic tools [125, 126] commonly employ simulators to capture and visualize a GPU architecture's dynamic behavior of an application run. However, the inferences deduced from such simulated systems do not always translate to large-scale applications that demand complicated data movement strategies to exploit the full potential. Existing tools like NVIDIA's Nsight systems [127], and AMD's ROC Profiler [128] are good at reporting raw performance metrics. However, they fail to provide insights or guide users toward *when* (*i.e.,* time of occurrence) and *where* (*i.e.,* data and code locality) performance slowdowns occur. Moreover, these tools only enable critical-path analysis using a static visualization, with no correlation to the context in which the events in the timeline occur. This significantly limits the user in connecting the performance across the HAC domains. Similar to *DMV*, Kousha *et al.* [129] design a profiler integrated with supporting visualization to track the communication in large-scale GPU clusters. Rather, *DMV* links the *execution timeline* with the communication (*i.e.,* data transfers) through animation allowing users to track individual data movement and understand their origin (using both CCT (application) and topology (hardware)).

Chapter 3
Visualization of Sampled Profiles from the Application Domain

Computation simulation codes are executed in parallel on large supercomputers with tens of thousands of processors. To achieve faster scientific breakthroughs via the 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, HPC experts are interested in identifying functions or code regions that are responsible for significant fractions of the overall execution time, *e.g.,* using *gprof* [15], as well as a *calling context* for each function invocation (obtained by walking up the call stack from the function), *e.g.,* using *HPCToolkit* [7] to generate a calling context tree (CCT). 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 profiling tools generate CCTs for the whole application, including the libraries it depends upon.

However, 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. When analyzing the CCTs, the functional affinities between nodes (both among siblings and across levels) carry a deeper semantic meaning that can be significantly informative on where the performance of simulation codes lie. 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 enables analyst to perform specific and well-defined queries, *e.g.,* extracting hot paths enabling an automated workflow to their analysis.

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, 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 [7, 8, 16, 17], node-link diagrams, treemaps [130], or icicle plots [131]. Although familiar to most users, expandable tree layouts do not scale with the size and depth of CCTs. In contrast, 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.

In this chapter, I introduce an open-source visual analytic tool named CallFlow[1] developed to support interactive exploration of large-scale CCTs and call graphs through a visual design process, including data and task abstractions. CallFlow was developed in an iterative manner, with the collaborators having access to the evolving prototypes, allowing us to refine the functionalities to best assist the experts in their inquiries. CallFlow presents the generic notion of *super graphs*, which can be used to represent sampled profiles at user-controllable levels of detail, including but not limited to CCTs and call graphs. Using the super graph notion, I describe new abstractions of the data using semantic *filtering*, *aggregation*, and *splitting* operations. CallFlow uses 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, execution time is used as the resource spent to encode the program execution along a given call stack. CallFlow presents the realization of the visual encoding as an open-source visualization tool, CallFlow (see Figure 3.1). The tool enables interactive exploration

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

of large-scale CCTs through focus+context visualization by expanding or contracting a super graph where desired. This work was done in collaboration with Huu Tan Nguyen and HPC experts from Lawrence Livermore National Laboratory.

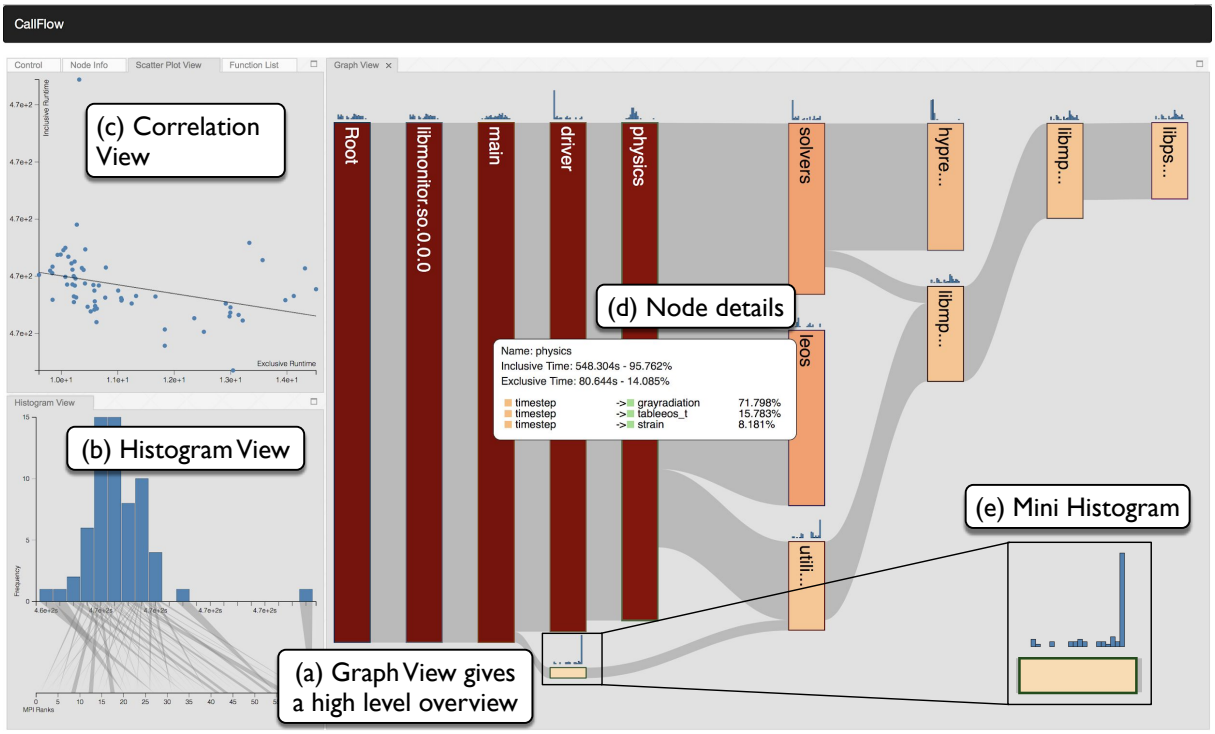


Figure 3.1: CallFlow presents dynamically interlinked visualizations to explore calling contexts of large-scale parallel applications. (a) The graph view visualizes the call graph using tailored Sankey diagrams at the desired level of detail. (b) The histogram view enables identifying runtime variations across processes, using histograms and shadow lines, which map histogram bins to process ids. (c) The correlation view allows finding the 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.

3.1 Domain Problem Characterization

The first phase of the design study involved developing knowledge about the domain problem. Over a period of about one year, several interviews were conducted with various HPC experts at LLNL, who are interested in improving the performance of large-scale parallel applications. The 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 [7, 132]. 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 1.1). 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 [27], 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 [7] 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.

3.2 Super Graphs and Graph 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 CCT or a call graph, with its root at the first call of the application, usually, the function `main`.

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 [30], 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 Figure 3.2).

Formally, I 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. The performance metrics for nodes are stored as rows in pandas [133] 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, I translate the domain-specific

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

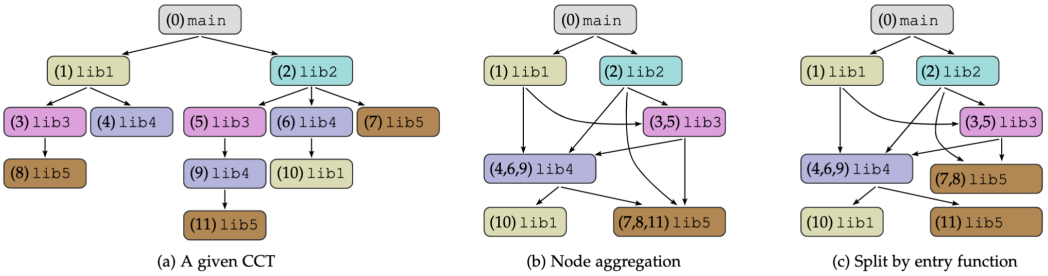


Figure 3.2: Node aggregation and splitting operations. (a) shows the original tree, labeled as "[function_name] `module_name`"; (b) shows aggregation operation where nodes from the same module are merged together. The aggregated super graph contains two *supernodes* corresponding to `lib1` to prevent a cycle; (c) shows a split by entry function operation where `lib5` supernode is split with respect to its entry functions, (7,8) vs. (11).

goals into more-specific graph operations: *filtering*, *aggregating*, and *splitting* of nodes.

3.2.1 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 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 the experiments, I 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.

3.2.2 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, I 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 $\mathbf{v}^{\text{s}}$.

It is to be noted 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, I 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 Figure 3.2(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, I employ a layout that does not support cycles, since their downsides outweigh the benefits.

3.2.3 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. Figure 3.2(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, I 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.

3.3 Visual Design of CallFlow

CallFlow is an interactive tool with three linked views: control flow view, histogram view, and correlation view, (see Figure 3.1 for an overview). Together, the three views support the queries of domain experts in an interactive manner.

3.3.1 Control Flow View

The control flow view presents an overview of the application's control during execution. I 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. I 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), I 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, the 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 Figure 3.1. 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, constant widths (horizontal spans) are used 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, placement of supernodes can be varied 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.

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, I 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, I follow the method described by Alemasoom et al. [40]. 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. I 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. Figure 3.3 demonstrates the value of such an optimization.

3.3.2 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 Figure 3.1). 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, I 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. Figure 3.1(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.

3.3.3 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 Figure 3.1), where each point in the scatter plot represents a process. If there is a correlation between two metrics, I expect processing elements to form clusters, whose size informs the extent of correlation. I 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.

3.3.4 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 Figure 3.1(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.

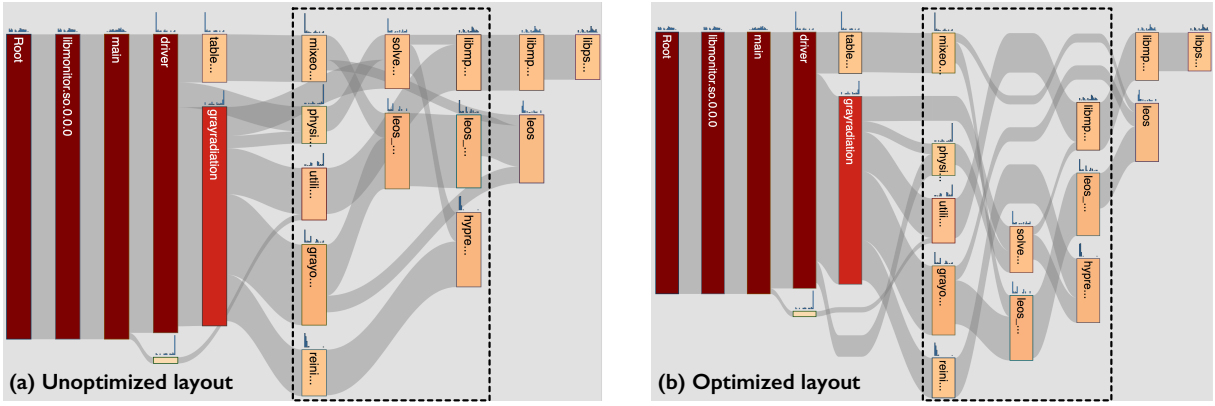


Figure 3.3: CallFlow uses a modified version of layout optimization presented by Alemasoom et al. [40]. With many overlapping edges in the unoptimized layout, (a) the connectivity is harder to decipher. Using the optimized layout (b), e.g., edges connecting `mixeo` and `leos`, and `util` and `libmpi` can be seen more clearly as compared to (a).

To mitigate this burden, I use a consistent naming scheme for supernodes to provide a consistent context in the transition. In particular, I concatenate the names of the (original) supernode and the (new) split supernodes separated by a hyphen. Figure 3.2(c) 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 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 Figure 3.4) 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.

Implementation Currently, CallFlow supports two data formats: HPCToolkit [7] and Caliper [11] formats. CallFlow is a web-based system implemented in Javascript. I used a client-server architecture to implement the system. The client is written using Node.js [134], which processes and serves the data to the client. The server performs many of the operations mentioned above which includes filtering, merging and spiting nodes. The client is responsible for performing the layout calculation for the graph view as well as handling user interactions.

3.4 Case Studies

Here, I 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 the 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 [7], Scalasca [16], and

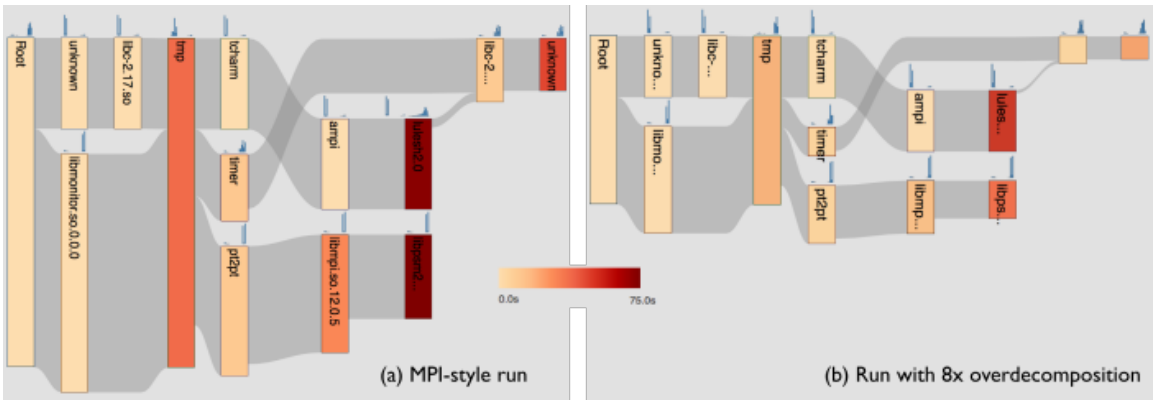


Figure 3.4: Understanding the impact of over-decomposition on LULESH. (a) MPI-style execution with 1 MPI rank per process: most time spent in LULESH internals, MPI/`libpsm`, AMPI internals (*tmp* bar), and C-library. (b) With over-decomposition, time spent in most modules reduces, but histograms of several modules show significant load imbalance.

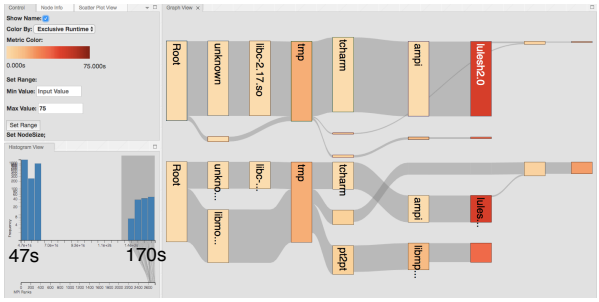


Figure 3.5: Analysis of load imbalance in LULESH by dividing the processes based on the time spent in LULESH internals. The splitting shows that processes with light load for LULESH internals (bottom view) spend significant time in MPI/`libpsm`, AMPI, and C-library, while remaining processes (top view) spend minimal time in these modules.

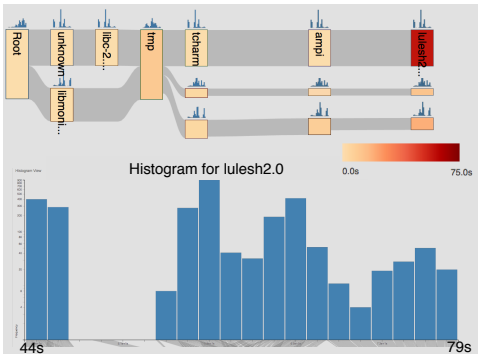


Figure 3.6: Impact of load balancing for LULESH (bar heights scaled as Figure 3.4): total runtime decreases and time spent in `libc` and MPI/`libpsm` is reduced. The load for LULESH internals is more evenly distributed across processes.

Vampir [10]. The following describes the studies and summarizes some of the informal feedback provided by the experts.

3.4.1 Study 1: Load Balancing of LULESH

LULESH [135] is a proxy application used for modeling the performance of large hydrodynamics simulations. LULESH solves a simple Sedov blast problem with analytic answers, but represents the numerical algorithms, data motion, and programming style

typical in scientific applications written in C/C++. It has been used to study the performance of different parallel programming models. 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, I use CallFlow to understand the performance of LULESH when implemented using Adaptive MPI (AMPI) [136] 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.

Figure 3.4(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. 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, I 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 Figure 3.4(a) and Figure 3.4(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

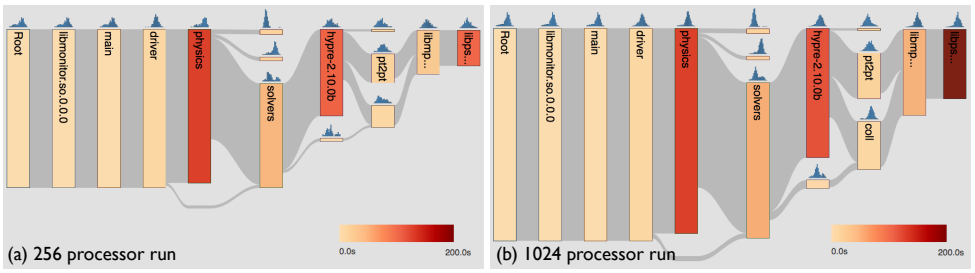


Figure 3.7: Comparison of Miranda execution on 256 and 1024 processors; height and color of a bar represent the inclusive and exclusive time spent in the module. These visualizations help identify the modules in which significant time is spent in the execution, and reveal that the increase in the exclusive time spent in the `libpsm` module is the primary reason for lower performance on 1024 processors.

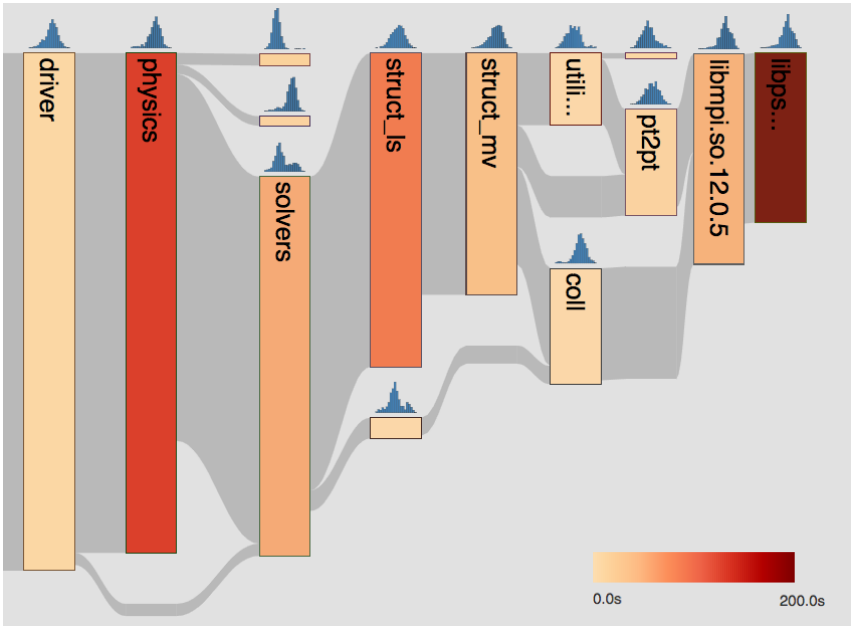


Figure 3.8: Splitting `Hypre` allows identifying its components responsible for performance-degrading communication.

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 Figure 3.4(b)) appear to be heavily skewed. To explore this further, I split the processes based on the time spent in LULESH (see Figure 3.5). 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, enabling load balancing in AMPI resulted in 30% better performance. The resulting profile (see Figure 3.6), 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.

3.4.2 Study 2: Scaling Performance of Miranda

Miranda [137] 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, 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. 3.7(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). Figure 3.7 also shows a juxtaposed comparison of profiles from 256 process and 1024 process executions. To facilitate such a comparison, I use CallFlow's feature to rescale the height of the root modules based on

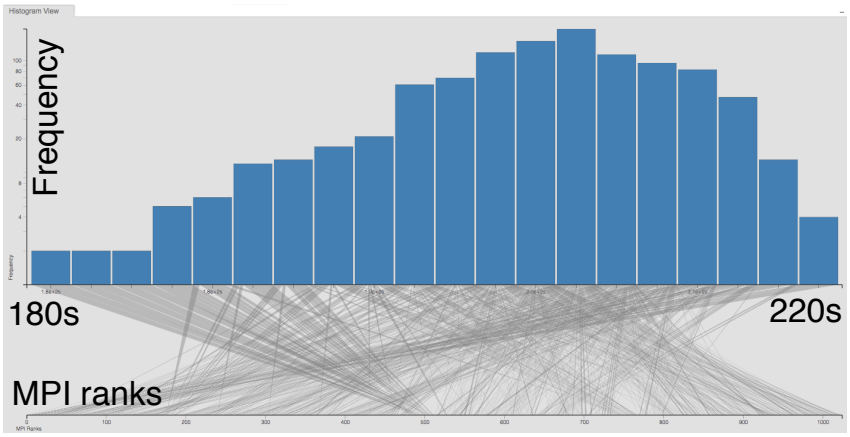


Figure 3.9: Histogram for time spent in the `libpsm` module: the distribution is normal and not heavily skewed; the bin-to-MPI rank connections reveal that lower-ranked MPI processes are more likely to have higher `libpsm` time.

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 Figure 3.8). 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, the collective calls made by the `struct_mv` module caused 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 Figure 3.9). I observed 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.

3.5 Potential Improvements

Despite the initial successful use cases and positive feedback from the domain experts, I identified several avenues for further research and development, which were incorporated in Chapter 4. First of all, CallFlow only allows analyzing a single dataset a time. However, simulation codes typically run under different conditions and it is necessary to compare the performance under these conditions. Enabling CallFlow to support multiple datasets and incorporate animation to transition between different datasets would be the next immediate target. Second, in both case studies, I noticed the domain experts were interested in comparing performance across MPI ranks (with LULESH dataset(see Subsection 3.4.1)) and across runs (with Miranda dataset (see Subsection 3.4.2)). Furthermore, experts were keen on expanding the tool to support exploration of the *performance variability*, a common occurrence in the HPC field.

3.6 Summary

In this chapter, I introduce CallFlow, 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 throughput. Catering specifically to the target data an users, CallFlow customizes and enhances the layout of Sankey diagrams, and uses multiple linked views to provide a holistic exploration of CCTs. Through interactive operations on the underlying graph, CallFlow provides both a high-level, system-oriented overview of CCTs and the ability to drill down to detailed information, and enabling analysis of large-scale CCTs more accessible and explorable.

Chapter 4
Ensemble Visualization of Sampled Profiles from Application Domain

Large-scale parallel applications demand a variety of deployment modes to identify performance bugs across versions of code or understand how different application parameters and/or initial conditions may affect the performance. Much importance lies in detecting the performance variations and enabling domain experts to choose the application parameters that correspond to the best-observed performance. For example, a simulation code may utilize computing resources inefficiently, possibly due to a suboptimal implementation, such as inefficient file I/O or unnecessary communication between processing elements.

Experts often conduct various 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. In particular, to identify optimal execution parameters and configurations, experts often conduct a variety of test runs for varying hardware configurations, system software choices, as well as application parameters, resulting in large ensembles of call graphs. Therefore, comparing the call graphs of several executions becomes essential to reveal the differences in both the performance metrics and the calling structure.

As discussed in Chapter 3, visualization of the collected performance metrics along with the *calling context tree* [13] helps gain insights into the execution of an application. However, existing tools [7–9, 16, 17, 37, 138], including CallFlow, either lack support for large ensembles of profiles or only provide primitive functionalities, *e.g.,* juxtaposed comparison [138], making an adequate evaluation of hundreds of profiles almost infeasible.

Alternative approaches, such as statistical summaries, may be used to analyze the recorded performance metrics. Still, the lack of interactive visual analytic support severely limits the experts' ability to understand new, unanticipated causes of bottlenecks.

In this chapter, I will discuss how I extended the visual design of CallFlow [138] to enable scalable performance analysis on large ensembles of profiles using the *ensemble super graph*. By combining data analysis and visualization, CallFlow's visual interface comprising multiple interactive linked views enables the comparison of the performance variability across ensembles. I introduce a new visual design, *ensemble-Sankey*, which combines the strengths of resource-flow (Sankey) and box-plot visualization techniques, facilitating the visualization of large ensembles of resource-flow graphs. An *ensemble view* is then built upon ensemble-Sankey design, provides a complete, high-level visualization of the ensemble. This view can be used to understand the overall distribution of the concerned profiles as well as identify the behavior of individual runs. I introduced new functionality to CallFlow that preserves this information, which can be revealed to the user upon request via the *supernode hierarchy view*, which augments the ensemble view by adding execution details within a selected node using icicle plots.

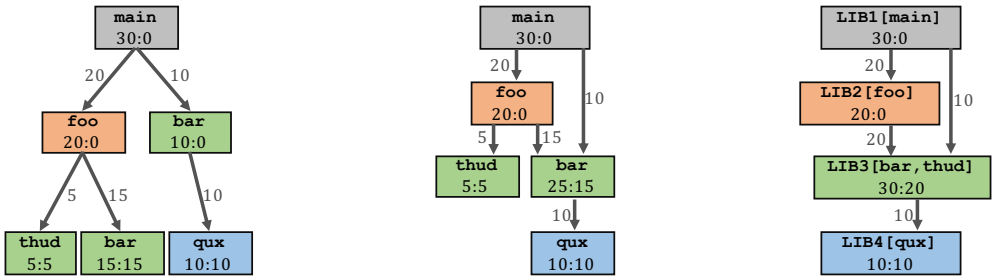
4.1 Need for studying call graph ensembles

Although call graphs provide an accurate and reduced representation of calling contexts, their visual analysis is nevertheless severely constrained by their scale, especially for large applications with calls to multiple libraries that may lead to hundreds of call sites. To simplify the visual analysis, CallFlow [138] introduced a *super graph*, created by aggregating the nodes of a call graph based on tool- or user-defined semantic attributes, *e.g.,* the library name, module name, or file name. *Supernodes* (the nodes in a super graph) represent collected call sites with a shared semantic attribute. For example, the supernode `lib3` in Figure 4.1 combines the call sites `bar` and `baz` that are part of the same library.

Although super graphs are useful for exploring large-scale call graphs, the semantic aggregation leads to loss of detailed information within a supernode, *e.g.,* the calling

context and the performance metrics of the functions within a given library.

Large profile collections that contain a lot of individual, but related, datasets with slightly different characteristics are called ensembles. The individual datasets being part of an ensemble are called the ensemble members. In Chapter 3, I visualized individual call graphs using Sankey diagrams [139] to indicate both the flow and distribution of resources of interest, *e.g.,* execution runtime, in a *single* call graph. CallFlow maps the chosen metric to Sankey nodes using color, which helps identify the resource consumption of modules and/or call sites. As in typical Sankey layouts, the edges encode the amount of resource being transferred as vertical thickness. The user may use *graph operations*, such as filtering and aggregation, to visualize call graph at desired details and explore the data interactively with respect to the metric of choice. However, such encodings and interactions are suitable only for the case of single call graphs and cannot be applied to ensembles. For example, a single color mapped to a Sankey node cannot capture the variation across ensembles and the intuition of resource transferred for edges is uniquely defined only for individual call graphs. Furthermore, comparison tasks cannot be



(a) Calling Context Tree

(b) Call Graph

(c) Super Graph

Figure 4.1: Performance profiles of applications are captured as their calling contexts, which represent call sites (functions) and their callees all the way up to `main`, along with performance metrics, *e.g.,* runtimes. CCTs (a) and call graphs (b) are simplifications of calling contexts created by aggregating call sites representing the same function. A super graph (c) introduces additional simplification through semantic aggregation, *e.g.,* based on the libraries they belong to. The nodes are labeled by the function/library name and inclusive:exclusive runtimes, and colors represent the nodes by library. Each edge is labeled with the amount of "resource" (exclusive runtime) flowing through.

performed realistically, since the user needs to perform pairwise comparison via juxtaposed visualization — a limitation that quickly becomes prohibitive.

Another use case for studying ensemble members is to detect performance variability among experiments or jobs submitted to large supercomputers. It is common to hear science users complain about performance variability within a single job from one time step to another and across equivalent jobs run over a period of time. This performance variability creates practical issues such as making performance debugging difficult and estimating the runtime of a job more challenging. The reasons for such runtime differences could range from network congestion to the variation in the input parameters set to the jobs. Therefore, domain experts were also interested in exploring an ensemble for performance variability and understand if the cause can be attributed to a library.

4.2 Domain Problem Characterization

Similar to Chapter 3, I tackle the characterization through close collaboration with domain experts. In particular, I focus on enabling *exploration of the performance variability captured across large ensembles of call graphs*. I identified five specific requirements for a new visualization tool.

4.2.1 Requirements for Exploring Call Graph Ensembles

R1. Compare two call graphs (*i.e.,* a *diff* view). In many cases, the users are interested in comparing the performance of two executions. Such a comparison must highlight faster/slower portions of a given execution to enable easy identification of performance bottlenecks.

R2. Visualize performance variability within several call graphs. Working with more than two graphs, users are interested in identifying the performance variability within a single function or a single module (in general, a single supernode). Whereas simpler statistics, such as mean, median, and variance, are easy to compute already, the users are instead looking for a visual depiction of the entire distribution to allow them to understand the overall trend as well as identify outliers.

R3. Accommodate ensembles of "reasonably different" call graphs. Whereas the

tool is not expected to summarize ensembles of profiles of different applications, the visual exploration must accommodate "reasonable differences" in the topology of call graphs of a single application, *e.g.,* differences in call paths due to application parameters or MPI implementations.

R4. Visualize additional fine-grained details on demand. Although a simplified visual layout is essential for the complete ensemble additional details, *e.g.,* about the module subtree and variance across processes, should be available to the user on demand.

R5. Interactive exploration and graph operations. The users desire the above functionality as well as the existing [138] graph filtering and splitting operations are interactive and scalable to ensembles of several hundred call graphs.

To understand the scope and requirements of this work in the context of the existing functionality of CallFlow, the new design solves a significantly different and more complex problem, in addition to while retaining the features of the previous version. Most prominently, I introduce two new analytic modes: *ensemble mode*, which visualizes the performance variation across multiple call graphs, and *diff mode*, which directly compares two call graphs or a selected call graph against the ensemble. To enable these analyses, I develop new visual encodings that are built on top of the Sankey layouts, which I call *ensemble Sankey*. Furthermore, the tool is also capable of interactively revealing the additional details on demand using the *supernode hierarchy view*. Finally, adding to the histogram view, which allows exploration of performance slowdown/speedups among MPI ranks with the physical domain, I also helps explore the variation in the observed runtime distribution using boxplots views.

4.2.2 Identification of Targets and Actions

Comparison targets are data entities integral to the comparison task, *i.e.,* the specific data elements being compared. Here, I identify four comparison targets. Whereas the first three targets are *explicit*, **T4** is an *implicit* target because it is not known from the data itself, but rather is based on analysis of the ensemble [140]. Here, comparing explicit targets is challenging because the comparison with respect to call graphs is difficult, and

the ensembles pose scalability issues. On the other hand, the challenge with implicit targets is that they tackle unknown elements that requires user knowledge to interpret the differences.

T1. Calling contexts are an overarching target for the comparative visualization. Any optimization efforts usually focus on identifying parts of the applications that must be improved, for which, understanding the entire calling context of the application is essential to visualize. Furthermore, different execution modes (*e.g.,* MPI libraries and application parameters) may lead to slight differences in the calling contexts. Understanding such differences is also important to reason about the variability in the performance within ensembles.

T2. Performance variability across runs. Production codes with high run-to-run variability may cause unpredictable slowdowns and diminish the reproducibility of successful experiments. Highlighting such variability, at both module (summarized) and call site (detailed) levels, is one of the most critical comparison targets in this work.

T3. Performance variability across MPI ranks. For MPI-enabled applications, the overall performance degrades when the runtime is less uniform across resources [141], *e.g.,* due to load imbalance. Identifying heavily over- or under-utilized resources from the runtime distribution is important to design codes that can achieve peak performance.

T4. Execution parameters. In many cases, an application's performance may be correlated with certain execution parameters [142]. Therefore, identifying the parameters with more significance can help understand explored multiple optimization strategies simultaneously.

Comparison actions are visualization tasks needed to understand the relationships between and within comparison targets. Formulation of actions influences the visual encoding and the user interactions that link individual visual components. Visualization of ensembles of call graphs broadly require six actions.

A1. Compare calling contexts across runs. Due to potential differences in calling contexts of the different executions (**T1**), it is essential to highlight any differences, *i.e.,*

missing nodes/edges in the graph.

A2. Compare performance variability across runs. Displaying the aggregated performance from individual executions is not a scalable approach considering the effort from the user, and the corresponding visualization complexity. Instead, the recorded performance (**T1**) and calling contexts (**T2**) must be summarized not only within a single execution, but also across multiple call graphs.

A3. Compare performance variability across MPI ranks. Summary statistics (*e.g.,* mean runtime) across MPI ranks (**T3**) is not sufficient to analyze runtime distribution across resources because the distribution is not expected to be normal. To aid the exploration, complete distributions are essential to visualize.

A4. Compare call graphs across levels of detail. Although comparisons across super graphs (semantically aggregated call graphs) is useful, when experts typically identify problem areas, they are interested in looking at selected regions in more detail. Therefore, it is important to manage fine-level details and visualize them upon request.

A5. Compare two call graphs. Differences between two call graphs can exist structurally (**T1**), as runtime variations (**T2, T3**), and/or in execution parameters (**T4**). Simple, visual representations of such differences is of value to the user.

A6. Compare a selected run with an ensemble. Comparison tasks often require a baseline to compare against. For all of the targets (**T1**–**T4**), it is important to compare a selected run with the ensemble behavior. An important constraint to consider is that the constructed baseline must match the expert's understanding of the overall calling context, despite minor differences in the individual ensemble members.

4.3 Ensemble SuperGraph

4.3.1 Data Representation

To efficiently represent the data, the given sampled profiles are converted into *GraphFrames* using Hatchet [143], an open-source profile analysis tool. A GraphFrame ($\mathcal{G}_{\text{cfg}}$) is a Hatchet construct that consists of two data structures: a *directed acyclic graph* ($\mathcal{G}$) that represents the CCT or call graph, and a Pandas [144] *DataFrame*

($\mathcal{D}$) that stores the associated performance metrics. Hatchet represents the collected performance metrics into individual columns of $\mathcal{D}$ and establishes a consistent indexing scheme to link the call site in $\mathcal{G}$ to its performance data in $\mathcal{D}$, which not only enables combining data operations with graph operations, but also improves the scalability of various analytic tasks.

Each call site in $\mathcal{G}$ is associated with semantic information from the source code, *e.g.,* load module, file name, line number, obtained either automatically via the profiler or through explicit user annotations. This semantic data is extracted from $\mathcal{G}$ and stored as additional columns in $\mathcal{D}$ for fast access. Furthermore, the performance metrics recorded on each call site typically contains metrics from multiple processing units (*e.g.,* MPI ranks). Fast access to the data frame is facilitated through hierarchically multi-indexing $\mathcal{D}$, where the indices are *module*, *function name*, and *MPI rank*, respectively. This multi-index representation also facilitates easy integration of higher-order statistical summaries (*e.g.,* variations across runs, and variations among ranks) over different levels of details (*e.g.,* modules, call sites, or anywhere in between).

4.3.2 Construction of Ensemble GraphFrames

A GraphFrame ($\mathcal{G}_{\text{cfg}}$) is the key data source for CallFlow. In the case of ensembles of profiles, each representing the performance of an application under different executions, the different $\mathbf{G}_i$'s are stored individually. Generally, meta information about the different executions are available, *e.g.,* machine architecture and/or environment (*i.e.,* the number of cores or processes, maximum available bandwidth, and load capacity of the supercomputer), or application parameters. These *execution parameters* are also stored as additional indexed columns in the corresponding $\mathcal{D}_i$. In order to support the comparison target and actions described above, it is required to have a consistent and scalable approach to store, access, and update $\mathbf{G}_i$'s.

To this end, I describe how to concisely represent the entire ensemble as a single *ensemble GraphFrame*. Given user-defined options as configuration files, two unification steps are performed during preprocessing.

Step 1. First, I ***unify*** **all dataframes** ($\mathcal{D}_i$'s) and concatenate them into an *ensemble*

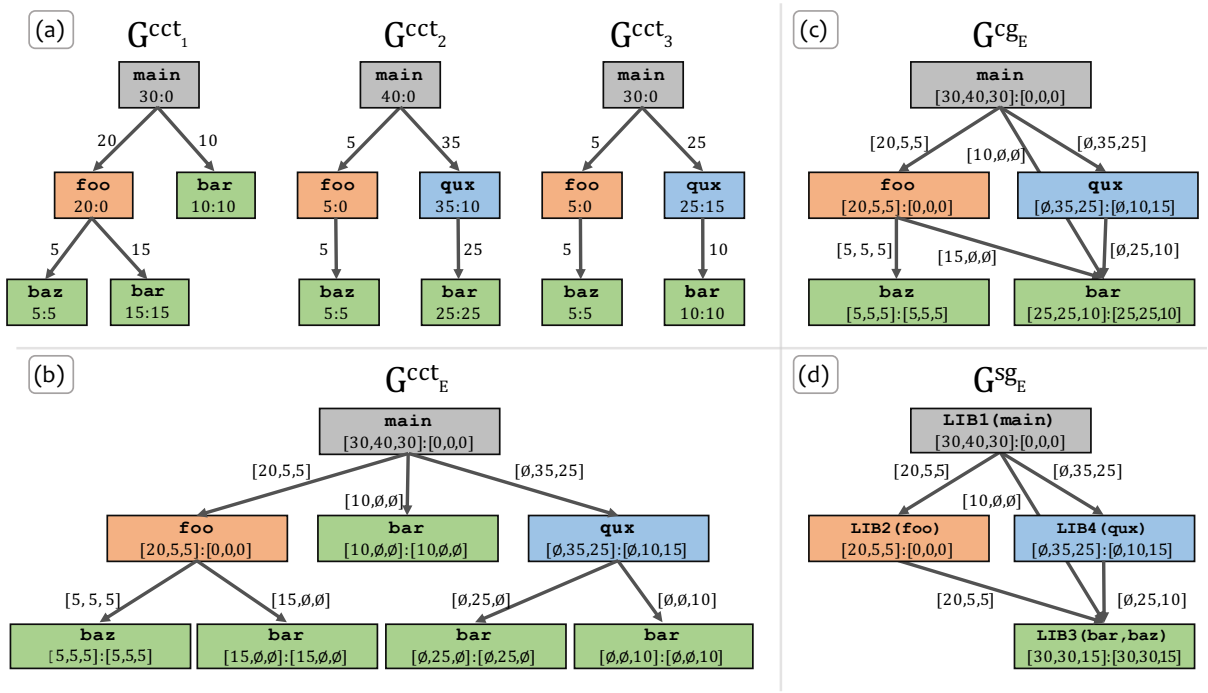


Figure 4.2: Construction of *ensemble super graph* from 3 given graph frames shown in (a). (b) First, the ensemble CCT, is constructed to include all unique calling contexts across each CCT. The performance metrics for corresponding call sites (nodes in the graph) are concatenated into associated vectors, where missing nodes in any given CCT are denoted with ∅ in the vector. (c) Next, ensemble CCT is converted into an ensemble call graph, by grouping call sites representing the same function into a single node. The associated metric vectors are element-wise added. (d) Finally, semantic information, *e.g.,* library names, are used to group call sites to create the ensemble super graph.

dataframe ($\mathcal{D}_E$) by performing a unify operation, where we add column *"runName"* to tag the call sites belonging to individual runs and aggregate the required information.

The *"runName"* column becomes the primary index of the multi-indexed hierarchy of $\mathcal{D}_E$. However, for several hundred large dataframes, $\mathcal{D}_E$ can become prohibitively large, incurring large cost for data operations. To improve the processing time for data operations, I abstract the information stored in $\mathcal{D}_E$ at two levels of detail (**A4**), with respect to *module* and *name*, where the data is grouped based on the module/library and the name of the call site, respectively. Both levels of detail are stored as separate files using the HDF5 [145] data model, a community standard format used for storing hierarchical-structured datasets. HDF5 allows attaching semantic information (*i.e.,* library and call site names) as *"attributes"* that group data together for faster lookup.

Step 2. Next, I ***unify all calling contexts*** ($\mathcal{G}_i$'s) to construct an *ensemble CCT* ($\mathcal{G}^{\text{cct}_\text{E}}$). This unify operation aggregates each call site's performance metrics and 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 contexts. The corresponding performance metrics are also aggregated and stored as vectors. In Figure 4.2(a), I demonstrate the unify operation using three "reasonably different" $\mathcal{G}_i$'s (**R3**) comprising 5 call sites belonging to 4 libraries. Although the are minor differences in the graph (*e.g.,* missing nodes), the unify operation accounts for such inconsistencies by assigning a null value ($\varnothing$) to the missing nodes in any $\mathcal{G}_i$. For example, whereas the calling contexts of `thud` remain consistent across the three CCTs, the contexts of `bar` and `qux` differ across the runs. The resulting $\mathcal{G}^{\text{cct}_\text{E}}$ (see Figure 4.2(b)) represents a super set of the given $\mathcal{G}_i$'s, preserves all calling contexts, and merges the nodes with identical contexts.

Next, $\mathcal{G}^{\text{cct}_\text{E}}$ is converted into an *ensemble call graph*, ($\mathcal{G}^{\text{cg}_\text{E}}$), according to the usual definition, *i.e.,* call sites with the same function name are merged (see Figure 4.2(c)). The associated vectors are element-wise added to summarize the performance runtime on each call site. Finally, if needed, the call sites are filtered by inclusive/exclusive runtime using user-defined thresholds, and grouped with the available semantic information, to construct the *ensemble super graph*, ($\mathcal{G}^{\text{sg}_\text{E}}$), which is a module-level super graph. Finally, I combine $\mathcal{D}_\text{E}$ with dgse to construct the *ensemble GraphFrame*, ($\mathbf{G}^{\text{sg}_\text{E}}$). The ensemble GraphFrame captures all equivalence relationships among call graphs using nodes and the edges store the subtle differences in the performance and calling structure. The aggregated $\mathcal{G}^{\text{sg}_\text{E}}$ can now be used as the *baseline* super graph to compare against the ensemble (**A3**).

4.4 Visual Analytic Design

To build an effective tool supporting visual comparisons, we closely follow the design considerations proposed by Gleicher [140]. Here, I detail the approach to the design of CallFlow to support the domain requirements. The improved visual interface of CallFlow (see Figure 4.3) comprises of five inter-linked views and three analytic modes.

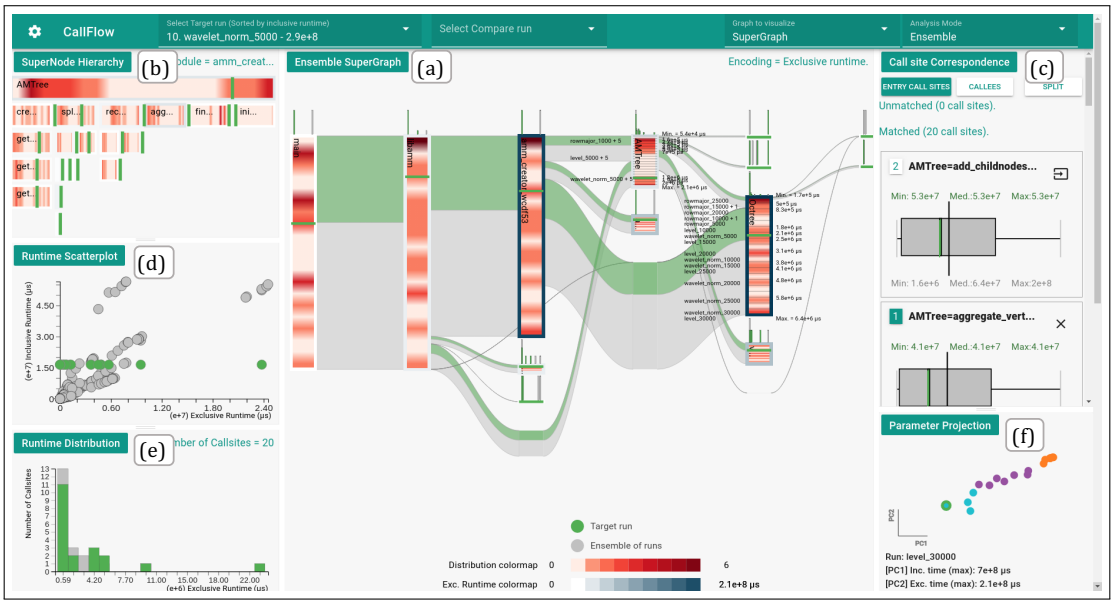


Figure 4.3: CallFlow enables scalable visual analytics of ensembles of call graphs using a novel visual design, *ensemble-Sankey* (a), and several linked views (b–f), which provide additional, fine-detail information (*e.g.,* hierarchy within a node of the Sankey), as well as statistical descriptions of the data (*e.g.,* run time distributions). Through interactions with visual elements as well as UI-based options, the tool allows thorough exploration of the ensemble data, serving a wide variety of application-specific requirements.

4.4.1 Ensemble-Sankey: The Ensemble SuperGraph View

In this work, I expand on the visual design of CallFlow to support visualization of ensembles of super graphs, and create a novel visual encoding – the *ensemble-Sankey*. Our decision to preserve and enhance the Sankey layout emanates from the effectiveness of the existing design. In particular, the resource flow conveyed by Sankey layout provides an excellent way to represent calling contexts and associated performance metrics, as it matches experts' intuition about the internal representation of the application flow. Furthermore, perceptual simplicity is vital for the comparison task, as the user has to build the proficiency to "spot the differences" in an ensemble of runs. However, using the existing encoding of nodes and edges are not directly usable for the ensemble, since only a single graph may be visualized. Instead, I develop new node and edge encodings to concisely represent an ensemble in our *ensemble-Sankey* design.

Ensemble Nodes. In the (standard) Sankey diagram, each node in the graph is visualized as a rectangular bar, whose height corresponds to the "resource of interest". In

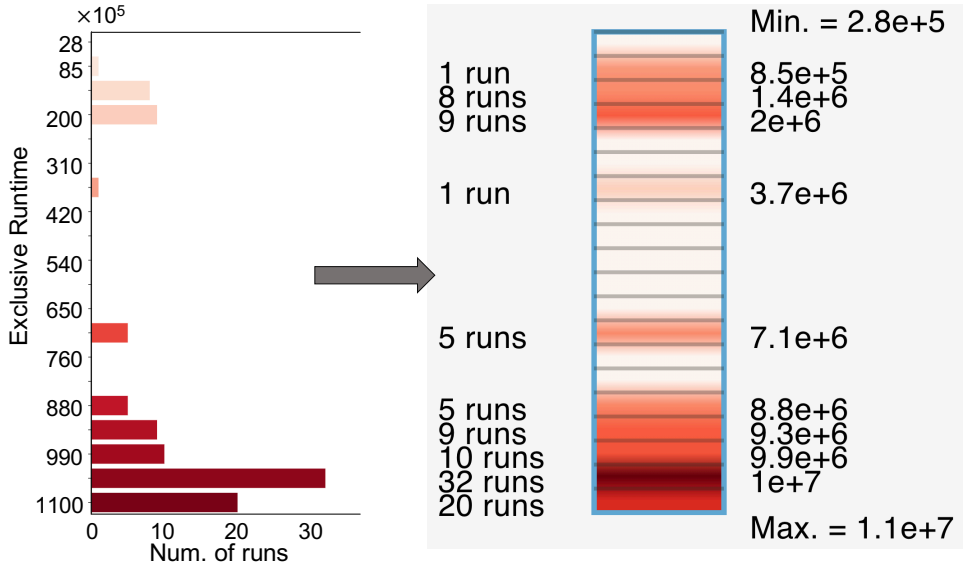


Figure 4.4: Given an ensemble of super nodes, the histogram of the performance metric (*e.g.,* runtime) is mapped to a Sankey node as an ensemble gradient.

the case of ensembles, however, there are many graphs, each with a different resource flow. Given the requirements of visualizing the variability within the ensemble members (**R2**) and maintaining the same interpretation of "resource flow" (for **R1** and **R4**), the height of the rectangular bar is used to represent the full range of values. In particular, the height of a given supernode in the ensemble-Sankey is scaled proportional to the *maximum* inclusive metric across all executions. This encoding preserves the notion of resource flow from one module in the call graph to another and provides visual elements of consistent shape and size that encode the "ensemble supernodes" such that the "standard supernodes" of any given ensemble members are simply subsets. The width of the rectangle is computed in the same manner as done by CallFlow [138].

Ensemble Gradients. In order to visualize the distribution of a chosen metric across the ensemble, we use the color channel to encode the distribution and overlay on the node. We compute the histogram of the chosen metric and map it vertically to the height of the supernode (as shown in Figure 4.4). Since the true histogram may contain sharp spikes (we do not necessarily expect to find a smooth distribution), we use smooth gradient fading to ensure all peaks are highlighted. Although we could directly compute a density

estimate (*e.g.,* using KDE) instead of a histogram to generate a smoother distribution, such techniques introduce additional parameters (*i.e.,* kernel width), which are typically harder to interpret and may miss features regardless due to an unsuitable choice. Instead, we use histograms (with a customizable bin count) and use linear gradients to smooth the distribution. In this way, the resulting *ensemble gradient* shows the full distribution of the selected metric across runs. The default choice of color for the distribution is a single-hue white–red colormap. The colors are mapped consistently across all nodes to allow evaluation of distributions not just within a single supernode, but also across supernodes (**A2**).

Borders and Text Guides. With the inclusive metric mapped to the height and gradient of the rectangle, we are also interested in highlighting the exclusive metric. To this end, we draw colormapped border on the node. As above, we use the *maximum* exclusive metric across all executions, as it can help identify exceptionally slow runs. Although the default border coloring is the (maximum) exclusive metric, the user can interactively avail other options, such as color by (maximum) inclusive metric. Additional runtime information can be revealed by toggling the *text-guides*, which mark the bins of the histogram and also display the minimum and maximum runtimes of the executions among the ensemble. The user can further refine the exploration by clicking on the text-guides, which updates the visualization to the subset of the ensemble in the corresponding bin.

Ensemble Edges. In the (standard) Sankey layout, the edges encode the flow of the resource; the vertical thickness of edges is proportional to the inclusive metric consumed by the target node (*i.e.,* the resource being transferred to a target node). Whereas this metaphor is straightforward to interpret for a single super graph, it cannot be enforced for the superedges of an ensemble-Sankey because of the aggregation (max) performed for each supernode. In particular, since the maximum values of child nodes (for a given node) may be from different runs, the sum of all outgoing resources is not necessarily equal to the height of the node (minus the exclusive time). As such, there is no unique mapping that can be defined for the height of a superedge. For example, consider Figure 4.2(d), where the aggregated metrics for each supernode is the maximum of the corresponding vector,

implying the values of the nodes corresponding to `LIB1`, `LIB2`, LIB3, and `LIB4` to be 40, 20, 30, and 35, respectively. However, the aggregated (max) metrics for the outgoing edges from `LIB1` are 20, 10, and 35, respectively, which sum to 65. Such scenarios are common in profiles with high performance variability from different libraries. To present a consistent encoding and interpretation of superedges, we instead scale the two ends of a superedge proportional to the respective nodes, thereby, smoothly varying the height of the superedge (from left to right). In addition to a neat visualization, this encoding facilitates visualizing a superedge with respect to both its source and destination.

4.4.2 Supernode Hierarchy View

When scanning the ensemble gradients, the user is typically limited to exploring the ensemble distribution for the revealed supernodes only. Although graph operations, such as splitting, can reveal additional interesting nodes but the overall context is changed (since the split graph is different). For example, splitting a supernode by its entry functions would reveal all the entry functions belonging to a supernode, but if there are multiple entry functions, the Sankey layout may change significantly. Through discussions with potential users, we instead choose to visualize the additional details separately, in particular, as a *supernode hierarchy* using an icicle plot, similar to the flamegraph visualizations [131, 146]. Although a hybrid approach like nodetrix [147], which presents the global structure using node-link diagram and presents the internal structure using a matrix representation, could be adopted, users considered the adjacency matrix to be difficult to interpret for topological operations (i.e., splitting), because the adjacency matrix does not present the continuity perceived through sequence of links (in Sankey diagram) and hierarchy (in icicle plots).

Icicle plot places the call sites of the selected library based on their depth inside the supernode hierarchy from top to bottom. Each call site in the supernode hierarchy is visualized as horizontal rectangular bars. As with the ensemble-Sankey, ensemble gradients and runtime borders are used to encode the ensemble distribution and the runtime distribution for the call site, respectively. Additionally, when a target run is selected, the supernode hierarchy view also enables the comparison of multiple calling

contexts (**A1**), as the nodes with ∅ have no ensemble gradients that fill the rectangular bar, allowing the user to identify the missing call sites easily. Furthermore, call sites of interest from the supernode hierarchy can be selected by clicking to reveal all call sites in the module's context. Revealing a call site's calling context in the ensemble-Sankey helps compare the runtimes to identify performance slowdowns among call sites.

4.4.3 Complementary Views

Call site correspondence view (see Figure 4.3(c)) lets us focus on process-level distribution, and also scan call site information based on data and graph properties. To explore variations across multiple target metrics, we utilize a boxplot to study the different runtime ranges occupied by the observed runtime distribution for each call site (**A3**). *Boxplots* use quartiles of the distribution to indicate the spread of data, and can also reveal outliers. In this view, we enumerate the call sites that are not present in the ensemble view, and highlight the median, the interquartile range (IQR = Q3−Q1), as well as outliers (above and below 1.5× the IQR). Additional information is also provided as text labels (*e.g.,* minimum, median, and maximum runtime).

Metric correlation view. In order to compare inclusive and exclusive metrics for a given call site, we produce a scatterplot that captures the correlation between the two (Figure 4.3(c)). For ensembles, each "dot" in the scatter plot represents a callsite for a single run, making it possible to study such correlations across ensemble members, *e.g.,* by comparing a target run to the ensemble, as shown in the figure. The scatter plot itself is also interactive, as hovering over a dot highlights call site by their names for the user to compare the runtime metrics (**A1**).

Runtime distribution view. To assess the distribution of runtime, we show histograms for the chosen metric (inclusive or exclusive) for a selected supernode in three modes: (1) *call site mode*, which shows the distribution of runtime metric with respect to individual call sites (*i.e.,* the vertical axis is the number of call sites with a given range (bin) of runtime, as shown in Figure 4.3(e)); (2) *call graph mode*, which shows the distribution with respect to the call graphs (*i.e.,* the vertical axis is the number of call graphs with

a given range (bin) of runtime); and (3) *MPI rank mode*, which shows the distribution with respect to the MPI ranks (*i.e.,* the vertical axis is the number of MPI ranks with a given range (bin) of runtime). These distributions are important to explore for a complete understanding of the chosen supernode in an ensemble, with or without a target run.

Parameter projection view (Figure 4.3(f)) is an experimental feature in tool that aims to explore correlations between execution parameters by projecting them onto a 2-D space using multidimensional scaling [148].

4.4.4 Visual Analytic Modes

Ensemble summary mode is the default analytic mode when studying an ensemble of runs.

Target-ensemble comparison mode is triggered when the user selects a particular execution from the ensemble to study in detail from "Select Target run", which lists all ensemble members. This operation allows the user to compare a selected run's performance with the ensemble (**A6**) across all 6 views. Then, the visual elements belonging to the selected target run are revealed in green throughout the system (see Figure 4.3) – for consistent context, the elements corresponding to the ensemble are shown in gray. In particular, the *target superedges* are overlaid on top of the ensemble edges to compare the proportion of inclusive runtime that calling contexts (**T1**). The *target-lines* (*i.e.,* green colored lines) place the selected run in the distribution for each supernodes (**T2**). Boxplots are revealed on top of the ensemble boxplot to compare the process runtimes of the target run vs. ensemble.

Target-target difference mode. During comparative analysis, the user might find two executions that exhibit a variation in the ensemble distribution, or appear an outlier from the projection view. To allow the user study the differences between two supergraphs, we compute a *diff call graph* that subtracts the mean runtime across the corresponding supernodes. The resulting *diff view* is visualized as a Sankey diagram and is colored with a green-red colormap, where hues of red and green colors represent negative and positive values, or performance slowdown and speedup, respectively.

4.5 Case Studies

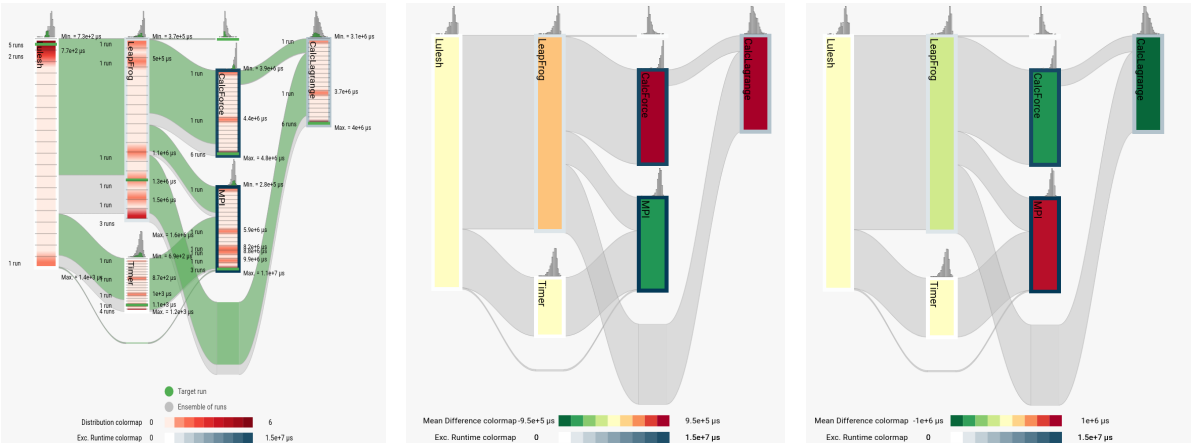
To demonstrate and validate the visual analytics design, I present two case studies illustrating its value in HPC.

4.5.1 Study 1: Performance Variability due to Application Parameters

First, I study the performance of a single-process C++ library, AMM [149], which creates adaptive representations on-the-fly for streaming volumetric data. AMM is currently under development and the code developers are keenly interested in improving its performance for various application parameters. Among others, a key parameter that affects the performance is the type of data stream (*i.e.,* ordering of data, such as row-major and wavelet transform subband order) [150]. With the general goal of obtaining insights into the performance variability of AMM as well as identifying any potential optimization opportunities, experts explore 18 profiles (captured through Caliper [11]) that represent three different data streams with other parametric variations (*e.g.,* data sizes). Figure 4.3 shows the CallFlow visualization for the given ensemble.

Although AMM does not use MPI and therefore the corresponding profiles cannot fully leverage the functionality of CallFlow, they make an excellent case study due to high performance variability. Indeed, the ensemble view (Figure 4.3) highlights the high variability in the distributions of runtimes across different run modes (**A2**). AMM makes use of several recursive functions, and despite different recursion depths, the call graphs of the different runs are very similar. Given the application parameters explored here, code developers expected high variation across runs but very similar patterns of variations across modules and call sites. Instead, to the surprise of code developers, the distribution patterns (shown by ensemble gradients) vary significantly across different modules (**A1**). This behavior hints at problems with recursive functions (*e.g.,* potential memory leaks) — a potential improvement in the code that developers are currently exploring.

Through more detailed analysis of the ensemble, *e.g.,* using the supernode hierarchy view and call site correspondence view, it was noted that several `get_*` functions in modules `AMTree` and `Octree` consume significant run time (**A3**). Through



(a) Target-Ensemble view (b) Diff view (64-cores − 27-cores) (c) Diff view (216-core − 125-cores)

Figure 4.5: Study of LULESH's performance profiles. (a) The *ensemble-gradients* and *target-guides* highlight the target run's runtime in contrast with the ensemble (the target run is 216-cores). (b) and (c) visualize pairwise differences of the runtimes between 2 runs using a green-red colormap. Hues of red color highlight regions of code that cause a performance slowdown between the two profiles (*e.g.,* CalcForce and CalcLagrange in (b), and MPI in (c)), and green hues highlight the performance speedup (*e.g.,* MPI in (b), and CalcForce and CalcLagrange in (c)).

discussions with developers, it was learnt that these functions make use of the default `std::unordered_map`, and the problem appears to be due to unnecessary hash collisions in the map. Developers are currently experimenting with customizing the usage (*e.g.,* different hash and different bucket counts) as well as a custom data structure to improve performance.

Overall, AMM code developers were very impressed with CallFlow's capability to easily and concisely describe the performance variability and help identify potential bottlenecks. In general, performance profiling and optimization is an ongoing process, and with the availability of the tool, developers will reevaluate the forthcoming development versions of their code.

4.5.2 Study 2: Performance Trends for a Weak Scaling Study

Large-scale parallel applications are designed to scale to several hundreds to thousands of computational nodes and/or utilize several cores per node. Application developers and HPC experts are often interested in scaling studies of such applications to understand

whether the codes are leveraging parallelism (*e.g.,* via MPI) effectively. Previously, a typical workflow for such use-cases would be the user generating static charts (*e.g.,* bar plots) to determine the changes in the overall run time, perhaps at the module level or call site level. Usually, the level of granularity is scripted in to generate the plot, and then analysis performed, possible across levels of detail. Nevertheless, the lack of an automated UI-based visualization imposes high time-to-insight, even for an expert user.

Here, I are given an ensemble of multiprocess performance profiles to study weak scaling of an application across eight execution parameters: 1, 8, 27, 64, 125, 216, 343, and 512 processes. In particular, the application is LULESH [151], a Lagrangian shock hydrodynamics mini-application that uses both MPI and OpenMP to achieve parallelism. Recently, a similar case study was conducted by Bhatele *et al.* [143] to study run-to-run performance differences for identifying the most time-consuming regions of the code. I used a similar collection of profiles and showcase the advantages of visual analytics for such exploration using CallFlow.

Exploratory overview using ensemble-Sankey. The given profiles are converted into an ensemble super graph, which is shown to the user using the ensemble-Sankey layout (see Figure 4.5(a)). Here, $\mathbf{G}^{\mathrm{sg_E}}$ contains 33 call sites, which we group into six modules. Immediately, the user is conveyed the overall flow of the resource (runtime) that establishes the context as the user can identify the expected calling pattern. The ensemble gradients on the different modules of the visualization describe the *complete* distribution, as compared to a summary, *e.g.,* the mean value. The interactivity of CallFlow allows the user to select different ensemble members as target runs (**A4**) — a functionality particularly appreciated by the users because it allows comparing data in the backdrop of a consistent context, *e.g.,* it is straightforward to compare the "thickness" of the green (target) edges against the gray (ensemble) edges, in Figure 4.5(a). Thus far, an initial exploration of the data using the ensemble-view provides a good understanding of the collected profiles — not only qualitative but also quantitative (due to the associated labels and colormaps). Overall, although this is a small ensemble, the gradient patterns largely indicate a reasonably good weak scaling.

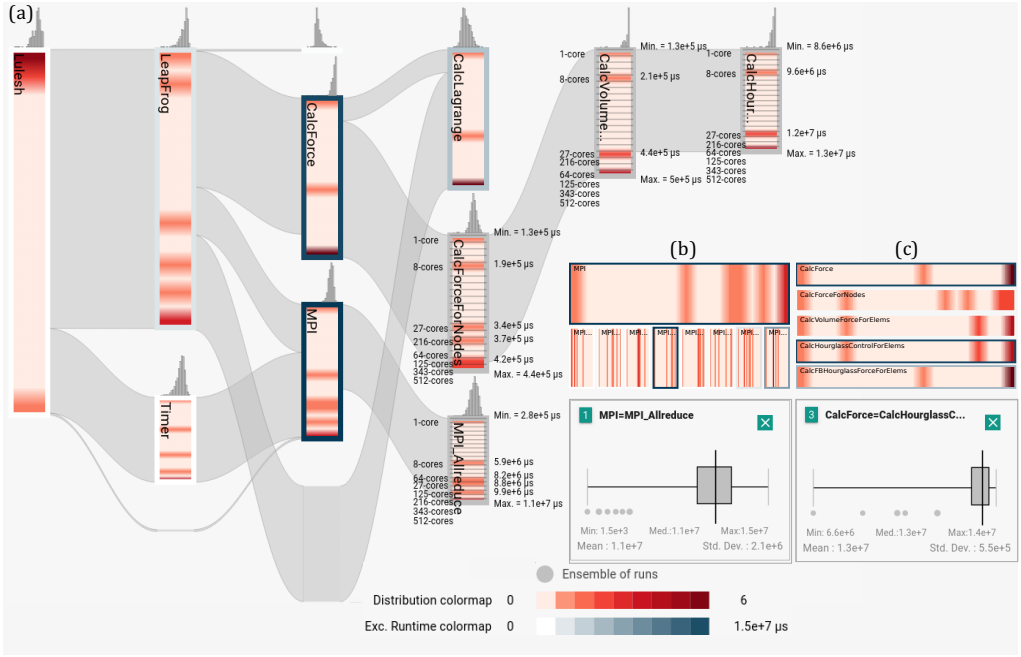


Figure 4.6: LULESH profiles comprising of 8 runs using graph splitting operations and text labels reveal out-of-order runtimes (with respect to the scaling of resources). Further investigation through a sequence of splits as well as through (b) and (c) supernode hierarchy views lead to the problematic call sites: CalcHourGlassControlforElems and MPI_Allreduce.

Pairwise comparison of profiles. Users are often also interested in comparing pairs of profiles within a larger ensemble, *e.g.,* a median profile vs. a slow profile. This is specifically true for the given data set as a curious behavior is observed: two pairs of executions appear out of order in the ensemble-Sankey (labeled later in Figure 4.6).

Although the analysis tool Hatchet can compute the *diff* between two profiles, the resulting information is displayed using a color-mapped text-based tree visualization (similar to linux's *tree* command) [143, Fig. 11].

Visualization of differences using colored text, however, imposes additional perceptual complexity due to a lack of contrast among the colormapped values with respect to the background. Such visualization also suffers from scalability issues and is static with no opportunity for interactive analysis. Instead, the *diff view* provided by CallFlow can reproduce such analysis through a more-effective visual medium (see Figure 4.5(b) and Figure 4.5(c)). The result highlights not only the modules that are slower (with respect

to the diff order) but also communicates the relative degree of performance degradation easily (**A5**). For example, when scaling from 27 to 64 cores per node (Figure 4.5(b)), `CalcForce` (1.39×10^7 for 64 cores, and 1.31×10^7 for 27 cores) becomes about 5% more slower than `CalcLagrange`. On the other hand, the `MPI` module now takes longer when scaled from 125 to 216 cores (Figure 4.5(c)). By encoding the pairwise differences onto the ensemble graph, the visualization not only readily highlights the faster/slower modules, but also allows the opportunity to interactively request more information, *e.g.,* through additional views and/or graph splitting operations. Furthermore, even though the *diff view* compares only two graphs, changing the pair through CallFlow UI allows visualizing the various pairwise differences within a consistent context (*i.e.,* the same ensemble graph), irrespective of any fine-detail changes in the underlying CCT.

Comparing run-to-run slowdowns. Toward the ultimate goal of identifying call sites that exhibit inconsistent and/or unexpected runtime behavior, further exploration is needed. Here, by toggling the text guides (using the click interaction), the runs corresponding to the different bins in the ensemble gradient are listed along with the bin value representing the gradient (see Figure 4.5(a)). The text guides reveal two cases of out-of-order runtimes (with respect to increasing core count) for `MPI` and CalcForce

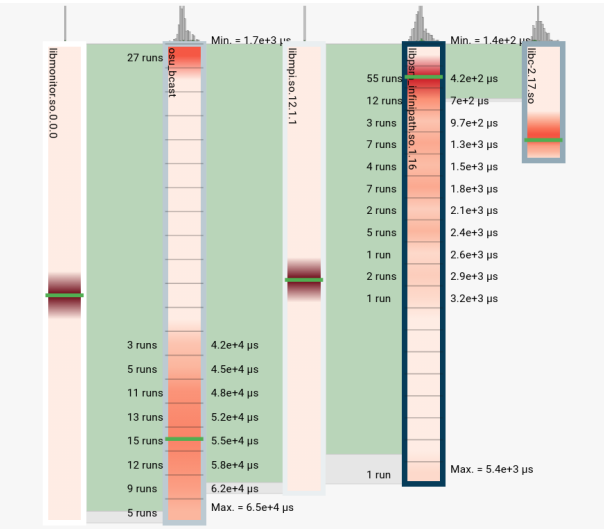


Figure 4.7: CallFlow provides a scalable visual design that can accommodate large ensembles. Here, a case study with 100 profiles is shown.

libraries (**A5**). One way to highlight such differences is using the diff view (Figure 4.5). Here, the ensemble view is further refined to use the interactive *split graph* operation (also available in CallFlow v1.0 [138]) to reveal the nodes inside these two modules (**A1**). Indeed, the text labels in the figure help identify the culprit call site hierarchies, [MPI → MPI_Allreduce] and [CalcForce → CalcForceForNodes → CalcVolumeForceForElems → `CalcHourGlassControlForElems`]. A key advantage of using my framework is the immediate availability of alternate views, *e.g.,* split ensemble view, supernode hierarchy view, and the call site view (all shown in Figure 4.5). In particular, the supernode hierarchy view (Figure 4.5(c)) reasserts the upward propagation of this behavior (**A1**). Finally, I look at the summarized distribution of the respective call sites. Even though the box plots highlight the outliers (with respect to the interquartile range), by design, they are incapable of capturing this anomalous behavior in the data (out-of-order runtimes) (**A2, A3**).

4.6 Summary

In this chapter, I described how CallFlow was extended to create a scalable, interactive visual analytic tool to study ensembles of call graphs. Working closely with domain experts, I identify the specific problems faced in the analysis of collections of performance profiles, and map them to concrete comparative visualization tasks using the framework of Gleicher et al. [140]. I develop a scalable visual encoding, ensemble-Sankey, to describe the ensemble along with several other linked visualizations. Our visual design itself is arbitrarily scalable in size of the ensemble, since I enocde the distributions of runtimes for each supernode. In general, getting an overall context of the ensemble is integral to studying performance profiles. It was noted that the new encoding, the ensemble-Sankey, not only provides a new way of exploring the distributions of interest, but is also remarkably straightforward and intuitive. Supplementary linked views help tie in the various pieces of information to put together a cohesive story about the profiles that the user wishes to understand.

Chapter 5
Visualization of Multivariate Network Traces from the Communication Domain

Analyzing performance behaviors and optimizing HPC applications require programmers to collect various performance metrics from each computing node at different time points as well as the communication events among the nodes. Typically, HPC applications run on a cluster of interconnected computing nodes, a communication protocol, such the Message Passing Interface [63], is often used for coordinate for parallel and distributed computing. To analyze the behaviors and performance of HPC applications, system or application level data need to be collected on each computing node with the communication data among the nodes. Therefore, the collected dataset contains multivariate time-series and communication network data, which makes data analysis and exploration challenging.

A common technique to explore such large-scale multivariate data is to employ automated analytical methods based on statistical methods and unsupervised learning. However, characterizing the analysis results for HPC datasets is often a challenge as end-users often fail to derive insights, especially for large-scale data. To effectively explore and analyze HPC datasets, visual analytics techniques that combined interactive visualization and machine learning methods are required. Multiple approaches and tools have been developed for analyzing the performance of HPC applications. However, conventional visualization methods and tools are insufficient for analyzing both multivariate time-series and communication network data as well as correlating them for performance analysis. In particular, Fujiwara et. al [73] developed a visual analytics system for analyzing Dragonfly [67] networks. They applied various time-series analysis

methods and visualization techniques to allow users to explore both temporal behaviors and network traffics in a dashboard with multiple coordinated views. However, their system lacks support for *correlating the temporal behaviors* and *identifying similarity of the computing nodes to communication patterns*, which is often needed for performance optimization of HPC applications. In addition, the visual analytics methods need to be generalized and extended to provide sufficient support for analyzing large-scale HPC systems and applications.

In this chapter, I present a visual analytics framework for effectively analyzing HPC datasets using automated time-series analysis methods to reveal temporal behaviors and identify important time intervals for potential performance issues. For correlating multiple performance metrics with communication patterns over time, I visualize the communication patterns for important time intervals along with the time-series clustering results. In addition to closely coupling these analysis and visualization methods, the framework also supports interactive visualizations for exploring HPC datasets. To demonstrate the applicability and usefulness of the framework, I use it to develop a visual analytics system for analyzing parallel discrete-event simulation (PDES) [152], which is a cost-effective tool for modeling and evaluating scientific phenomena and complex systems. Through several case studies, I show that the data analytics and visualization methods incorporated by the framework can effectively analyze and correlate multivariate time-series and communication network data. Finally, the work in Chapter 5 and Chapter 6 was done in collaboration with Takanori Fujiwara, Kelvin Jianping Li, and HPC experts from Argonne National Laboratory and Rensselaer Polytechnic Institute.

5.1 Domain Problem Characterization

Our goal is to design a visual analytics framework that helps real-time analysis of streaming performance metrics and communication data. Simultaneously analyzing both performance metrics and communications is critical because the communications heavily affect the performance, and vice versa. For example, a communication bottleneck in a network of compute nodes could be due to a large number of packets transferred between

certain compute nodes leading to excessive wait times for the next computation. To understand such specific analysis needs, we work in collaboration with HPC experts and enumerate the design requirements of the visual analytics framework.

The design process followed a user-centered approach consisting of several discussions where I presented a prototype to the experts to probe further requirements, and modified the visualization framework, accordingly. Below I describe the design requirements (**R1**–**R5**) developed during these discussions. These requirements comprehend analysis tasks of both active monitoring [81] (**R1**, **R2**, and **R4**) and situational awareness [81] (**R1**–**R5**).

R1: Detect key changes that deviate from a baseline behavior. Because large-scale simulations are often long-running applications, it becomes impractical to review performance metrics for changes constantly. Hence, it is critical to automatically find when and where a change occurs in the behavior to narrow down the search space.

R2: Enhance the interpretability to identify performance behavior patterns. As discussed in Section 6.1, the collected data contains time-series data continuously arriving from multiple processes. A common challenge the analyst faces with streaming data is that different factors that affect the perception of the user (e.g., velocity, volume and variety of data). From the streaming data, finding similar and dissimilar behaviors from many processes in real-time is extremely challenging because of its high-volume and high-velocity. Therefore, we need to provide a functionality that helps the analyst identify the behavior patterns.

R3: Derive causal relations among different performance metrics. A behavior change (e.g., excessive wait times) can trigger an undesired effect (e.g., low data receives). Therefore, the analyst would want to identify the causal relations among the multiple performance metrics. Since it is difficult to monitor several metrics (e.g., five metrics) all at once, we aim to support the analysis of causal relations between metrics that significantly affect the performance.

R4: Provide visualizations for analyzing communication patterns together with performance behaviors. As mentioned in Section 6.1, analyzing the performance

data from both hardware and communication domains and relating these domains are important. Thus, the framework should provide visualizations that help the analyst relate the performance behaviors to the communication patterns.

R5: Enable users to analyze the data interactively. Due to the complexity of HPC performance data, there are various aspects that the analyst wants to analyze while cooperating with his/her domain knowledge. The analytical process may proceed and change according to discovered patterns or findings. Therefore, it is essential to provide interaction methods that can be used during analysis.

5.2 Methods

The multivariate time-series and communication network data are first preprocessed and indexed for efficient analysis. Time-series clustering is used to classify each process in a HPC application based on their changes of performance metrics over time, and change point detection algorithms help identify the most important time intervals. To analyze communication patterns, we can filter and aggregate the communication network data based on these time intervals to generate multiple views of the network for identifying bottlenecks. Combining these network views with the temporal behaviors views can help correlate among different performance metrics and communication patterns. An overview of the framework is shown in Figure 5.1.

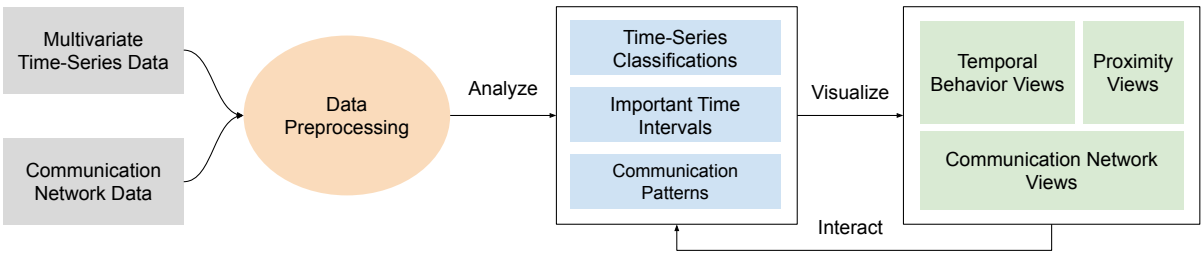


Figure 5.1: Overview of the visual analytics framework for analyzing HPC applications and systems.

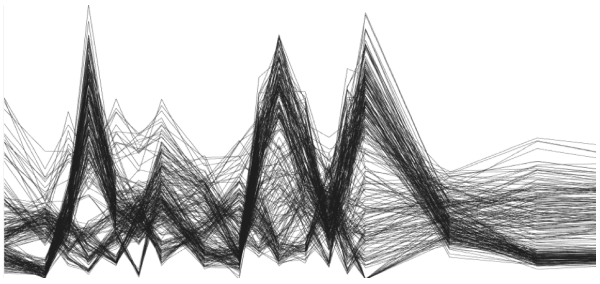
5.2.1 Time-Series Clustering

Analyzing the temporal behaviors of HPC applications is useful for identifying bottlenecks and optimizing performance. By applying time-series clustering techniques, we can easily

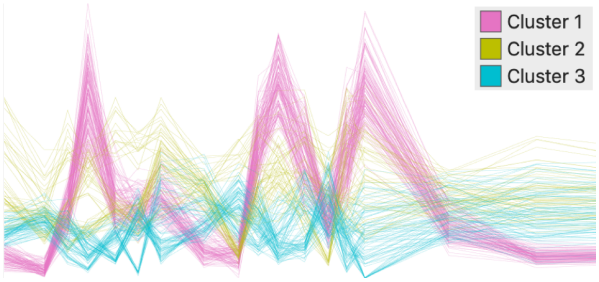
identify the subgroups among the computing nodes at different granularities.

In my approach, I employ the multiple time-series clustering methods [153] and similarity measures used in [80]. In particular, we use the Hartigan-Wong method [154] as a k-means clustering, the partitioning around medoids (PAM) as a k-medoids clustering [155], and the complete-linkage clustering as a hierarchical clustering method. k-means clustering is the fastest method among these options, with time complexity of $O(nk)$ (where n is a number of observations, and k is a number of cluster centers). However, it requires observations of l-dimensional vectors as inputs. Thus, we treat each time-series as one observation and each processor's metric value as an element of the vector. Also, to avoid the initial centroid dependency, the system runs k-means clustering multiple times (ten as a default) with different initial centroid seeds, and then selects the best result. While k-means clustering uses observations as inputs, the other two clustering methods use dissimilarity between each observation as their inputs. Even though their complexity ($O(n^2)$) is worse than k-means, these clustering methods are useful for analysis since (1) they are more robust to noise and outliers [156], and (2) other similarity measures developed for the time-series analysis can be easily applied.

To effectively perform time-series clustering, we use three similarity measures: traditional Euclidean distance, Dynamic Time Warping (DTW) [157], and Time Warp Edit Distance (TWED) [158]. Since DTW and TWED are the elastic similarity measures, we use them to perform flexible matching in the time-series data. In comparison to Euclidean distance which is the simplest and fastest way (complexity of $O(l)$) to calculate dissimilarity of each time-series, DTW and TWED (complexity of $O(l^2)$) have performed better for classification of time-series data according to the recent research [159]. Figure 5.2(a) and Figure 5.2(b) show examples of visualizations without any clustering and with PAM with Euclidean distance as the similarity measure, respectively. The cluster labels are encoded with line colors, as shown in Figure 5.2. In Figure 5.2(b), we can clearly see the different patterns of performance behaviors. Refer to the works of [80, 159] for more details about each similarity measure and the differences between these three measures.



(a) Without clustering



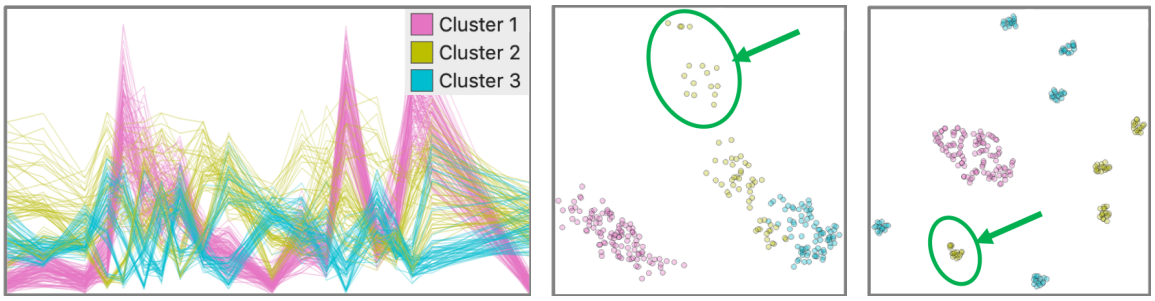
(b) PAM with Euclidean distance

Figure 5.2: Visualized results of the number of secondary rollbacks without and with clustering. Colors represent the clustering labels where the lines belong to. (a) Without clustering, it is difficult to find important patterns. (b) Using PAM clustering with Euclidean distance as the similarity measure, we can easily find the patterns. For example, the pink lines show that the corresponding simulation entities' number of secondary rollbacks have high fluctuations.

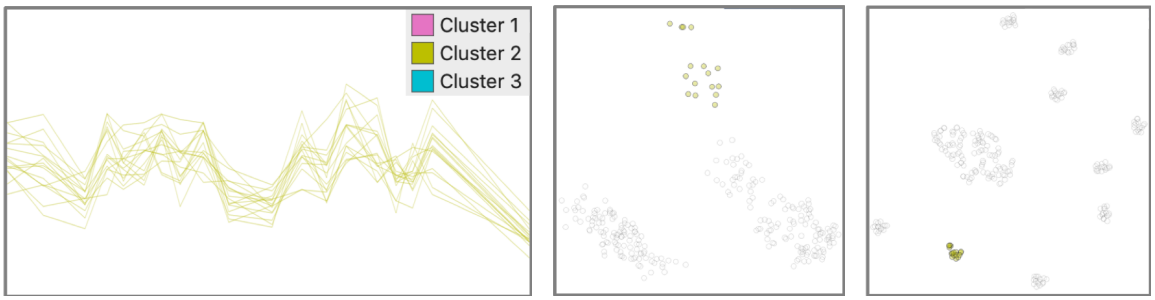
5.2.2 Time-Series Dimensionality Reduction (DR)

Because time-series clustering methods can only reveal certain temporal patterns, it might fail to inform some important patterns which are derived from a small set of the entities (e.g., sub-clusters and outliers). DR methods can supplement time-series clustering because they depict more detailed similarities between each entity.

We leverage classical Multi-Dimensional Scaling (MDS) [160] and t-Distributed Stochastic Neighbor Embedding (t-SNE) [93]. For calculating a similarity between each time-series, we use the same similarity measures used with the time-series clustering. By using these DR methods, we can notice entities with similar behaviors being close together. While the classical MDS is a linear DR method and good for looking at the global structure of the multi-dimensional data, t-SNE is a nonlinear DR method and useful to visualize the local structure of the data. Because t-SNE often requires a longer



(a) The number of secondary rollbacks (b) MDS (c) t-SNE



(d) After selection of (a) (e) MDS (f) t-SNE

Figure 5.3: Examples of time-series DR. (a) shows the number of secondary rollbacks clustered with PAM with Euclidean distances. (b) and (c) are results after dimensionality reduction with MDS and t-SNE, respectively. When compared to the clustering result in (a), we can find clusters of smaller size in MDS and t-SNE, as indicated with green circles and arrows in (b) and (c). (d), (e), and (f) show the small clusters selected from (b) or (c). When compared with MDS in (b), t-SNE in (c) finds more small clusters (e.g., cyan clusters in (b) are separated in four small clusters in (c)).

calculation time, to apply t-SNE more interactively, we use Barnes-Hut t-SNE [161] (while the complexity of the original t-SNE is $O(n^2)$, this implementation has only $O(n \log n)$ complexity). t-SNE has the perplexity as a tuning parameter, which controls a balance of the effects from local and global structures of the data [93]. While a large perplexity will preserve more of the distance relationship in the global structure, a small perplexity will focus on preserving the distance relationship between a small number of neighborhoods In most applications, the perplexity is set between 5 and 50 [93]. We set the default value to be 30, and the user can change the value based on the analysis.

Examples of visualizations with different DR methods are shown in Figure 5.3. These examples show that DR-based visualization helps find subgroups and outliers within the clusters of time-series.

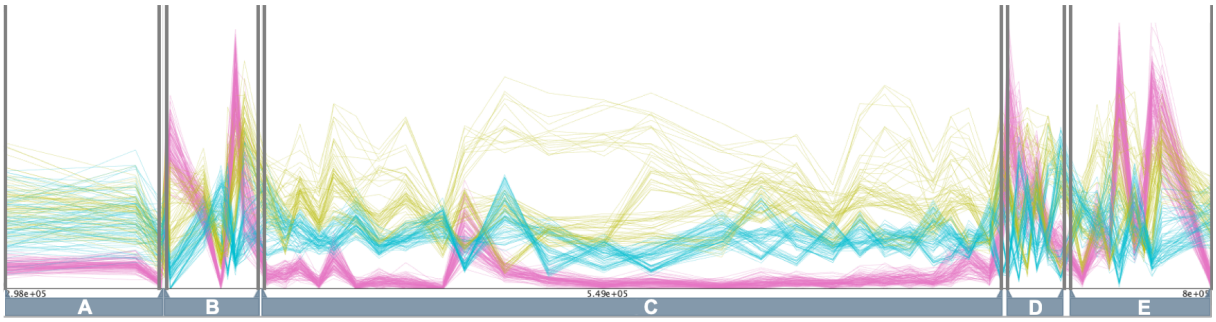


Figure 5.4: An example of segmentation for the temporal performance metric. The E-Divisive detects five segments (from A to E) from multiple lines.

5.2.3 Time-Series Segmentation with Change-Point Detection

With time-series clustering and DR, we can quickly analyze temporal patterns in performance metrics. Remaining required analysis is to understand the effect of communication patterns on the performance [76]. However, exploring and comparing communication data one-by-one at each time is a time-consuming task. To help effectively compare communication data across time, we want to obtain a temporal summary of the changes in communication data. We achieve this by segmenting time-series data and summarizing each segmentation with visual aggregates (e.g., sums or mean values). This idea is inspired by the temporal summary images (TSIs) [162], which is designed for generating narrative visualizations by summarizing the time-series data.

To segment time-series data, we can use the change point detection, which is developed for time-series analysis [163]. We use the E-Divisive method [164] because it can detect multiple change points for a set of time-series in a reasonable amount of time. Figure 5.4 shows an example of segmentation with the E-Divisive method. Combining time-series segmentation with visualization for communication data (discussed in the Subsection 5.2.4), we provide a visual summary of changes in communication patterns in the system, described in Section 5.3.

5.2.4 Visualization of Communication Patterns

For parallel and distributed computing applications, exploring the communication pattern between the processes is essential. In the framework, we use the hierarchical circular visualization techniques for visual analysis of the communication patterns. As shown

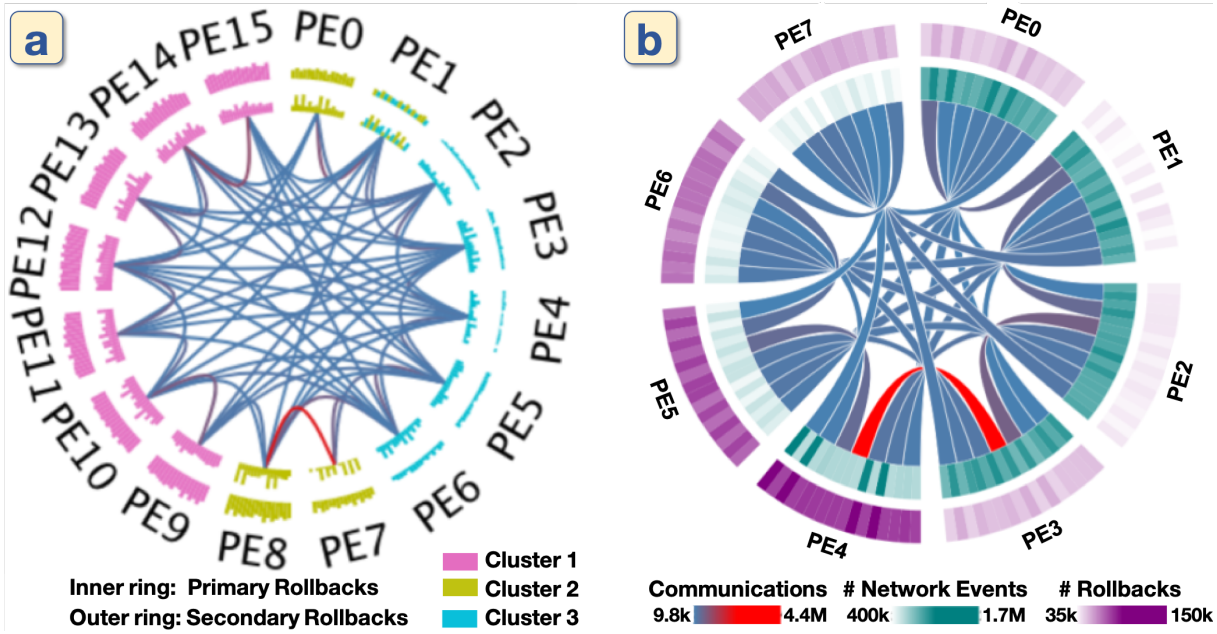


Figure 5.5: Hierarchical circular visualizations for showing communication patterns between entities as well as a correlation between performance metrics. While the color of the ribbons shows a value of the selected communication metric, the color of the rings can be used to encode clustering results (a) or metric values (b).

in Figure 5.5, in a hierarchical circular visualization, lines or ribbons at the center represents the communications (e.g., network sends, network receives, the sum of these, or the maximum value of these) between entities, while different performance metrics (e.g., primary and secondary rollbacks) can be stacked on the circumferences (or rings) to show their correlation to the communication patterns. Hierarchical aggregation is used to group the entities for organizing the circular visualization, so load balancing and distributions can be easily observed.

In addition to the communication patterns and performance metrics of the computing nodes, I also encode the time-series clustering results in the circular hierarchical visualization. Figure 5.5(a) provides an example that we visualize both the time-series clustering results (encoded with color) and performance metrics (encoded with the size of the bars). Comparing with Figure 5.5(b) where I only show the performance metrics, Figure 5.5(a) allows us to correlate time-series clustering results along with the communication patterns, understanding how computing nodes with different temporal behaviors communicate with each other. This allows application programmers to gain

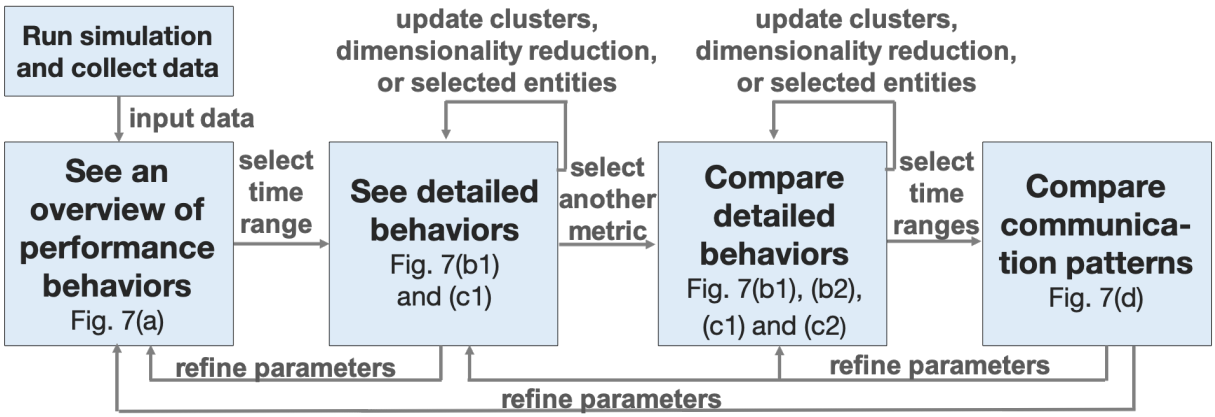


Figure 5.6: The analytical flow of using the system. Each step involves one or more views.

insights on how to assign and map processes on the network to optimize performance and remove bottlenecks.

The selected communication metric is encoded with a blue-to-red colormap. In Figure 5.5(a), the colors of the circular bar charts are used to denote KP's cluster labels obtained with the time-series clustering method described in Subsection 5.2.1 and the heights of the circular bar charts are used to show the values of performance metrics. Alternatively, circular heatmaps can be used instead of circular bar charts to use color or opacity to represent the performance metrics of the KPs, as shown in Figure 5.5(b). As shown in these two examples, communication hot paths (shown in red lines or ribbons) and the workload distributions are clearly revealed.

5.3 Visual Analytic System

Based on the framework, I have developed a visual analytics system for analyzing the performance and behaviors of the ROSS PDES engine. The design of my system and user interface is based on a similar approach used in the previous work for analyzing network performance in supercomputers [80]. My system clearly shows temporal behaviors of the performance metrics (e.g., primary, secondary rollbacks, network sends, etc) by coupling with unsupervised machine learning methods. Also, the system supports visual comparisons of not only temporal behaviors of multiple performance metrics but also communication patterns between the simulation entities (e.g., PEs and KPs) across multiple time intervals.

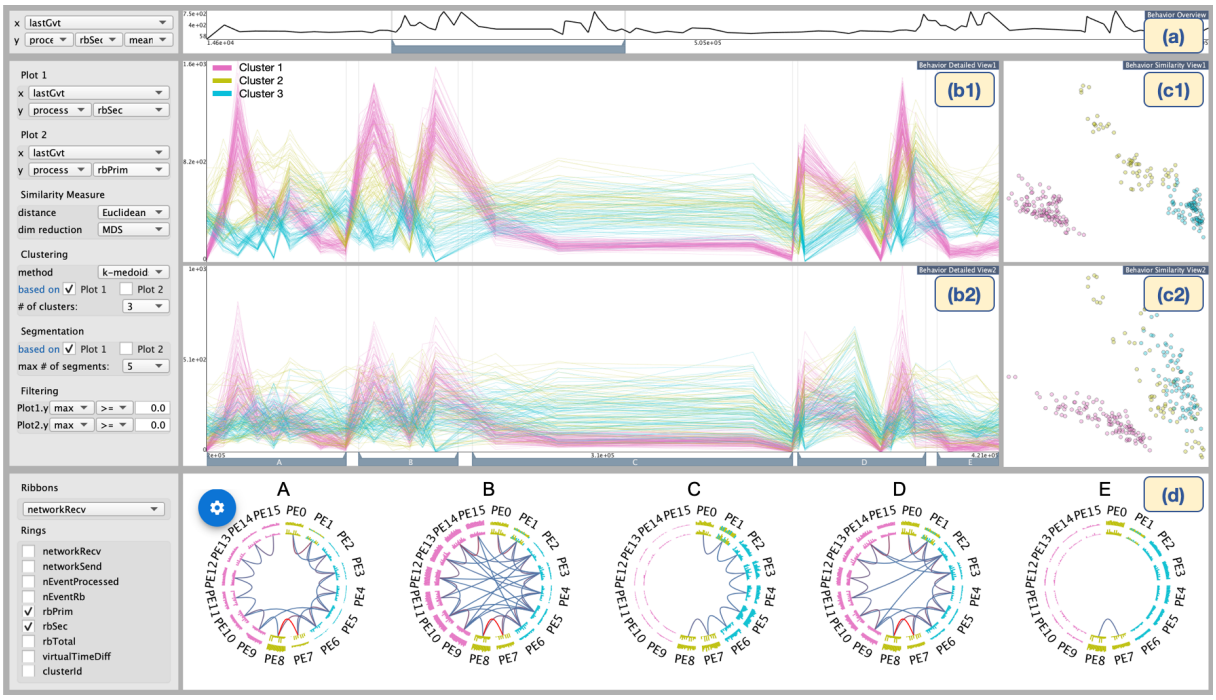


Figure 5.7: The user interface of the system, which contains four components: (a) the behavior overview, (b1, b2) the behavior detailed views, (c1, c2) the behavior similarity views, and (d) the communication network views. This example shows the secondary and primary rollback behaviors obtained in Subsection 5.4.2. (a) shows an overview (the mean in this example) of the number of secondary rollbacks over time. (b1) and (b2) show details of temporal changes of the numbers of secondary and primary rollbacks in the selected time range in (a), respectively. (c1) and (c2) show the similarity of each time-series shown in (b1) and (b2), by using the DR method. In (d), summaries of the metrics and communications of each simulation entity in the selected time ranges.

5.3.1 Visualization for Analyzing Temporal Changes

The first principal component of the system is to visualize the temporal changes in the simulation entities/' performance metrics. As shown in Figure 7.1, the analysis starts with an overview of the temporal changes in the behavior overview Figure 5.7(a). From this overview, the user selects a time range and metrics of interest, and then reviews the details of the performance behavior in the behavior detailed view Figure 5.7(b1) and the behavior similarity view Figure 5.7(c1).

5.3.1.1 Statistical Summary of Performance Behaviors

To help the user find a performance metric and a time range of his/her interest, I provide a statistical summary of a selected performance metric for each time point across time in

the behavior overview Figure 5.7(a). The selected time metric (e.g., GVT, virtual time, or real time) is encoded in x-coordinates. As for y-coordinates, from the collected dataset, the user can select a performance metric (e.g., primary rollbacks and network sends) and a statistical measure (e.g., the maximum value, mean value, or standard deviation) as values for the y-direction. For example, in Figure 5.7(a), GVT and the mean of secondary rollbacks of all KPs are selected for x- and y-coordinates, respectively. This view is also used for a time range selection with a single range selector placed at the bottom to show more detailed information in the other views. For instance, in Figure 5.7(a), the time range where the mean of the number of secondary rollbacks is increasing is selected.

5.3.1.2 Detailed Visualization of Performance Behaviors

The performance behavior of each computing node or process in the selected time range from the behavior overview is visualized in the behavior detailed view, as shown in the Figure 5.7(b1). Similarly with the behavior overview, x- and y-coordinates represent the selected time and performance metric values, respectively. For example, in Figure 5.7(b1), the number of secondary rollbacks for each KP/'s is visualized. Because many polylines would be drawn (e.g., 256 polylines in Figure 5.7(b1)), without adequate visual support, finding interesting patterns from these polylines is difficult. To address this, the system allows users to select a clustering method described in Subsection 5.2.1, with the number of clusters and a similarity measure from the settings, placed on the left-hand side of the behavior detailed views. I select categorical colors, each of which has enough saturation to recognize the differences of each color line with a narrow width. Furthermore, the behavior detailed view and similarity view share the color scheme to correspond the time-series with the clustering results.

5.3.1.3 Visualization of Similarities

While the behavior detailed view Figure 5.7(b1) shows behaviors of performance metrics for each simulation entity, it is difficult to convey the dissimilarity of each behavior in detail. By using DR methods, I visualize the dissimilarity of the behaviors in the behavior similarity view, as shown in Figure 5.7(c1). The clustering methods, described in subsubsection 5.3.1.2, are effective for grouping the behaviors in a macro sense;

however, it would not be enough to find the patterns that occurred in the small set of simulation entities (e.g., outliers and anomaly behaviors). Also, while the hierarchical clustering method can inform the potential subgroups within each cluster if I visualize its clustering dendrogram, k-means and k-medoids clustering cannot provide the subgroup information. The DR-based visualization can help find these patterns (outliers, anomalies, and subgroups). The behavior similarity view in Figure 5.7(c1) shows a result obtained by applying the DR for the behaviors visualized in the corresponding behavior detailed view Figure 5.7(b1).

5.3.2 Visual Comparison of Multiple Performance Metrics

In Subsection 5.3.1, I describe how the system help the user analyze the temporal behavior in one selected performance metric. In addition to this univariate time-series analysis, understanding the relationships between multiple performance metrics is necessary to know how I can achieve better PDES performance. For example, to understand the rollbacks, I need to know why rollbacks happened (i.e., the cause of the rollbacks) and what occurs after the rollbacks (i.e., the effect of the rollbacks).

The system can be useful for visually comparing between the temporal behaviors of multiple performance metrics, shown in Figure 5.7 where the detailed performance behaviors (b1 and b2) and their similarities (c1 and c2) are presented together. To make the comparison between two different metrics easier, the same color that represents the cluster label is used for the corresponding simulation entities (i.e., lines in (b1) and (b2) or points in (c1) and (c2)). The user can select which behavior detailed view will be used for clustering using the settings panel, placed on the left-hand side of the behavior detailed views. Additionally, if the user wants to cluster the network behaviors based on multiple metrics (e.g., the numbers of primary and secondary rollbacks), the clustering methods and similarity measures described in subsubsection 5.3.1.2 can be used to support multivariate time-series data. In this case, the system processes all metrics on a scale between 0 and 1.

Moreover, to support comparison within a subset of the simulation entities, the system provides multiple selection methods. First, the user can apply filtering to the metric

value for each view from the settings. Second, the user can select which clusters to visualize in the view from a context menu, which will be displayed with a right-mouse click. Additionally, in the behavior detailed views, the system provides a freeform selection that selects intersected lines with the freeform drawn by the user. An example of the freeform selection can be seen in Figure 5.9. For the behavior similarity views, a lasso selection is available to select a subset of points. After these selections, the user can filter out the unselected lines or points. The filtered out simulation entities in one view will also be filtered out from the other views at the same time.

5.3.3 Visual Comparison of Communication Patterns

While all the views described above depict the performance behaviors from their time-varying aspects, I visualize a summary of temporal changes in communication patterns as the last system component.

By obtaining time segments with the change-point detection method describe in Subsection 5.2.3, the system visualizes the summaries of behaviors. For each time segment, I calculate mean values for each metric (e.g., network receives and primary rollbacks) for each processor (e.g., PE, KP, and LP), then depict them with the circular visualization method described in Subsection 5.2.4. The user can also adjust these time segments based on their observations or background information. Figure 5.7(d) shows an example of the visualized result. The circular visualization results are placed from the left in the order of the five segments (indicated with the alphabets from A to E), as shown in Figure 5.7(b2). This example shows the network receives between PEs as ribbons drawn in the center. Also, the KPs/' primary rollbacks and secondary rollbacks are visualized in the inner and outer rings, respectively. To allow the user to compare the changes in each metric across time, for each metric, the range of the height or the heatmap used in the rings is shared across different segments. The user also can filter out the ribbons based on the metric value from the blue setting menu placed on the top left of the view.

5.4 Case Studies

To demonstrate the effectiveness of the data analytics and visualization methods, we use the visual analytics system to evaluate the efficiency of ROSS. ROSS can run a large-scale PDES that processes up to billions of events. Interactive analysis and visualization of the instrumented data are necessary for understanding performance issues and removing bottlenecks in order to achieve the highest possible efficiency. Here, we first provide details about the setup of the experiments, then we present three cases for showing the effectiveness of the methods for analyzing various factors that affect the performance of ROSS in simulating next-generation supercomputers.

5.4.1 Experiment Setup

For the experiments, we use ROSS with the Dragonfly [67] network simulation model provided by the CODES simulation framework [165]. The Dragonfly configuration that we use models a system similar to the Theta Cray XC supercomputer [166] at Argonne National Laboratory with 864 routers and 3,456 compute nodes. Each router is represented by a single LP that handles all router functionality, while each compute node is represented by two LPs—one for generating the workload and one handling packet send and receive functionality. This setup results in a total of 7,776 LPs. The workload replayed over the Dragonfly network is an MPI trace from the DoE Design Forward program of the Algebraic Multigrid (AMG) solver for unstructured mesh physics packages with 1,728 MPI ranks [167].

5.4.2 Analysis of PDES Performance

In this case study, we use the visual analytics system to analyze a ROSS simulation with 16 PEs, with each MPI rank associated to a PE with 16 KPs. The result is shown in Figure 5.7. The result provides summaries of the simulation as well as useful insights for identifying and removing performance bottlenecks. The behavior detailed views(Figure 5.7(b1) and (b2)) show the secondary and primary rollbacks, respectively. Change point detection is performed to automatically select the five salient time intervals (A to E). For each of these five time intervals, the system automatically generates a

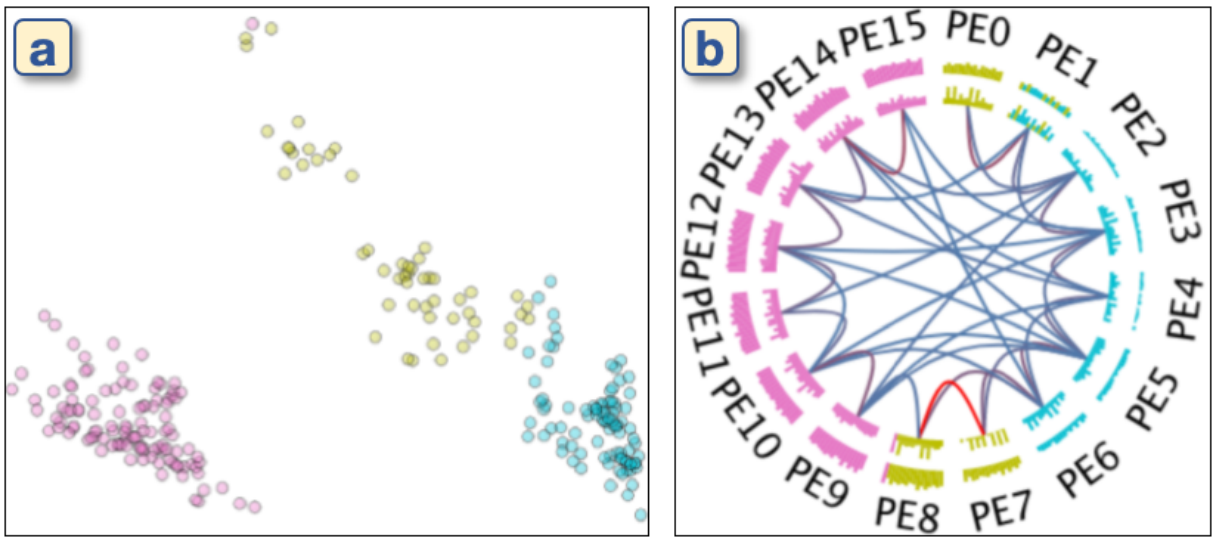
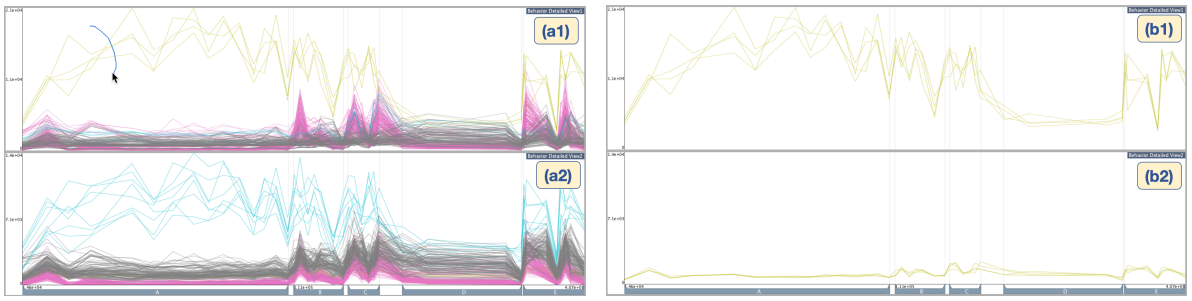


Figure 5.8: DR and clustering results (a) are visualized in communication view (b) to analyze the similarities of the KPs in each PE.

hierarchical circular visualization to show the communication patterns between PEs. Also, the numbers of primary and second rollbacks of the KPs across the PEs as the inner and outer rings of the circular visualization, respectively. The colors in the circular visualizations show the clusters of KPs from the time-series clustering results. This allows us to see the distributions of KPs with different similarities among the PEs.

From the behavior detailed views(Figure 5.7(b1) and (b2)), we can see that the numbers of both primary and secondary rollbacks rise and fall in time interval A, generate two peaks in time interval B, stay low in time interval C, generate two more peaks in time interval D, and then drop to very low in the final time interval. In Figure 5.7(d), the communication patterns for each of these five time intervals are visualized. We can see that the number of communication events increases as the number of rollbacks increases, and the number of communication events between PE 7 and PE 8 is significantly higher in all the time intervals with a high number of rollbacks (A, B, and D). This suggests that the communication between PE 7 and PE 8 might have caused many of the secondary rollbacks, which indicates this is a potential performance bottleneck.



(a) Before selection (b) After selection

Figure 5.9: Visual comparison of the network sends (a1) and network receives (a2) with the behavior detailed views. PAM clustering with Euclidean distance is applied based on both network receives and sends. In (a), the entities which have high network sends are selected with the freeform selection, as indicated with a blue-curved line. In (b), only the selected entities/' network sends (b1) and network receives (b2) are visualized in the behavior detailed views.

5.4.3 Proximity and Communication Patterns

For parallel and distributed computing, assigning processes with very different behaviors to the same computing node might result in performance degradation. In order for PDES to achieve good efficiency, the KPs in the same PE should have similar behaviors to reduce the number of rollbacks. In addition, the KPs that are communicating intensively should have similar behaviors as well, otherwise they can cause rollbacks to each other. With the time-series clustering results providing a measure of similarity for each KP over time, encoding this information in the hierarchical circular visualization allow users to visually correlate the communication patterns to the temporal behaviors based on similarity, which can provide more insights for optimizing performance. Figure 5.8 shows the two views from Figure 5.7 for informing users the similarity among the KPs and the communication view with color encoding the classification of the KPs, respectively. As the hierarchical circular visualization shows, the KPs in each PE have similar temporal behavior, which indicate a good assignment and mapping of KPs to the PEs. However, there are also many communication events occurred between KPs in different clusters, which can increase the number of rollbacks.

While the classification provided by the time-series clustering method only shows a fixed number of clusters (e.g. three clusters in this case), the behavior detail view

(Figure 5.8(a) can reveal the similarity within each cluster. As we can see here, the KPs in the yellow cluster have relatively low similarity when comparing to the other two clusters. We can also see from the hierarchical circular visualization that PE 8 have higher number of rollbacks than PE 0 and 7, and having intense communication with PE8 within the same cluster can also cause performance bottlenecks. This case study demonstrates the usefulness of the visual analytics framework that includes time-series clustering result in the hierarchical circular visualization for visually correlating between temporal performance behaviors and communication patterns.

5.4.4 Interactive Analysis

Analyzing the time-series data of PDES simulations can help understand and correlate the performance behaviors over time, which can gain useful insights for removing bottlenecks and improving performance. My time-series clustering methods and the user interface of the visual analytics system provide effective ways to analyze, compare, and correlate the temporal behavior of PDES. To analyze the temporal behaviors of the simulation with 16 PEs, we can compare two performance metrics over time. Figure 5.9(a) shows time-series clustering (PAM with Euclidean distance) results for the number of network sends and receives over time, with each line representing a KP. The line charts show the peaks of the network sends and receives occur in the middle and near the end of the simulation. The colors of the lines are based on the time-series clustering results of the first selected metric (network sends), showing the subgroups of KPs with similar temporal behaviors.

Users can select any two different performance metrics for comparison and analysis. In addition, the system also supports data filtering for interactive visual analysis of details on demand, which allows a better correlation of performance metrics and temporal behaviors. From Figure 5.9(a1), we can select a subset of KPs with a large number of network sends (yellow lines) and show these in Figure 5.9(b1). Figure 5.9(b2) shows the number of network receives for those selected KPs. We can see that KPs with a large amount of network sends in most of the simulation time have very few network receives.

Encoding time-series clustering results in the hierarchical circular visualizations also helps users to make selections for interactive analysis. As we see from the result in

Figure 5.8, the KPs in the yellow cluster have more intense communication and relatively low similarity comparing to other clusters. Base on this result, we can make select the yellow cluster on the behavior views (line charts) to further investigate and verify our findings. By allowing interactive analysis of different aspects of PDES or other parallel and distributed applications, more insights for optimizing performance can be provided to the users. More important, the analysis and visualization methods in the framework can support users to make selection to better facilitate interactive visual analysis.

5.5 Potential Improvements

Our visual analytics framework is designed for reasoning and interpreting multivariate time-series and communication data collected from HPC applications. As the current effort is just an initial step to develop a full framework for building visual analytics system to analyze HPC datasets, several possible extensions can be added to the framework.

In addition to interactive visualizations, progressive visual analytics can be used to support analysis of large HPC datasets. Progressive visual analytics provides useful intermediate results within a reasonable latency even when the computational cost to complete entire calculations is too high. An advantage of using progressive visual analytics is that we can support the analysis and visualization of streaming data [92], which can be used to enable real-time monitoring and analysis. However, converting the existing workflow for streaming data is challenging, especially for multivariate time-series data. One of the major challenges is how we show important changes or meaningful patterns with a low visual cognitive load since available data is constantly updated [82]. To address such issues, I can use Approximated-tSNE [92] instead of Barnes-Hut t-SNE [161] for dimensionality reduction, and incremental time-series clustering methods [168] instead of conventional k-means clustering. However, PCA-based approaches still suffer from significant false alarms, as they are highly sensitive to changes in any features. The actual projection of points in DR view using incremental PCA methods is highly sensitive to changes in features (*e.g.,* indeterminant sign flipping [90, 169]), thereby causing a lot of visual changes at every time step.

Furthermore, we also plan to leverage in situ techniques [170] for performing data analysis on the computing nodes running the HPC applications. For example, we can perform in situ data processing and visualization within the PDES process. With in situ data processing, the simulation only needs to stream the analysis results to the visualization system, which can significantly reduce the requirement of network bandwidth. For example, we can perform progressive change point detection [171] in situ and send only the data associated with important time intervals instead of logging data for every time step during the simulation. As PDES already leverages distributed computing, combining in situ data processing and progressive analytics can be a scalable solution for real-time monitoring and visualization of the large-scale PDES.

5.6 Summary

In this chapter, I present a visual analytics framework designed to effectively analyze the communication in large-scale HPC applications. The framework utilizes automated time-series analysis methods to reveal temporal behaviors and identify crucial time intervals for potential performance issues. By visualizing communication patterns and time-series clustering results, multiple performance metrics can be correlated with communication patterns over time. The framework not only integrates analysis and visualization methods but also supports interactive visualizations for exploring HPC datasets. To demonstrate the applicability and usefulness of the framework, a visual analytics system was developed for analyzing parallel discrete-event simulation (PDES), a tool used for modeling and evaluating scientific phenomena and complex systems. Through several case studies, it has been shown that the data analytics and visualization methods incorporated in the framework effectively analyze and correlate multivariate time-series and communication network data. Overall, the visual analytics framework presents a comprehensive approach to exploring large-scale multivariate data and offers insights into performance behaviors and communication patterns. Further research and development in this area will be crucial for advancing the field of HPC and improving the efficiency and effectiveness of HPC applications.

Chapter 6
Streaming Visualization of Network Traces from Communication Domain

An ongoing trend in HPC is an exponential increase of the computational throughput, but the bandwidth and capacity of disk storage systems has increased at a much more moderate rate [172, 173]. This has lead to a growing divergence between the amount of data that is generated from computation and the amount of that data that can be captured on the storage system for post-hoc analysis. Like mentioned earlier, existing supercomputers like ORNL's Summit have far more computational abilities in comparison to the storage.

With computational abilities reaching exascale computing (i.e., capable of performing 10^{18} FLOPS in the near future), adopting in-situ workflows [170, 174] has become popular. In-situ workflows integrate visualization into the simulation pipeline to steer the simulation by executing real-time performance analysis. In particular, computational steering method provide the experts a complete understanding the state of running HPC programs and performing interactive control over the programs. Therefore, performance visualization tools not only have to handle streams of real-time performance data but also implement visualizations that can help the analyst comprehend their underlying behaviors. However, existing performance analysis techniques are constrained by several challenges. First, from streaming data, a performance analyst needs to catch important patterns, changes, or anomalies in real-time (active monitoring [81]) without missing them. Furthermore, the analyst often wants to identify causal relations of an occurred phenomena (situation awareness [81]). Because streaming performance data is continuously changing with high-volume and high-variety properties, without any algorithmic and visual supports, conducting the above analysis is almost infeasible.

While existing visual analytic systems [73, 175] aim to support the analysis of dynamic performance data, support for real-time analysis is lacking.

To address the above challenges, I introduce a visual analytics framework for analyzing streaming performance data [176]. I first identify the key analysis requirements through extensive exchange with HPC experts. The requirements include revealing temporal patterns from multivariate streaming data and correlate these temporal patterns to their network behaviors. To handle streaming data and fulfill the experts' requirements, the framework comprises (1) data management, (2) analysis, and (3) interactive visualization modules.

To support the analysis and visualization of streaming performance data, I contribute a data management module for efficiently combining and processing data streams at interactive speed (e.g., 1 second [90]). Our data management module is also designed to be run as a subsystem in remote HPC or simulation systems to enable in-situ data processing and interactive visualizations. It collects both multivariate time-series and network communication data, performs user-specified analytical processing, and delivers the results as data streams to the analysis and interactive visualization modules.

As for the analysis module, I design a set of algorithms for visually analyzing multivariate streaming data in real-time. In particular, I apply change point detection to identify key changes, time-series clustering and dimensionality reduction to reveal both common and outlier behaviors, and causal relation analysis to identify metrics' co-influences. A major challenge to develop these analysis methods is the constraint of the computation time. As new data keeps coming in, the computation must be fast enough to provide up-to-date results. To achieve this, I design the algorithms in either an online or a progressive manner. Online (or incremental) methods [177] calculate the new result by using the result from data already obtained as the base and then updating it according to the newly obtained data. Therefore, if applicable, online methods are suitable for streaming data analysis. On the other hand, progressive methods [87] provide useful intermediate results for data analysis within a reasonable latency when the computational cost to complete an entire calculation is too high. While many progressive methods

internally utilize online algorithms to generate the intermediate results, they may use a different approach, such as reducing computation with approximation [92].

Another challenge is that the results from the analysis may keep causing drastic changes in the visualization content, such as node positions obtained from dimensionality reduction. Showing such results may disrupt the analyst's mental map. Therefore, in this work, I consider mental map preservation while designing the analysis methods. The interactive visualization module is for the analyst to interpret the results from the analysis module. The module provides a fully interactive interface to help relate the temporal behaviors from the historical context with the intermediate results to make critical observations. Additionally, I provide visual summaries for indicating when and what causes a particular performance bottleneck. Finally, I demonstrate the efficiency and effectiveness of the framework with performance evaluation and a multi-faceted case study analyzing the data collected from a parallel-discrete event simulator.

6.1 Characteristics of Streaming Performance Data

In this section, I describe the characteristics of streaming HPC performance data. These requirements lead to the design of the framework which can translate unsupervised machine learning techniques to work in a progressive setting and discuss the design of the interactive visual interface to facilitate analysis of large streaming HPC data.

Analysis of performance bottlenecks in HPC can be categorized into three key domains according to the HAC model [6] as follows: (a) Hardware, (b) Application, and (c) Communication domains. The hardware domain consists of network nodes and physical links between them; the application domain represents the physical or simulated system designed to solve the underlying problem; the communication domain represents a communication network, which captures the communication patterns of the application. Since there exists a large number of visualizations that enable data analysis on the application domain using both post-hoc [178] and in-situ [170, 174] based workflows, we especially focus on the data derived from the hardware and communication domains.

Performance counters and other measurement devices on modern microprocessors

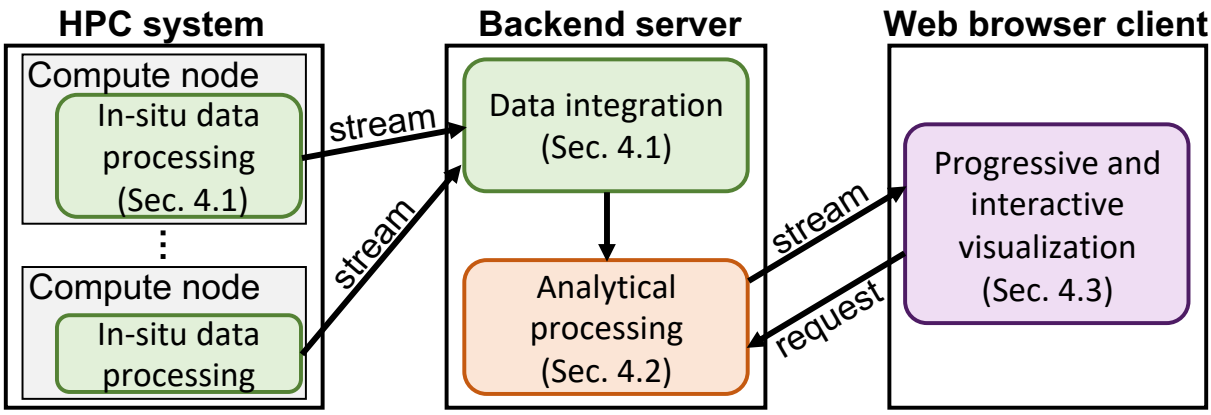


Figure 6.1: The overview of pipeline of the visual analytics framework. Data management module (green) joins and indexes the time-series and communication data collected from the HPC system, and then passes them to the analysis module (orange) for analytical processing. The results are then streamed to the interactive visualization module (purple) for rendering. Also, the analysis results can be updated based on interactions (e.g., changing algorithms/' parameters).

record various metrics from the hardware domain at a uniform sampling rate. The recorded data can be represented as a d-dimensional vector, where d is the number of measured metrics. Let n be the number of entities (e.g., compute nodes, network routers, or MPI processes mapped to CPU cores) which provide d measured metrics. The resulting streaming data can be represented as the third-order tensor object of shape $n \times d \times t$, where t is the number of samples recorded and is continuously growing with time.

Additionally, in general, n entities communicate with each other to run the parallel applications. The communication data can be represented as a weighted graph where nodes correspond to the entities, and links represent the amounts of communications (e.g., message packet sizes) between the entities. Because communication bottlenecks often cause the performance bottlenecks, analyzing the performance metrics with the communication data can help the analyst in locating performance issues.

6.2 Data Management Module

For performance analysis using streaming data, the data management module is designed to leverage with the in-situ data processing and manage data flows in a server-client architecture, as shown in Figure 6.1. Our framework follows the co-processing model for

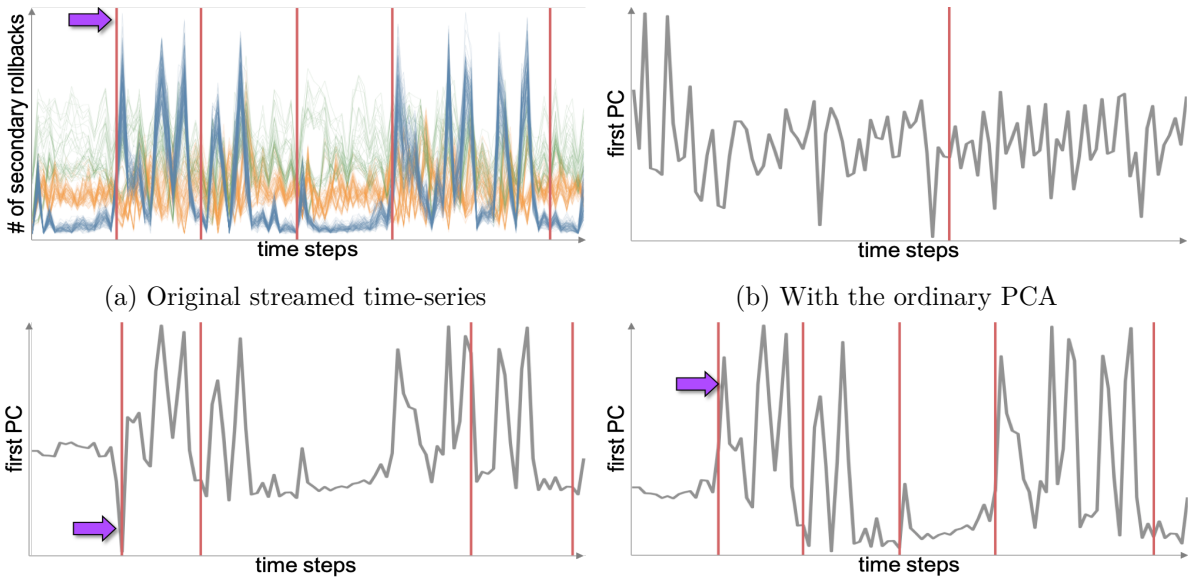
in-situ data processing described in [170]. HPC applications or parallel simulations that adopt this model can be seamlessly integrated into the framework for real-time analysis and visualization. To collect performance data from HPC systems, the data management module uses agent programs that can be installed in multiple compute nodes. Since the performance data on each compute node can be very large, directly collecting the raw data and streaming to the backend server of the visual analytics system requires high network bandwidth. Our data management leverages in-situ data processing techniques [170] for only collecting the data we needed for real-time analysis and monitoring. The minimum data granularity and resolution can be specified for in-situ processing of the time-series and communication data. Common data transformations, such as sampling, filtering, and aggregation, can be specified in the in-situ data processing. The results of in-situ data processing are streamed to the backend server, where the data management module performs data integration and indexing to allow the analysis module to perform analytical processing on the results effectively. The analytical processing results are then streamed to the web browser for rendering visualizations. The use of in-situ data processing greatly reduces network bandwidth, and thus allows the visual analytics framework to effectively facilitate real-time analysis and monitoring of streaming performance data.

6.3 Analysis Module

To support the design requirements, the analysis module provides several automatic analysis methods. The difficulty in applying the algorithms to streaming data is that the computation time should be shorter than a time span of the data update. Thus, instead of using traditional offline algorithms, we introduce methods using an online or progressive approach. Additionally, to make it easier to follow changes in the analysis results, we also provide algorithms to keep visual consistency between the previous and updated results.

6.3.1 Online Change Point Detection for Multiple Time-Series

To address **R1**, we design an online change point detection (CPD) method. As discussed in Section 6.1, for each metric, data obtained from HPC systems consist of n multiple



(a) Original streamed time-series

(b) With the ordinary PCA

(c) With incremental PCA (IPCA)

(d) With IPCA and sign adjustment

Figure 6.2: The comparison of change points detected with the different approaches. In (a), each y-coordinate represents the number of secondary rollback in each entity. The color of each polyline represents the cluster ID obtained with the method described in Subsection 6.3.2. In (b), (c), and (d), the resultant single time-series with each approach are shown with the gray polylines. The detected change points are also indicated with the red vertical lines (in (a), we show the same change points with (d)).

time-series [79,80,175] (e.g., message packets sent from each router). However, the existing online CPD methods are designed to identify change points for a single time-series. To apply CPD on multiple time-series data, we employ a similar approach to [179]. Before using CPD, to obtain a single time-series, their method reduces multiple values to a *representative* value for each newly obtained time point with PCA.

While the approach in [179] applies the ordinary PCA [180], we use incremental PCA (IPCA) of Ross et al. [95]. By using IPCA, we can incrementally update the PCA model when we obtain a new time point and then use this updated model for generating the representative value. This approach can capture the important information from multiple time-series with consideration of the variance in the past observations. Figure 6.2(a), Figure 6.2(b), and Figure 6.2(c) show the original streamed data comprising of 256 time-series, the result with the ordinary, and the result with IPCA, respectively. While Figure 6.2(b) does not summarize any useful patterns from the original data, we can see

that Figure 6.2(c) tends to summarize the blue lines since these lines have higher variances over time compared with the others.

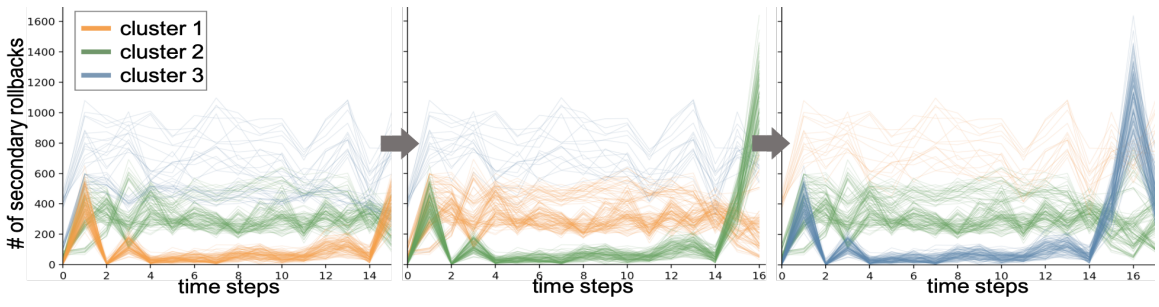
However, (incremental) PCA causes an arbitrary sign flipping for each principal component (PC), as discussed in [90, 94, 169, 181].

The arbitrary sign flipping may cause or hide the drastic changes in a PC value, and thus a CPD method may misjudge the flipping as the change point or overlook the change point. To solve this issue, we generate coherent signs based on the cosine similarity of the PCs of the previous and current results. If the cosine similarity is smaller than zero, these PCs have opposite directions from each other, and thus, we flip the sign of the updated PCA's PC. Figure 6.2(c), and Figure 6.2(d) show the results without and with the sign adjustment, respectively. While the first peak in Figure 6.2(a), as indicated with the purple arrow, is appeared as the negative peak in Figure 6.2(c), Figure 6.2(d) shows the corresponding peak in the same direction as in Figure 6.2(a).

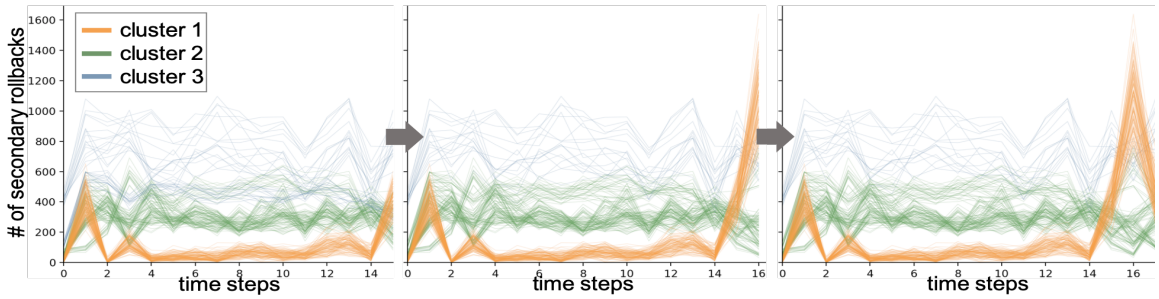
The remaining process of the algorithm is applying an online CPD method that is designed to detect changes on a single streamed time-series. We choose the method developed by Bodenaham and Adams [182] because, unlike most of the others [171], their approach requires only one parameter. Minimizing the number of required parameters is important, especially for streaming data, since parameter tuning is extremely difficult while the data is continuously changing and users are often unaware of the characteristics of the data (e.g., trend, patterns, and cycles) in advance. The required parameter in [182] is called the significance level α ($0 \leq \alpha \leq 1$), which controls a time-window width referred for CPD. As α increases, the detector becomes more sensitive and also tends to generate more false detections [182]. We set a default value of $\alpha = 0.01$. Figure 6.2(a) shows the result with my method. We can see that my method detects changes corresponding to the starts and ends of the high peaks.

6.3.2 Progressive Time-Series Clustering

To support **R2**, we introduce a progressive time-series clustering method for streaming data where a fixed number of entities (e.g., routers) keep obtaining values (e.g., message packets sent) for each new time point (i.e., with the notations in Section 6.1, the fixed n



(a) Without cluster ID reassignment



(b) With cluster ID reassignment

Figure 6.3: The results of progressive time-series clustering (a) with and (b) without the cluster ID reassignments. We plot the number of secondary callbacks for 256 entities with 15 (left), 16 (center), and 17 (right) time points. We set 50 ms as a required latency for this example and 147, 147, and 141 of 256 entities are processed to obtain the clusters with 15, 16, and 17 time points, respectively.

and the increasing t for each metric). While many online clustering methods have been developed [171], these methods are suited for streaming data only when a new entity will be fed with a fixed number of metrics (i.e., the increasing n with the fixed d) [171, 183]. Therefore, we cannot directly apply the existing methods to cluster entities/' behaviors.

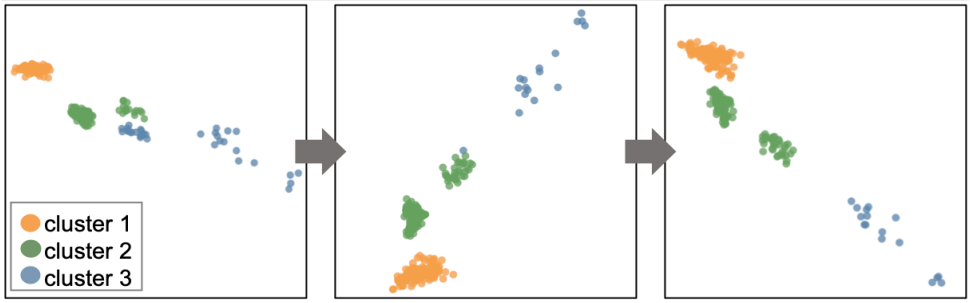
Instead, we develop a progressive clustering method for streaming data to obtain the reasonable results with low latency. Similar to [90], my method internally utilizes the incremental update mechanism used in online clustering. We employ mini-batch k-means clustering [184] which is a variation of k-means clustering. Unlike the ordinary k-means clustering, mini-batch k-means can incrementally update the clustering result with the new subset (or mini-batch) of entities. When n entities' values for a new time point arrive, we keep updating the clustering results with m selected entities ($m \ll n$) until the specified latency or finishing to process all the data points. Then, we obtain the final k cluster centers and then assign each entity to a cluster that has the closest center. Since

it might not be able to process all n entities within the specified latency, the processing order would affect the clustering result. In order to select a wide variety of entities, the algorithm randomly selects m entities from each of k-clusters which are calculated from the previous data points. Figure 6.3 shows the clustering results of the progressive clustering with a latency of 50 ms. We can easily note that three clusters indicated with the orange, green, blue colors have different behaviors. We can easily note that the clustering results produce similar results with the ordinary k-means clustering with a latency of less than 50 msec.

Another issue in applying a clustering method to streaming performance data is the consistency in the assigned cluster IDs between the previous and updated results. As shown in Figure 6.3(a), most of the clustering methods including mini-batch k-means clustering generate the arbitrary order of cluster IDs for each execution. This causes a critical issue in the user 's mental map preservation because the polyline colors would be changed every time when each entity obtains a value for a new time point. To solve this issue, the algorithm reassigns cluster IDs in the updated clustering results based on the relative frequency of previous cluster IDs which were assigned to each updated cluster ID/'s entities. Similar to the progressive clustering as described above, we incrementally calculate the relative frequency by checking a randomly picked-out entity from each updated cluster ID to provide the result within the specified latency. Afterward, the algorithm sorts the relative frequency among all the updated cluster IDs and then reassigns the updated cluster ID to the previous cluster ID from which has the highest relative frequency. Figure 6.3(a) and Figure 6.3(b) show visualized results without and with the cluster ID reassignment, respectively. We can see that the method reduces unnecessary color changes between the previous and updated results.

6.3.3 Progressive Time-Series Dimensionality Reduction

To further support **R2**, we introduce progressive time-series dimensionality reduction (DR). DR methods can visualize (dis)similarities of temporal behaviors among entities as spatial proximities in a lower-dimensional plot (typically 2D). DR methods supplement analyzing temporal behaviors with clustering methods because DR results can help



(a) Without the Procrustes transformation

(b) With the Procrustes transformation

Figure 6.4: The results of progressive time-series DR (a) with and (b) without the Procrustes transformation. The same time-series and cluster IDs shown in Figure 6.3(b) are used. In this example, we set 1 ms as a required latency and 132, 130, and 138 of 256 entities are processed for the left, center, and right results, respectively.

reveal small clusters (e.g., subclusters) and outliers (e.g., entities which have abnormal behaviors), which are difficult to be found when using only clustering methods. However, similar to the situation of time-series clustering, the existing online DR methods (e.g., [92,94,95]) are not designed to update a lower-dimensional representation when new values arrive.

Therefore, we again adopt the incremental update used in incremental DR methods to obtain reasonable results within a required latency. We employ IPCA [95]. To obtain principal components (PCs), we keep updating the IPCA result with m selected entities until the specified latency or finishing to process all the entities. Afterward, we project all entities/' time-series into a 2D plot with the first and second PCs. We use the same processing order as Subsection 6.3.2. Subsection 6.3.2(b) shows results obtained with the method. From Subsection 6.3.2(b), we can easily discern a subcluster and an outlier from the main clusters.

As discussed in [94], (incremental) PCA also has an issue in the user/'s mental map preservation because PCA generates arbitrary sign flipping and rotations. An example of this issue is shown in Subsection 6.3.2(a). We apply the Procrustes transformation [185] used in [94] to provide consistent plots between previous and updated results. The Procrustes transformation tries to find the best overlap between two sets of positions (i.e., the previous and updated PCA results in the case) by using a combination of translation, uniform scaling, rotation, and reflection. Subsection 6.3.2(b) shows visualized results with the Procrustes transformation. In Subsection 6.3.2(b), we can see that flips of data points/' positions along y-direction in Subsection 6.3.2(a) are resolved.

6.3.4 Progressive Causal Relation Analysis Methods

To support **R3**, we introduce progressive causal relation analysis methods. We support three methods: Granger causality test [186], impulse response function (IR) [186], and forecast error variance decomposition (VD) [186]. Based on the definition of Granger causality, we can judge *a time series* X_t causes another time series Y_t... if present Y can be predicted better by using past values of X than by not doing so [187]. Granger causality test evaluates a statistical hypothesis test where one time-series has this *Granger causality* from/to another time-series. Similar to the other statistical tests, Granger causality test provides whether there exists Granger causality between two time-series with p-value. However, with Granger causality test, we cannot measure how much one time-series affects another time-series. IR and VD can provide such quantitative information. IR describes how much a shock to a variable of interest at a certain moment affects the other variables at subsequent time points. On the other hand, VD provides the contribution of a shock to a variable of interest at a certain moment to the variance of the forecast error of other variables. An example of the result of Granger causality test, IR, and VR can be found in Subsection 6.4.3.

All of these three analysis methods can be used after fitting one multivariate time-series to a vector autoregression (VAR) model [186]. However, there are two challenges in applying VAR model fitting to streaming performance data. One problem is that because HPC performance data consists of multiple entities, there are multiple time-series for each

variable (i.e., metric), and thus, we cannot directly fit to a VAR model. To solve this problem, we use IPCA with the sign adjustment described in Subsection 6.3.1.

Another problem is the high computational cost of VAR fitting: $O(t^2)$ (t is the number of time points). However, no incremental algorithm is available for VAR model fitting. To provide a result within around a specified latency, we adaptively control the numbers of time points used in VAR model fitting. Our progressive VAR fitting starts with s time points ($s \ll t$, we set $s = 10$ as a default), which are randomly selected from t time points. We then obtain the result with s time points in t_c completion time. Let t_l be the user-specified latency. Because the computational cost of VAR fit is $O(t^2)$, from the first calculation with s time points, we can roughly estimate how many time points we can fit to a VAR model within remaining time t_r (i.e., $t_r = t_l - t_c$). This estimation calculated with $s\sqrt{t_r/t_c}$. We update s with $s\sqrt{t_r/t_c}$. Then, the updated s is used for the next VAR fitting. These steps will be continued until $t_r \leq 0$ or obtaining a VAR model using all t time points.

Also, when we obtain a new time point, we can expect that the number of time points which can be processed will be similar to the last s in the previous progressive VAR fitting. Therefore, we start by using this s in progressive VAR fitting when obtaining a new time point. If the completion time t_c in the previous VAR fitting with the last s is larger than the required latency t_l, my method updates s with $s\sqrt{t_l/t_c}$ and uses this updated s for the next VAR fitting.

6.3.5 Performance Evaluation

While the qualities of results with the methods depend on the existing methods used as the bases (e.g., IPCA), we evaluate computational performance for each method to provide reference information of the handleable data size. We use an iMac (Retina 5K, 27-inch, Late 2014) with 4 GHz Intel Core i7, 16 GB 1,600 MHz DDR3.

For the online CPD method, the computational cost differs based on the number of time-series (i.e., n entities) which we will reduce to the representative time-series with IPCA. Therefore, we measure the completion time for each different n (from 100 to 100,000), as shown in Table 6.1(a). The computation of the progressive time-series

Table 6.1: The performance evaluation results ((b)-(d): processed in 1 s).

(a) Online CPD

n	completion time
100	0.01 ms
1,000	0.05 ms
10,000	0.46 ms
100,000	6.23 ms

(b) Progressive Clustering

t	# of entities
100	5,392
1,000	4,921
10,000	3,074
100,000	562

(c) Progressive DR

t	# of entities
100	10,000
1,000	10,000
10,000	2,396
100,000	118

(d) Progressive VAR fitting

d	# of time points
10	10,000
100	833
1,000	35

clustering depends on the number of iterations in the k-means algorithm, the number of clusters to form, and the data length for each time-series (i.e., t time points). We use fixed numbers of 100 and 3 for the numbers of iterations and clusters, respectively. Then, with different t values from 100 to 100,000, we measure the number of time-series the algorithm processed from 10,000 time-series in one second. The result is shown in Table 6.1(b). Similarly, as shown in Table 6.1(c), we evaluate the method with different t values because its computation depends on the value of t. Lastly, for the progressive VAR fitting, the number of time points can be processed is different based on the number of measured metrics (i.e., d metrics). Thus, as shown in Table 6.1(d), we measure the number of time points processed from 10,000 time points with multiple numbers of d.

From the results in Table 6.1, we can see that the online CPD is fast even when n is large (e.g., 6.23 ms for $n = 100,000$). With a setting of 1 s latency, the clustering and DR methods processed large numbers of entities up until $t = 10,000$. This could be a reasonable amount of processed entities when we want to analyze several or tens of thousand entities. Also, the VAR fitting processed more than 800 time points even when $d = 100$. The quality of the results with the progressive algorithms depends on how much of the input data would be processed. Thus, we should consider the balance of required latency and the size of input data based on available computational resources. The performance results above can help us decide the granularity level of entities to be analyzed (e.g., compute node or CPU core level), the width of the time window, and the

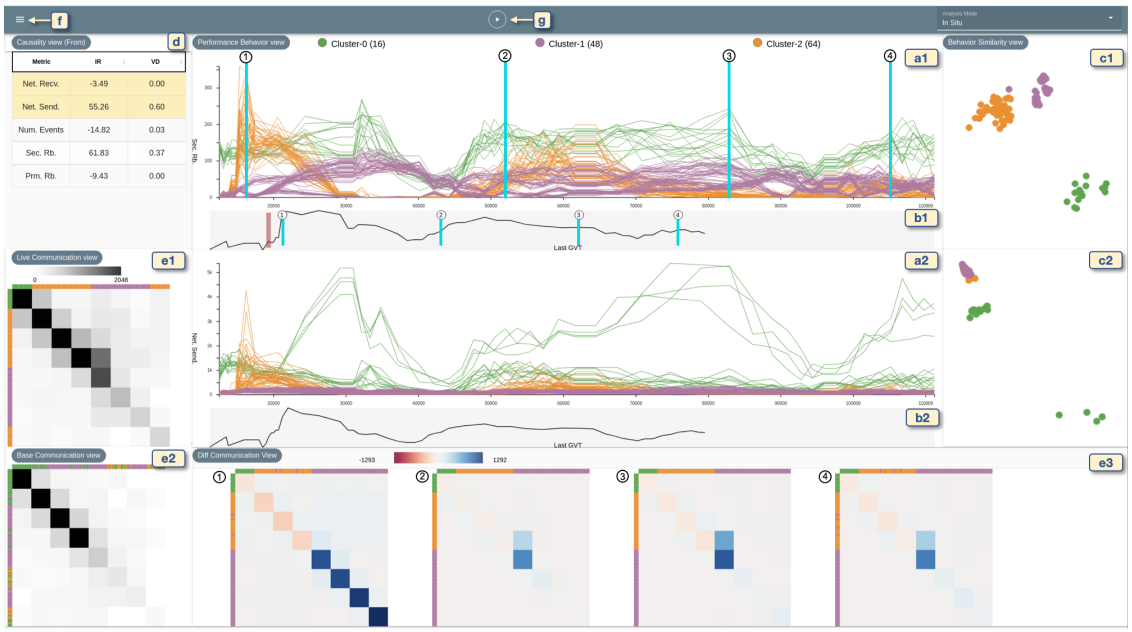


Figure 6.5: The user interface of the prototype system consisting of multiple views: (a1, a2) the performance behavior views, (b1, b2) histories of performance behaviors, (c1, c2) behavior similarity views, (d) metric causality view, and (e1, e2, e3) communication behavior views. (f) and (g) are the drop-down lists for settings and the button for pausing and resuming streaming updates, respectively.

number of performance metrics collected.

6.4 Interactive Visualization Module

This module provides a user interface, as shown in Figure 5.7, which comprises multiple views to fulfill all the design requirements.

6.4.1 Performance Behavior Views

To fulfill **R1**, **R2**, and **R3**, the performance behavior views, as shown in Figure 6.5(a1, a2), provide visualizations with the change points (refer to Subsection 6.3.1) and clusters obtained (refer to Subsection 6.3.2) in the analysis module. Using polylines, each performance behavior view presents the temporal changes of a performance metric for all the entities. Since the collected data comprises information from n entities, each view shows n polylines for each selected metric. x- and y-coordinates represent a time point and a metric value respectively. From the drop-down lists displayed by clicking the icon in the top left (Figure 6.5(f)), the analyst can select two metrics from d collected metrics

for the top (a1) and bottom (a2) views. For example, in Figure 6.5, as indicated by the y-axis labels, 'Sec. Rb.' (i.e., Secondary Rollback) and 'Net. Send.' (i.e., Network sends) are selected. We provide two views to support the comparison of performance behaviors for two different metrics, especially to review the causal relations (**R3**), as described in Subsection 6.4.3.

The online CPD described in Subsection 6.3.1 is continuously applied to the time-series shown in the top view as a default. The analyst also can apply the CPD to the bottom view by selecting from the drop-down list in Figure 6.5(f). When a change point is detected, the corresponding time point (i.e., x-coordinate) is indicated with teal vertical lines in the corresponding view. For example, four change points are detected in Subsection 6.4.3.

Similarly, the progressive clustering described in Subsection 6.3.2 is applied to the top view as a default. The metric to perform clustering and the number of clusters can be modified from the drop-down lists in Figure 6.5(f). Obtained cluster IDs are encoded as the colors of polylines. The categorical colors are chosen to provide sufficient contrast for distinguishing the clusters. A legend for the cluster IDs are shown in the top of Figure 6.5(a1). The number in each parenthesis represents the number of entities belonging to each cluster (e.g., Cluster-0 contains 16 entities). Also, to make the comparison between two different metrics easier using the top and bottom views, the same colors that represent the cluster IDs are used for the corresponding entities. For example, in Figure 6.5, the bottom view is colored based on the cluster IDs computed for the top view. We can see that green lines (i.e., Cluster-0) in both top and bottom views have high 'Sec. Rb.' and 'Net. Send.' values in many time points.

Since the data stream continuously feeds data for a new time point, visualizing all the fed time points consumes the visual space and, as a result, it becomes challenging to covey the detailed changes. Therefore, as new data arrive, we slide each view's time window for the visualization. Also, to provide the historical context with the past time points from the start of the simulation, as shown in Figure 6.5(b1, b2), the summary behavior placed at the bottom of each view shows the average value across entities for each time point.

6.4.2 Behavior Similarity Views

As discussed in Subsection 6.3.3, providing DR results along with cluster IDs can supplement analyzing temporal behaviors, such as finding outliers (**R2**). We apply the method described in Subsection 6.3.3 to multiple time-series shown in each performance behavior view (Figure 6.5(a1) and (a2)). Then, the behavior similarity views visualize the corresponding DR results (i.e., Figure 6.5(c1) and (c2) show the result from (a1) and (a2), respectively). In addition, to easily relate with the clustering results, we color the points with the corresponding colors used for encoding the cluster IDs in the performance behavior views. For example, all polylines and points shown in (a1, a2, c1, c2) are colored based on the clustering IDs obtained in (a1). From (c1), We can see that each entity's behavior of 'Sec. Rb.' is clearly separated into three clusters. However, in (c2), Cluster-0 (green) has four outliers as shown around the center. This can further be validated using the bottom behavior view (a2). The four green lines have much higher values when compared with the other green lines.

6.4.3 Metric Causality View

The metric causality view (Figure 6.5(d)) provides the results from the causality analysis described in Subsection 6.3.4 between the performance metrics (**R3**). This view can help the analyst determine which metrics share causal relationships with the chosen metric of interest for the top performance behavior view (Figure 6.5(a1)). Using Granger causality, we can derive two kinds of causal relationships: (1) from-causality: effects from other metrics on the metric of interest, and (2) to-causality: effect of metric of interest on other metrics. We display the from-causality results by default because the analyst often wants to review a metric which conveys the performance decrements first and then to identify which metrics likely cause it. However, the analyst can change to the to-causality results from the settings in Figure 6.5(f). We inform metrics of which p-value for Granger causality test is less than the user-defined value (the default is 0.05) with the yellow background. Also, we display the results of IR and VD. The metrics can be sorted based on the values for IR and VD by clicking the column title. Causality analysis results are useful for the analyst to decide the second metric to be shown in the bottom performance

behavior view (Figure 6.5(a2)). For example, in Figure 6.5, because we can see that 'Net. Send' has Granger causality and the highest IR value, we select and visualize its behavior in (a2).

6.4.4 Communication Behavior Views

The communication behavior views shown in Figure 6.5(e1, e2, e3) visualize the communication patterns between the entities (**R4**). I use an adjacency-matrix based visualization to show communications which can be represented as a weighted graph (refer to Section 6.1). Using the adjacency-matrix can provide flexibility to support any type of network topologies. While reviewing communications at the latest time point is not enough to grasp the changes in communication patterns, looking through communications at all the time points is not realistic in the streaming setting. Therefore, the module supports three different views to provide a summary of communication patterns and their changes. The details of each view are described below.

Live communication view (Figure 6.5(e1)) visualizes communications among entities during the sampling interval at the latest time point. While each row and each column of the matrix corresponds to one entity with the user-defined order, each cell's color represents the number of communications between the corresponding row and column. I use a gray sequential colormap (darker gray denotes higher communication). Additionally, to inform the cluster information of each entity, the cluster ID corresponding to each row/column is encoded as colored rectangles on the top and left sides of the adjacency matrix. For example, in Figure 6.5(e1), the entities in the green cluster placed at the first several rows and columns cause many communications within the entities in the same cluster.

To provide better scalability, this view can cooperate with the hierarchical information of entities, which often can be seen in HPC systems. For example, many HPC systems consist of multiple racks, compute nodes, CPUs, and cores. These have hierarchical relationships (e.g., a compute node contains multiple CPUs). The analyst can choose the granularity of entities (i.e., the level of hierarchy) to be shown in this view. When the lower granularity is selected instead of the original granularity, each cell shows the

average amount of communications within and among the groups of entities. For example, in Figure 6.5(e1), because many entities (128 entities) will make the available space for each cell small, we select one lower granularity consisting of 8 groups (i.e., each group contains 16 entities. This view also allows the analyst to show the communications with one higher granularity level by double-clicking the grouped cell.

Base communication view (Figure 6.5(e2)) visualizes the communication matrix at the user-selected time point. I use the same visualization methods as for the live communication view. The analyst can select the time point using a brown draggable vertical line placed on the summary behavior view.

Diff communication view (Figure 6.5(e3)) shows communications at multiple time points which are essential to understand the changes in communications. I use the change points identified by CPD as such essential time points because each change point shows significant changes from the previous time points. I order each matrix corresponding to each change point along the horizontal direction from left to right. Also, I indicate the corresponding change point with the numerical labels placed in Figure 6.5(a1, b1, e3).

To allow the analyst to compare the communications at each change point with the one at the selected time point for the base communication view (Figure 6.5(e2)), each matrix's cell shows the difference in the amount of communication between the selected time and each change point. As shown in the colormap placed in Figure 6.5(e3), the differences are encoded using a red-blue divergent colormap where the darker red and blue represent higher positive and negative values, respectively. The comparison with the selected base communications provides the flexibility in the analysis. For example, when the analyst selects a time point which has no communications (e.g., the start time point), the diff communication view visualizes the amounts of communications as they are. On the other hand, when the analyst selects one of the change points as a base, the view shows how the communications are changed from the selected change point. This helps understand which entities' communications affect the changes in the metric visualized in the performance behavior views.

6.4.5 User Interactions across Views

In addition to the interactions in each view, I provide several interactions that are linked to multiple views.

Pausing and resuming streaming updates. Since the visualization module updates its views as the analysis module sends new results, the updates occur at data collection rate at which the performance data is collected from the simulation. To provide a mechanism that allows the analyst to interact with the visualized results even when the data collection rate is high, I provide a *pause* button (Figure 6.5(g)) to pause the views from updating. Also, after pausing, the button is toggled to a *resume* button. When the *resume* button is clicked, the views return back to the current state of the simulation.

Selecting a time point of interest. As described in Subsection 6.4.4, the analyst can select a time point of interest in the summary of performance behavior shown in Figure 6.5(b1). Selecting a time point immediately updates the base and diff communication views.

Selecting entities of interest. Although a set of views can reveal overall patterns in performance and communication behaviors, the user would be interested in to analyze a subset of entities (e.g., a subcluster appeared in the behavior similarity views) in more detail. Therefore, I provide fundamental linking-and-brushing interactions. For example, the behavior similarity views support lasso selection to choose entities of interest. Once selected, the polylines in the performance behavior views belonging to the selected entities are highlighted by coloring the rest with gray and low opacity.

6.5 Case Studies

For demonstrating the applicability and effectiveness of the framework, I analyze the streaming data from parallel discrete-event simulation (PDES) along with the domain experts. In particular, I show that the visual analytics system can be used to effectively identify performance problems, investigate the source of the problem, and develop insights for improving performance. Besides, I collected feedback from the domain experts, which confirmed the usefulness of the framework and provided further design considerations.

PDES is used for studying complicated scientific phenomenon and systems. PDES typically runs on HPC systems for a long period using many compute nodes, and thus, simulations with low efficiency cost significant energy and time. Therefore, it is crucial to optimize performance and relieve bottlenecks. For distributed and parallel computing, PDES distributes a group of processes, called processing elements (PEs), that run across compute nodes. PEs communicate by exchanging time-stamped event messages that are processed to ensure the correct order of events (i.e., the future event must not affect the past event) [74]. As PDES, I use the Rensselaer's Optimistic Simulation System (ROSS) [75], an open-source discrete-event simulator. ROSS uses optimistic parallel event scheduling, where each PE contains a group of logical processes (LPs) that can independently process events to avoid frequent global synchronization with other PEs. Also, ROSS introduces kernel processes (KPs) for managing a shared processed event list for a collection of LPs mapped to a single PE, which has been demonstrated to process up to billions of events [188, 189]. When a KP rolls back, it must roll back all its LPs to the same point in virtual time. Therefore, having too many LPs mapped to a KP can result in performance degradation due to unnecessary rollbacks of other LPs in the same KP. Furthermore, the performance of ROSS depends on the model being simulated and the associated workload, which makes it difficult to investigate performance bottlenecks.

I set up the ROSS with the Dragonfly network [67] simulation model provided by the CODES [188]. The simulated Dragonfly network is similar to the network used by Theta Cray XC supercomputer at Argonne National Laboratory with 864 routers. The simulations are run with 8 to 16 PEs, where each PE has 16 KPs, with up to 16,384 LPs.

6.5.1 Monitoring Key Changes in PDES Performance

Because PDES uses a large number of entities in a complex structure, identifying the causes and sources of performance problems is difficult. With the visual analytics framework, we can effectively monitor the changes in the key metrics in real time, and analyze which computing entity is causing performance problems.

Since the efficiency of optimistic PDES depends on the number of rollbacks (including both primary and secondary rollbacks), we start to monitor each KP's 'Sec. Rb.' and

'Prim. Rb.' with the performance behavior views. We set the number of clusters to be 3, and the cluster IDs are calculated based on 'Sec. Rb.'. From monitoring the causality results, as shown in Figure 5.7(d), we notice that 'Net. Send.' has the Granger causality to 'Sec. Rb.' and has a large IR value. Hence, we update the second metric to 'Net. Send' and continue to study the influence of 'Net. Send' on 'Sec. Rb.'.

After the online CPD proceeded with several GVT intervals, we can see the first change point ① (refer to Figure 5.7(a1)). ① clearly shows that the 'Sec. Rb.' has drastically increased (Cluster-2 with the orange-colored polylines). This alerts us that we should closely monitor the rollback behaviors because the efficiency of the simulation can be significantly decreased if the number of secondary rollbacks stays high. Figure 5.7(a1) and (a2) show the result after the simulation is run for 75 GVT intervals. The online CPD detected four change points (①, ②, ③, and ④). While the metric causality view shows the detailed behaviors of each entity, the summary behavior view shows the trends of the PDES as a whole. As shown in (Figure 5.7(b1)), the average of secondary rollbacks have a peak at the start of the simulation close to ①, then reduces between ① and ②, and increases again from ② to ③, and maintains the values till ④. The summary view with CPD allows us to confirm the existence of performance problems in the simulations as the secondary rollbacks periodically increase at different points.

6.5.2 Tracing Performance Bottlenecks

If we can confirm the existence of performance problems, we can stop the simulation to prevent wasting time and energy by running the long and problematic PDES. Thus, we move on to the identification of the source of the problems, and the system can be used to trace the performance issues and bottlenecks. As shown in Figure 5.7(a1, a2), we can see that some of green-colored polylines (i.e., Cluster-0) continue to show relatively high values in both 'Sec. Rb.' and 'Net. Sends'. Thus, we further investigate the green cluster. By associating the behavior similarity view (Figure 5.7(c1)) to the performance behavior view (Figure 5.7(a1)), we realize that all the KPs belonging to the green cluster have a high similarity with each other for their rollback behavior. However, as shown in Figure 5.7(c2), for their 'Net. Send.', they are separated into one major cluster at the top

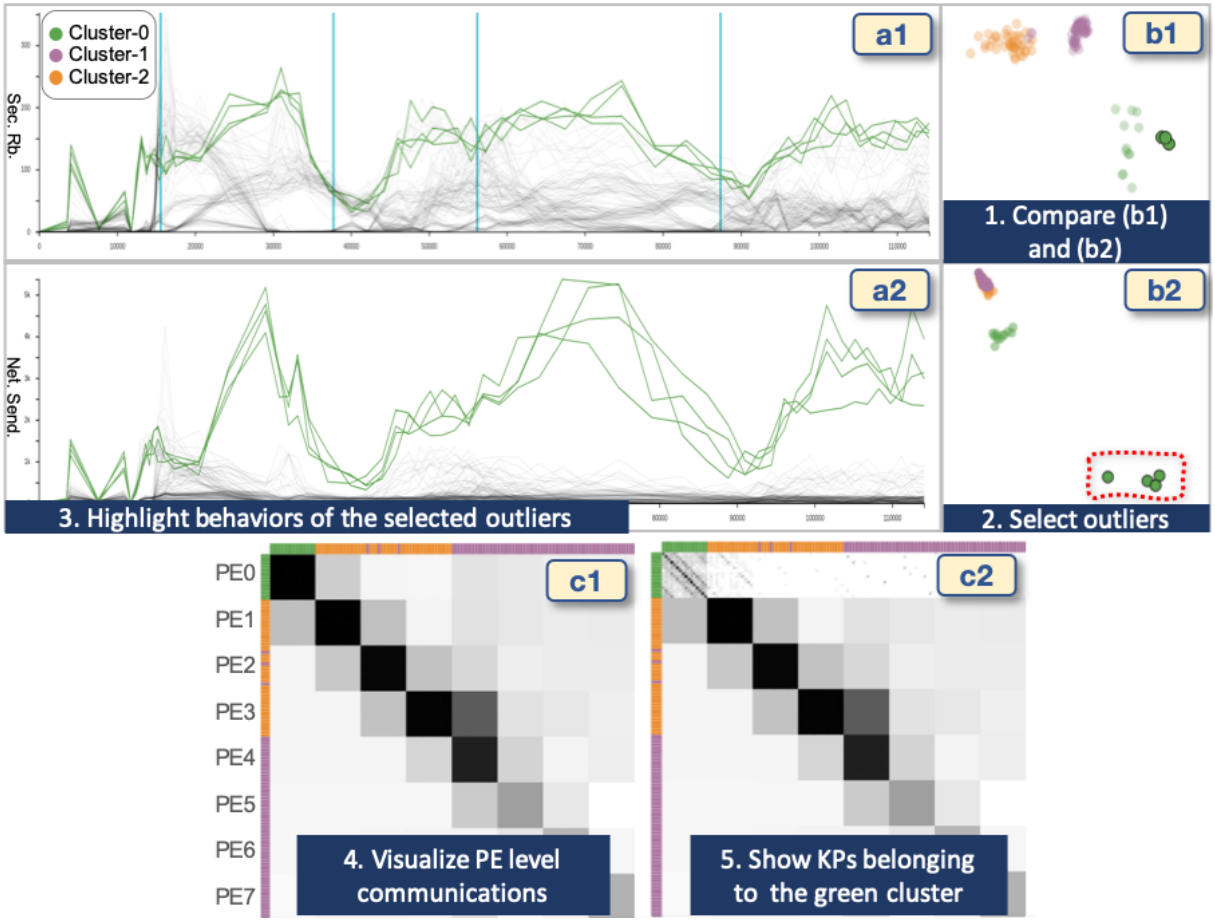


Figure 6.6: Detailed analysis of performance bottlenecks.

left and one small cluster that has four KPs at the right bottom. We can expect that these four KPs have dissimilar behaviors from the others. To further analyze their behaviors, we select the four entities with the lasso selection. The result is shown in Figure 6.6. From the performance behavior views (a1, a2), we can discern that these four KPs cause large numbers of network sends and, as an influence, they also cause large numbers of secondary rollbacks.

6.5.3 Analyzing Communication Patterns

After identifying the source of the performance problem, insights for removing the bottlenecks and optimizing the performance can be extracted from our visual analysis results. Since a LP that communicates with other LPs that have high number of rollbacks can increase its own chances to have rollbacks, a better control on the communication

events can reduce the number of rollbacks and improve efficiency. Therefore, we analyze the rollback behaviors with the communication patterns in the live communication view, as shown in Figure 6.6(c1, c2). (Figure 6.6(c1)) shows the communication between PEs or KPs with arrangements in rows and columns of the matrix based on their ranks. At default, the communications between the PEs are shown. We can see the cluster IDs of the KPs shown at the top and left of the matrix indicate that all the KPs in the green cluster (including the four KPs) belong to PE0. Also, we can see that KPs in the green cluster dominantly communicate with KPs within the same cluster. To further drill down to the KP level, we click on the matrix cell belonging to PE0 (Figure 6.6(c2)) and identify that all KPs, including the four KPs, generate many communications only within themselves (e.g., KP0 communicates to KP0). This indicates that LPs managed by these KPs have high communications with each other. Since these KPs has high number of rollbacks, they may cause other KPs communicating with them to have higher chances to have rollbacks. To avoid these rollbacks, the mapping from LPs to KPs should be changed to alleviate the unbalanced communications.

The case studies above show that the visual analytics framework can tackle the challenges of real-time performance monitoring and analysis of PDES. With effective supports for analyzing the streaming PDES data in real-time, the framework helps the analyst identify the time points and locations (e.g., PEs and KPs) that performance issues occurred. Once such a performance issue is realized, the analyst can stop the simulation to save energy and time. The analysis results also provide hints to the cause of the performance issues, allowing PDES developers to debug or optimize performance.

6.6 Summary

In this chapter, I develop a progressive visual analytic framework that helps gain insights on streaming performance data in real-time using algorithmic and visual analytics supports. I discuss the challenges arising from the increasing computational throughput of High-Performance Computing (HPC) systems compared to the more moderate growth in storage capacity. As a result, there is a growing disparity between the amount of data

generated by computations and the data that can be stored for analysis. To overcome this, the use of in-situ workflows, which integrate visualization into the simulation pipeline, have become popular. These workflows emphasize the need for real-time performance analysis. The framework consists of three main modules: data management, analysis, and interactive visualization. The data management module efficiently processes and combines data streams in real-time. It collects multivariate time-series and network communication data, performs analytical processing, and delivers the results as data streams to the other modules. The analysis module includes algorithms for real-time analysis of multivariate streaming data. Techniques such as change point detection, time-series clustering, dimensionality reduction, and causal relation analysis are employed to reveal temporal patterns and identify metrics' co-influences. The algorithms are designed to handle the high-speed data flow and provide up-to-date results. The framework's efficiency and effectiveness are demonstrated through performance evaluations and a comprehensive case study analyzing data from a parallel-discrete event simulator (PDES).

Chapter 7
DMV - A Unified Performance Analysis Framework for tracking Data Movement in the Hardware Domain

7.1 Introduction

With the end of Moore's law [190] and the increased flexibility of hardware accelerators [191] — *e.g.,* Graphics Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs) — the future of computing is rapidly transitioning towards the use of heterogeneous architectures [192, 193]. Today, heterogeneous computing is adopted extensively to push the frontiers of computational sciences in both the scale and complexity. In the current HPC landscape, large-scale supercomputers (161 of the current Top 500 [5]) increasingly rely on collaborative execution models [194], where computation is largely offloaded to accelerators (GPUs or FPGAs), whereas the CPUs manage non-portable computations, network communication, and workload distribution. Despite providing unprecedented capabilities to solve large and complex scientific computing problems [193, 195, 196], heterogenous architectures also introduce new challenges in effectively utilizing the available computing resources [197].

A significant roadblock in the widespread adoption of heterogeneous computing (in particular, GPUs) is the *copy-then-execute* programming model, where the onus of maintaining the often-complex data structure, data transfer, and explicit data migration falls on the application developers. With present-day accelerators constrained by memory (*e.g.,* 16 GB for NVIDIA's V100 GPU), leveraging the host's memory

(usually plenty) as an extension becomes a promising solution yet a challenging task. Especially with increased applications of large deep learning (DL) models, training DL models on GPU-accelerated systems becomes increasingly difficult as models no longer fit in a single GPU's memory [198]. With limited bandwidth for CPU–GPU communication (~32GB/s with PCIe), but better GPU–GPU bandwidth (~300GB/s with NvLink2)[1], careful compute orchestration and data management are imperative. Inefficient data management and movement can lead to unnecessary GPU idle times or lowered CPU/GPU utilization [97, 114], ultimately creating performance bottlenecks and mitigating the promised gains of accelerators. Broadly, data movement comprises direct transfers to/from the GPU device, and between libraries supporting computations. Therefore, it is critical to analyze and understand *data movement* (*e.g.,* CPU-GPU and GPU-GPU transfers) such that software applications are made to optimally utilize the heterogeneous execution model efficiently.

To harness the potential of GPU-accelerated HPC systems, developers utilize *performance analysis* to comprehend, explore, and optimize the data movement strategies. This is crucial because the performance of data-driven applications are becoming increasingly dependent on fast memory access, which can only be improved through efficient data locality mechanisms. Typically, developers perform runtime analysis by gathering performance metrics through profiling and tracing tools [11, 127, 128, 199]. The collected performance metrics (*e.g.,* execution runtime, calling contexts, page faults) are then attributed with specific code regions through performance visualization [125, 200]. Although runtime analysis aids in identifying time-consuming code sections, it falls short in diagnosing the impact of data movement on memory-bound applications. For instance, optimizing the performance of General Matrix Multiplication (GEMM) demands various optimizations like grid/block size and tiling strategies to maximize hardware utilization and adhere to physical memory constraints [110, 201]. Developers must *minimize unnecessary data movement operations* and *optimize the memory footprint* to achieve desired *scalability* and *performance*. Achieving this often requires domain experts

[1]This work primarily focuses on Nvidia's Volta architecture, which is a widely adopted GPU accelerator (as of 2023).

to experiment and evaluate several optimization strategies, under diverse conditions, such as data sizes, configurations, hyperparameters, network conditions, *etc.*. This necessitates tracking and analyzing data movement and runtime performance across numerous executions, which can be an even more formidable challenge.

To comprehensively investigate and analyze data movement within the heterogeneous paradigm, it is crucial to simultaneously explore the application behavior across the *Hardware*, *Application*, and *Communication* (HAC) domains [6]. As such, multiple state-of-the-art profiling and tracing tools need to be incorporated to capture all the relevant information in an easy-to-consume format. The sheer scale and complexity of the collected HAC data for large-scale applications make visualization indispensable for revealing and understanding key patterns. It turns out existing performance visualizations are limited to relating the bottlenecks to a single domain, leaving much to be desired for a full-scope study across all domains. For example, one might record the communications between the devices to understand data movement, but these data transfers are often not associated with the corresponding code or the hardware it utilizes. Moreover, no existing tool can help developers understand and correlate the performance across two or more domains. Consequently, the absence of appropriate tools to analyze, visualize, and monitor data movement hampers developer's ability to formulate strategies for reducing data movement overheads.

In this chapter, I present *Data Movement Visualized* (*DMV*) — a framework to track data movement across heterogeneous interfaces and understand their implication on performance trade-offs. I architect an API, *DMTracker*, enabling code-based instrumentation to track data movement across the HAC domains. *DMTracker* unifies the performance data across several state-of-the-art tools —CUPTI [202], Caliper [11], and hwloc [203] for C++ and Python applications that run on Nvidia GPUs. *DMTracker* also closely integrates with PyTorch API's to monitor and track data movement across MLOps. To facilitate a holistic exploration of data movement, I design a visual analytic interface, *DMVis*, which introduces four interconnected — *Hardware*, *Application*, *Communication*, *Summary* and *Ensemble* views. The hardware view summarizes the

hardware topology and the memory hierarchy of the allocated resources. By mapping the collected performance metrics, *DMVis* highlights where expensive data transfers are. The application view relates *bandwidth* and *latency* to the corresponding application code using a calling context tree (CCT) representation. The communication view summarizes the data transfers across the host–device and the device–device interfaces during the execution. Summary and ensemble view allows comparing and contrasting multiple executions in an ensemble for quick insights. I demonstrate the utility of the framework through two use cases which identify common types of data movement overheads (*e.g.,* frequent data transfers and alternating CPU/GPU transfers) and show if an application is compute- or memory-bound. *DMV* is released as an open-source, web-based tool to support performance analysis of the data movement across the HAC domains.

7.2 Domain Problem Characterization

Through collaboration with domain experts, we identified their primary challenges and derived their analysis goals, the current workflow, and the limitations therein. Based on this collaboration, we list out the requirements (**R1**–**R4**) for supporting performance analysis in their development workflow. To address these requirements, I develop *DMTracker* — a unified tracking interface (see Section 7.3) and *DMVis* — a visual analytic interface (see Section 7.4).

R1. Facilitate a simple API to collect performance data. The challenges of capturing, representing, and ingesting such performance data are compounded significantly when focusing simultaneously on the HAC domains, which requires utilizing several tools and libraries. Our collaborators expressed the need to have a simple and minimally invasive API that can orchestrate the different tools and represent the data compactly, offering a lightweight framework.

R2. Summarize the performance across the HAC domains. Efficient data management requires overlapping the data transfers with kernel computations using prefetching (for a single GPU) and communication patterns (for multiple GPUs). To determine if data orchestration is efficient, summarized performance from each of the

HAC domains is essential [6].

R3. Compare and contrast performance across multiple executions. Existing performance visualization tools generally focus on individual executions. However, comparing performance is integral when it comes to understanding different strategies (*e.g.,* explicit data transfer or unified memory), configurations (*e.g.,* single-GPU vs. multi-GPU), and experiments (*e.g.,* weak- and strong-scaling).

R4. Determine if an application is compute-bound or memory-bound. For domain experts, it is critical to ensure both *runtime* and *memory usage* are considered with equal importance in the performance analysis. Focusing on both aspects helps target suitable optimization efforts to improve performance.

7.3 *DMTracker* — A Unified Interface for Tracking Data Movement

DMTracker collects performance profiles and execution traces using four commonly used performance interfaces — NVIDIA's CUPTI [202] to collect GPU profiles and traces, Caliper [11] to collect CPU traces and Calling Context Trees (CCT), NVML[2] to monitor GPU devices (*e.g.,* power consumption), and hwloc [203] to determine the hardware topology. To facilitate a minimally invasive framework with a simple API (**R1**), we design *DMTracker* as a header-only C++ interface, offering easy integration into application codes, and that abstracts the complexity of handling these individual tools (refer Figure 7.1).

The abstraction is offered through the "`DMV` object" that allows a configurable interface to underlying libraries. For example, users can configure the vector `metrics` to request tracking of performance metrics, which are internally forwarded to CUPTI and Caliper (see default values in Table 7.1). Overall, *DMTracker* performs the data movement tracking in four phases — *setup*, *collect*, *categorize*, and *teardown*.

Setup: First, *DMTracker* gathers information about the underlying hardware topology requested by the application code. Domain experts commonly employ resource managers

[2]https://developer.nvidia.com/nvidia-management-library-nvml

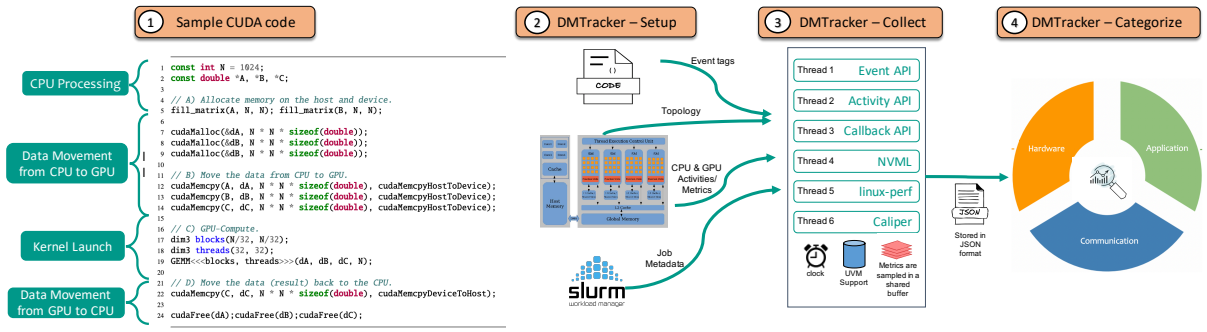


Figure 7.1: (1) A code snippet demonstrating data movement in a sample CUDA application code. (2) Setup phase involves collecting event tags, identifying topology, storing job metadata and setting up callbacks for activities and tracking for metrics. (3) Collect phase stores the events, activities, and metrics in a shared buffer (allocated for each GPU device). (4) Categorize phase aggregates the timeline of events according to the **tags** provided during code instrumentation.

and job schedulers (*e.g., slurm*) to allocate exclusive and/or non-exclusive access to computational resources. Although these schedulers can provide some monitoring capabilities, the hierarchy of the underlying topology and their metadata (*e.g.,* number of cores, size of caches) is usually not tracked. In other cases, applications may also allocate resources directly, in which case a more general solution to capturing the hardware topology is useful. *DMTracker* employs the `hwloc` library to identify all the connected components associated with the job execution (including the memory hierarchy). Additionally, *DMTracker* uses `cudaDeviceProp` to gather the metadata (*e.g.,* number of threads, blocks, SMs) for the GPUs and their corresponding interconnects (*i.e.,* PCIe, NvLink). To ensure all hardware components are captures, the user must

Domain	**Performance data**
Hardware	UUID, local device ID, global device ID, bus ID, CPU utilization (%), GPU utilization (%), Temperature (F), Power drawn (Watts), Memory used (MB), Memory free (MB), Clocks per SM (MHz)
Application	Correlation ID, Exclusive runtime, Inclusive runtime, Calling Context Tree, Page Faults, Active Warps, SM occupancy, Memory Bandwidth, Bank conflicts
Communication	Data sent/recv using PCIe, NVLink (MB), Bandwidth (MB/s)

Table 7.1: List of data collected by *DMTracker*.

```
1  #include "libdmv.h"
2
3  using namespace std
4
5  /* Initialize parameters for tracking metrics, activities and config */
6  vector<string> metrics = { "exclusive_runtime", "achieved_occupancy" ... };
7  vector<string> activities = { "CUPTI_ACTIVITY_KIND_PCIE", "CUPTI_ACTIVITY_KIND_NVLINK" ... };
8  unordered_map<string, float> config = { "sample_period_ms": 500.0, "warmup_ms": 3000 };
9
10 /* Setup the dmv context. */
11 auto &dmv = libdmv::DMV(activities, metrics, config);
```

Figure 7.2: **Setup API**: The "`DMV`" object takes in three input parameters: `metrics` to record performance metrics during code execution, `activities` to register callbacks into CUPTI Activity for asynchronous events, `config` to configure settings for *DMTracker* to track data movement.

perform all resource allocation before the *DMTracker*'s `start` function is triggered. "`DMV` object" takes in three input parameters: `activities`, `metrics`, and `config` (see line 14 in Figure 7.2). Once the "`DMV` object" is constructed, *DMTracker* allocates a buffer in the cache memory to collect the performance data stored.

Collect: Profiling and tracing across the HAC domains require separate tools, which incurs additional efforts from the user to orchestrate the data collection (**R1**). To simplify, *DMTracker* presents an integrated solution to capture the required performance data. *DMTracker* employs thread profiling, where a master thread spawns six worker threads across six interfaces — Activity API, Event API, Metrics API, NVML, linux-perf, and Caliper. Since different workers collect the performance data, *DMTracker* maintains an internal *clock* to ensure synchronization. Data movement tracking begins with the `start` function (see line 2 in Figure 7.3) and ends with the `stop` function (see line 9 in Figure 7.3). Additionally, the application code can be tagged by developers using "events" (through `start_event` and `stop_event`) to provide additional semantics for correlating the captured performance metrics during performance analysis (see line 5-6 in Figure 7.3). Below, we discuss how *DMTracker* collects performance data across the HAC domains.

Hardware: In addition to the hardware topology (captured during setup), *DMTracker* measures *utilization* and related metrics, such as temperature and power drawn, using `NVML` for GPUs and `linux-perf` for the CPUs at the requested sampling rate (default: 500 ms). *DMTracker* also tracks the memory usage across the topology to track if an application is memory-bound (**R4**). Adequate system-level permissions are necessary to facilitate the monitoring of all hardware components effectively.

Application: Using CUPTI's callback API, *DMTracker* registers callbacks to capture snapshots of activities from *device*, *context*, *stream*, *NVLink*, and *PCIe* during runtime. Each `activity` marks the entry and exit to the CUDA runtime and driver APIs. At the exit of each activity, *DMTracker* reads the event counters and metrics from CUPTI. We prefer the usage of CUPTI's callback and activity API since it allows asynchronous performance tracking for Nvidia GPUs. Similar to other tools that summarize the application domain, *DMTracker* collects runtime metrics (*i.e.,* inclusive and exclusive runtime), calling context trees (CCTs), and throughput (data consumed per second). *DMTracker* employs source-code instrumentation using Caliper to correlate the collected performance metric to the line of code. *DMTracker* adds an abstraction layer over CUDA runtime routines to automatically track the inclusive and exclusive runtimes. Additionally, users can add caliper instrumentation as desired. At the end of the execution, *DMTracker* reads the `caliper-json` output and appends the CCT (represented as nodes) to JSON.

Communication: DMTracker collects the communication traces exchanged between CPU–GPU and GPU–GPU asynchronously using the CUPTI Activity API. *DMTracker* records the amount of data transferred between the CPU and GPU interfaces (*i.e.,* NVLink and PCIe). Finally, *DMTracker* appends the bytes moved at the end of each activity along with their corresponding timestamps.

Categorize: For each `event` captured, *DMTracker* constructs a JSON object with the timestamp and the requested metadata. Depending on the context and device in which an `event` occurs, *DMTracker* categorizes events into four: (1) *CPU compute*, (2) *GPU compute*, (3) *Data movement*, and (4) *CUDA runtime*. The *CPU compute* category comprises events on the host that are *point-events* (*e.g.,* statements) or *range-events*

```
/* Start DMTracker tracking */
dmv.start();

/* Add custom event tags to track user events */
dmv.start_event("event_name", "event_tag"); // Optional
dmv.stop_event("event_name", "event_tag"); // Optional

/* End DMTracker tracking */
dmv.stop();
```

Figure 7.3: **Collect API**: *DMV* provides `start` and `stop` API functions to begin and end the data movement tracking. Users can also annotate specific sections of the code using `start_event` and `stop_event` functions.

(*e.g.,* loops, functions). The *GPU compute* includes kernel invocations and other operations performed on the GPU; these events can also be *point-events* (for asynchronous operations) or *range-events* (for synchronous operations). *Data movement* tracks the exchanges across the CPU–GPU and GPU–GPU interfaces through the PCIe and NVLink communication interfaces. Finally, the *CUDA runtime* category accounts for internal CUDA functions (all functions with prefix `cuda*`). This categorization effectively classifies a computation as either compute-bound or memory-bound (**R4**).

Teardown: Finally, *DMTracker* dumps the performance data from the buffer using the chrome trace JSON format. We chose this format to generalize to existing UI interfaces (such as UI Perfetto [3] and chrome tracing [4]) that also visualize the execution timeline.

[3] `https://ui.perfetto.dev`
[4] `chrome://tracing`

```
/* Dump the collected trace in the JSON format */
dmv.dump("/path/to/data", format="json");

/* Destroy the DMTracker context */
dmv.destroy();
```

Figure 7.4: **Teardown API**: *DMTracker* provides a `dump` function to store the collected traces in the JSON format. Finally, *DMTracker* instance can be safely destroyed using `destroy` function.

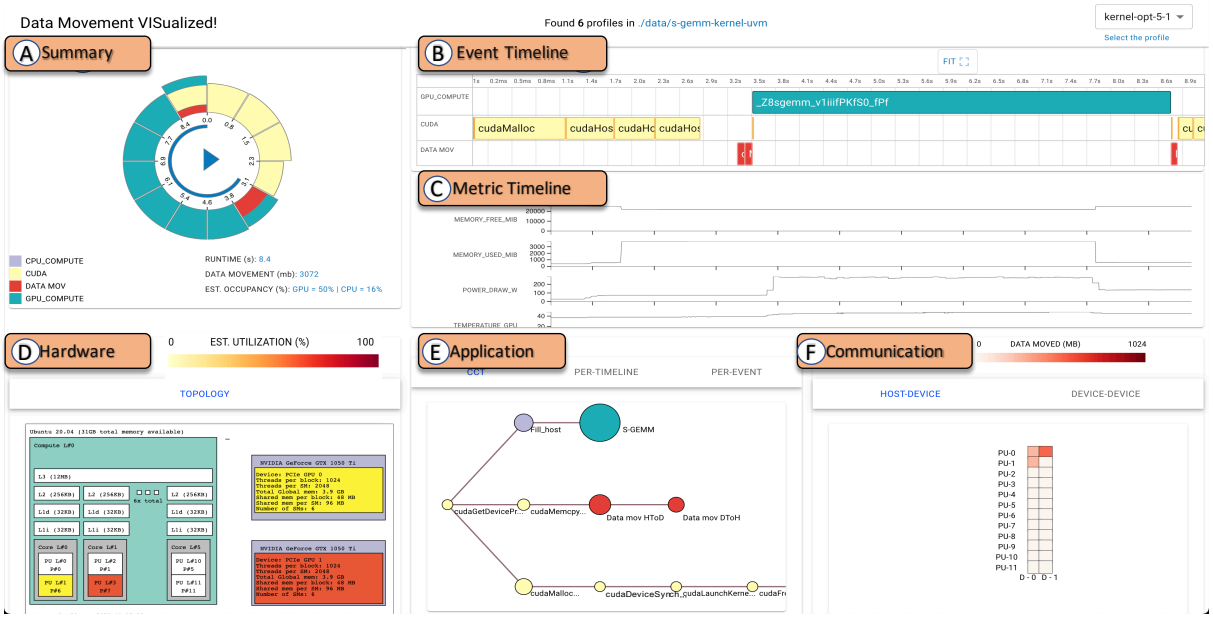


Figure 7.5: *DMVis* summarizes the data movement in an application run using a visual analytic interface. Users begin the performance analysis with the summary view (A). Users can play/pause the execution timeline. Event timeline (B) and Metric timeline (C) enable the analysis of timestamped traces. Hardware (D), Application (E), and Communication (F) views summarize the performance across the domains using linked views.

7.4 *DMVis* — Visual Analytic Interface

DMVis is a web-based visual analytic interface for studying the performance data collected by *DMTracker*. Users can load a directory containing the JSON files (one file per run) created by *DMTracker* to begin the performance analysis. Here, we discuss the visualization of a single profile first, followed by that of the ensemble. *DMVis* (see Figure 7.5) interface comprises of 5 views — *summary view*, *timeline view*, *hardware view*, *application view*, and *communication view*.

Summary view (see Figure 7.5Ⓐ) comprises of two visual glyphs — Utilization glyph(see Figure 7.6Ⓐ), Clock glyph(see Figure 7.6Ⓑ) to summarize the collected performance data from *DMTracker*.

Utilization glyph is introduced as a visual representation to effectively showcase spikes in CPU/GPU usage, providing valuable insights into the performance behavior of the application and enabling users to identify critical bottlenecks and performance

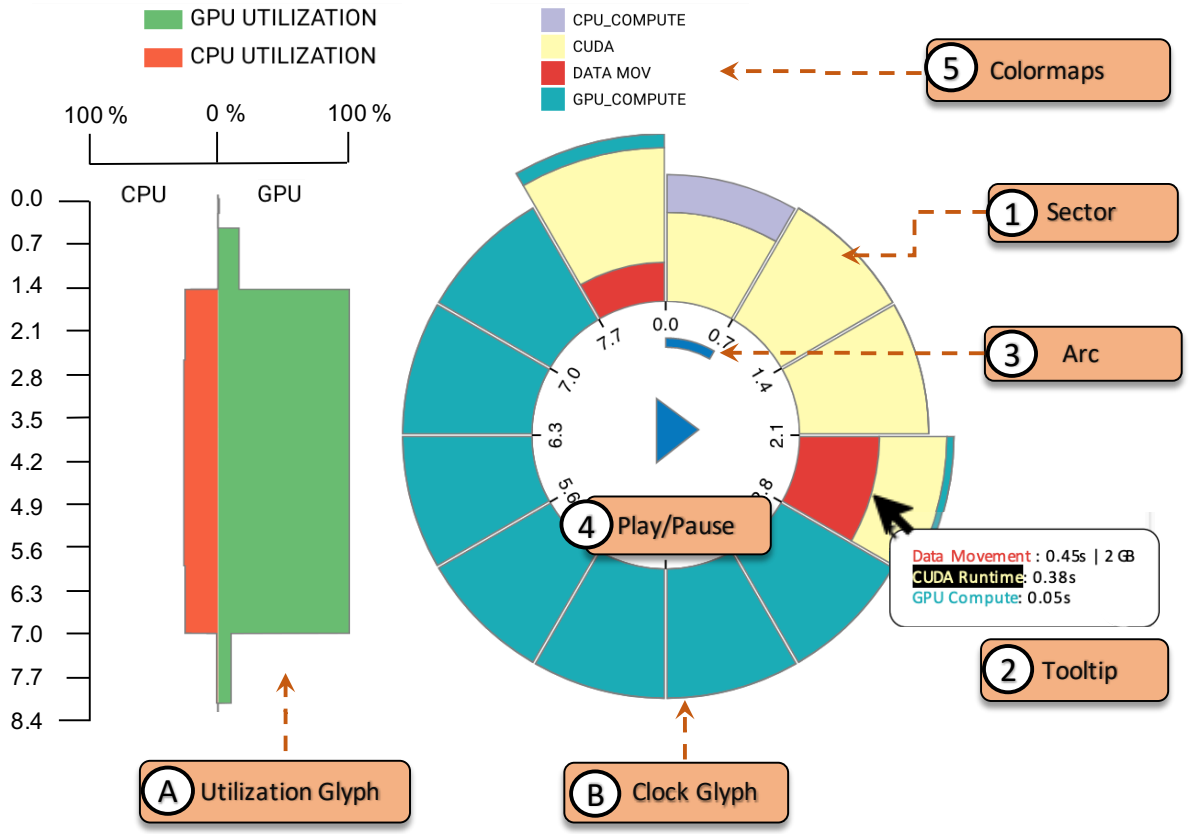


Figure 7.6: (a) Utilization glyph visualizes the CPU/GPU visualization during the execution timeline. (b) Summary glyph uses a "*clock*" metaphor to visualize the summary of the data movement across different `event tags` produced by *DMTracker*.

irregularities. We visualize the CPU- and GPU-utilization for the execution timeline as a vertical line graph, with CPU utilization on the left and GPU utilization on the right. To ensure flexibility, users may also choose any two metrics collected by DMTracker to encode the utilization glyph (refer Table 7.1).

Clock glyph presents a summary of the execution time based on the categorization provided by *DMTracker*— namely *CPU compute* (in purple), *GPU compute* (in green), *CUDA runtime* (in yellow), and *Data movement* (in red) (**R4**). To concisely represent large performance profiles, we present a visual design inspired by a *"clock"* — we do so because human brains are extremely well trained to interpret clocks instantly and correctly. *DMVis* applies a *clock* metaphor by splitting the entire execution timeline into 12 equal sectors. For each clock sector, *DMVis* calculates the contributions of the event tags and projects them as bars with the height representing the runtime of the

event (see Figure 7.6①). If multiple event tags are within a given sector, then the bars are stacked on top of each other. This facilitates the user to quickly identify sectors in the timeline that overlap between compute and memory operations (*e.g.,* see the overlap at four'o clock). Users can use the hover interaction to display the sector-wise breakdown using tooltips, which provides a textual breakdown of the *runtime*, *data movement*, and the *CPU/GPU occupancy* (see Figure 7.6②). Inside the clock, we embed an *arc* (see Figure 7.6③) to denote the current window and *play/pause functionality* (see Figure 7.6④) to navigate the timeline. When the user clicks the play button, the linked views update to summarize the performance of the current window. Finally, the summary view also shows the colormaps for the `tags` (see Figure 7.6⑤).

Timeline view (see Figure 7.5ⒷⒸ) presents a detailed view of all the `events` and `metrics` recorded by *DMTracker* as individual timelines. *DMVis* visualizes two kinds of timelines — (1) *event timeline* and (2) *metric timeline*.

Event timeline visualizes the *range-events* using rectangles (width corresponds to the duration of the event) and *point-events* with circles (with fixed radius). *DMVis* enables the user to interact with the execution using horizontal scrolling to zoom and drag to the regions of interest within the execution. When the window is adjusted, the HAC views automatically update to summarize performance for the selected region. For larger event timelines (with >1000 events), *DMVis* clusters the events based on context (*i.e.,* using the `correlationId`) to reduce the cognitive overhead of visually assessing a large number of events at once. Double-clicking a clustered event reveals the comprising events.

Metric timeline visualizes the trace data collected for each `metric` tracked by *DMTracker*. Since the number of metrics can be high depending on the analysis, we enable vertical scrolling for the metric timeline. To facilitate identifying a metric of interest, *DMVis* calculates the variance of the metric values in the current timeline window and automatically displays the metrics sorted by variance (high variance first).

Hardware view (see Figure 7.5Ⓓ) visualizes the topology collected from the *DMTracker*'s hardware interface (**R2**). *DMVis* encodes the estimated utilization for each component (CPU, GPU, and caches) from the topology using a white-red colormap(with

white signifying zero usage to red signifying 100% usage). Users can also use the full-screen mode button to view the topology with a better resolution. In the full-screen view, users can gather further information (*e.g.,* bandwidth, memory size) about each device in the topology.

Application view (see Figure 7.5Ⓔ) visualizes the CCT from the application code as a node-link diagram (**R2**). The nodes in the tree are colored based on `tag` so the users can relate the event timeline to their calling context. If a node is present in the current window, *DMVis* enables a pulsating animation to show the user regions of code that are active in the current window's context. The runtime and data movement statistics are also summarized as (a) *per-timeline* and (b) *per-event* using bar charts. These supplementary views help filter less significance nodes belonging to the corresponding event categories.

Communication view (see Figure 7.5Ⓕ) visualizes the communication involved between the components (*i.e.,* CPU-GPU and GPU-GPU) using a matrix visualization (**R2**). We choose a matrix visualization since large-scale machines typically have several hundreds of components and a node-link visualization will not be scalable. We color the matrix based on the amount of data moved by the component using a white-red colormap.

Ensemble view (see Figure 7.7) enables comparison of multiple executions by organizing each run's summary using a grid-based visualization (**R3**). If the directory loaded into *DMVis* has more than one JSON file, *ensemble view* serves to identify and select interesting executions to delve deeper into. The grid-based visualization lets users immediately visualize and compare the performance across different configurations and provides a high degree of flexibility to scale to larger ensembles. By clicking on a specific execution, users can explore it in detail. To facilitate the comparison of executions, the ensemble view supports two key interactions: (1) *sort* and (2) *compare*.

Sort (refer Figure 7.7Ⓐ) interaction allows users to order the executions based on their runtime. Executions with minimized runtime can be easily located at the top left of the grid, providing valuable insights into the optimal configurations.

Compare (refer Figure 7.7Ⓑ) interaction which scales the 12-hour clock to normalize sector widths to match the run with maximum runtime, making it easier to identify

similarities between two executions. This normalized view helps users identify common patterns or discrepancies in performance, making it easier to spot trends and anomalies.

Overall, the ensemble view empowers users to make informed decisions about their application's performance by offering a systematic and visual representation of multiple executions. It enables them to pinpoint significant variations in execution behavior, compare performance metrics, and identify factors that contribute to the application's efficiency or inefficiency under different settings. This functionality proves invaluable in optimizing applications and making data-driven decisions to improve overall performance and resource utilization.

7.5 Use Cases

DMV framework is first evaluated through case studies performed on CUDA-enabled applications. All measurements were obtained from the compute nodes, each of which has two CPUs (each comprising 20 processing units) and four 16 GB Nvidia V100 GPUs with NvLink connecting the GPUs. The case studies were conducted by collaborating with domain experts interested in identifying *where* and *why* data movements occur.

7.5.1 Case 1: GEMM on tall-and-skinny matrices

GEneral Matrix Multiplication (GEMM) is rich in data reuse, and improving the performance for "tall-and-skinny" matrices requires careful data management because it is memory-bound for both CPUs [204] and GPUs [205, 206]. For example, multiplying $m*k$ and $k*n$ matrices requires $m*n*(2k-1)$ floating point operations, so each element is accessed either $O(m)$ or $O(n)$ times. While coding a basic regular GEMM kernel (where $m == n$) is a fairly simple exercise, achieving high performance for irregular matrices requires much more insight into how the data is handled underneath. For a "tall-and-skinny" matrix (*i.e.,* $n << m, k$), each element in the input matrices is used $O(n)$ times on average. Depending on the size of n, GEMM can be either compute-bound (for large n) or memory-bound (for small n).

To perform the case study, we compare GEMM for matrices with $m = 65536$, $n = 65536$, and $k = 32$ matrices under five different memory allocation schemes —

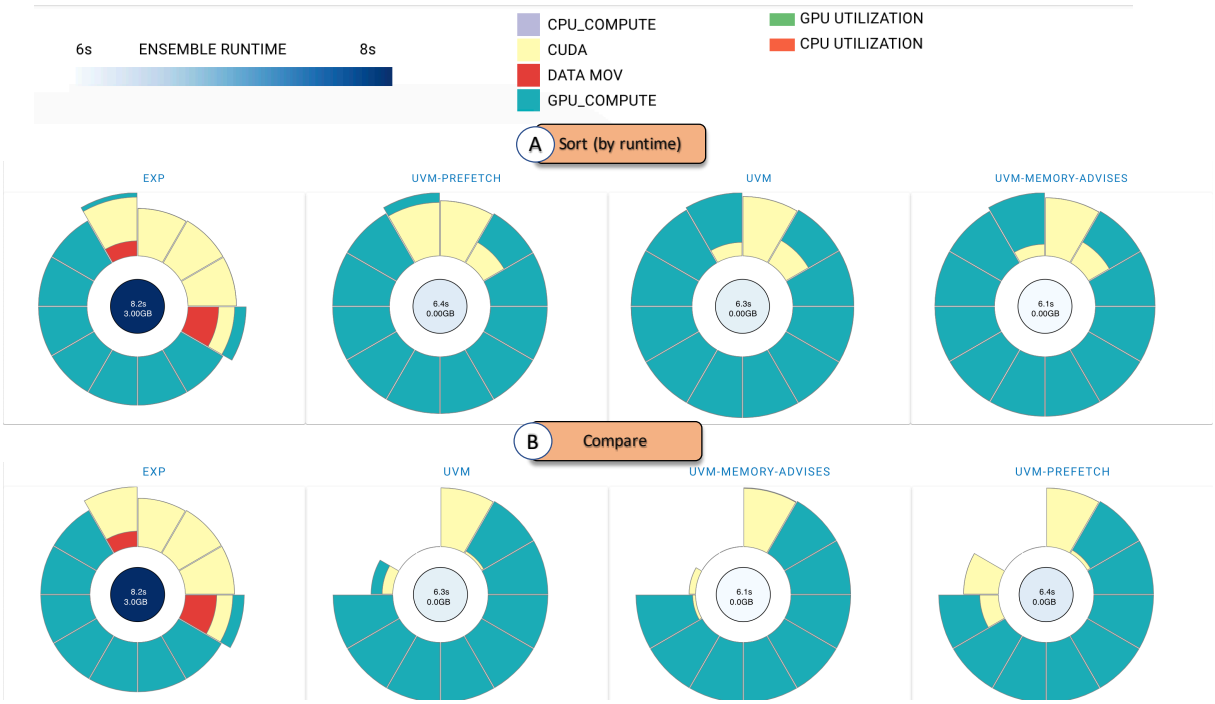


Figure 7.7: **GEMM on tall-and-skinny matrics**: We leverage the ensemble feature to investigate and compare the time spent across various unified memory management techniques. (A) Sorting the executions by total runtime, from maximum to minimum, highlights that explicit memory transfer is the most time-consuming approach. (B) By comparing the usage of Unified Virtual Memory (UVM) with memory advises and UVM with prefetching, we observe minor differences in the time distribution across CUDA.

explicit transfer (**EXP**), unified virtual memory (**UVM**), UVM with memory advise (**UVM-MA)**), and UVM with prefetching advise (**UVM-Prefetch**). We batch the matrix A ($m * k$) into multiple tiles (batch size=128) and perform a batched GEMM on a single node using `CUBLAS`. By dividing the entire matrix computation into multiple tiles, scalable performance improvement can be achieved. The study aims to determine which allocation strategy for matrix A leads to a performant GEMM kernel using *DMV* framework. To do this, we add `CUPTI_ACTIVITY_KIND_UVM` to track UVM-related performance metrics (*e.g.,* page faults and bank conflicts).

Using *DMVis*, we visualize the different allocation schemes using the ensemble view (see Figure 7.7). Comparing **EXP** with **UVM**, we can notice that a total of 3 GB data transfer occurs (see the 4'o clock and 11'o clock for **EXP**). However, there is no explicit data transfer in **UVM** since **CUDA** places the data in shared memory space.

For **UVM-MA**, we enable memory advises and set the matrix A and B as "read" mostly (using `cudaMemAdviseSetReadMostly`), while the matrix C is set to reside in the preferred location (using `cudaMemAdviseSetPreferredLocation`). Although memory advises improve the code quality by specifying the data locality, it does not provide any performance improvement in this scenario (performance decreases from 6.3s to 6.4s). This is mainly because memory advises are effective when both CPU and GPU are trying to access the data during GEMM kernel execution, but in this case, only GPUs are executing the matrix multiplication. Although **UVM** makes it easier for the developer, it incurs an extra penalty in resolving page faults. To avoid page faults, we prefetch the matrices A and B asynchronously for **UVM-prefetch**, which leads to reduced page faults and improves performance by 6% (from 6.3s to 6.1s). Overall, using the "clock" metaphor makes it easier to spot the regions of execution where the runtime is spent differently across executions.

7.5.2 Case 2: GPT-2 model training with TorchScript

With Deep Learning (DL) applications constantly increasing in size and complexity, runtime and/or data movement optimizations are critical in reducing the training and inference time. As DL models become larger, they exceed the memory limit of modern processors and require additional memory management techniques, such as checkpointing and hybrid learning techniques, like data- or task-parallel. Because most DL applications are developed using Python, incorporating efficient data management strategies can be challenging (because of the global interpreter lock). Recently, PyTorch TorchScript [5] introduced just-in-time (JIT) execution through a process called *tracing*. Tracing enables developers to extract computation graphs as an intermediate representation (IR) to enable further optimizations (*e.g.,* operator fusion). Once traced using `torch.jit.trace`, PyTorch outputs a *.pt* file that contains the optimized computation graph, which can be offloaded to the GPUs for parallel processing. Additionally, the JIT tracing allows dispatching execution to either the host or device (using `.to(device)`) and allows device pinning (explicitly transfer compute to device). In this case study, I was

[5] https://pytorch.org/docs/stable/jit.html

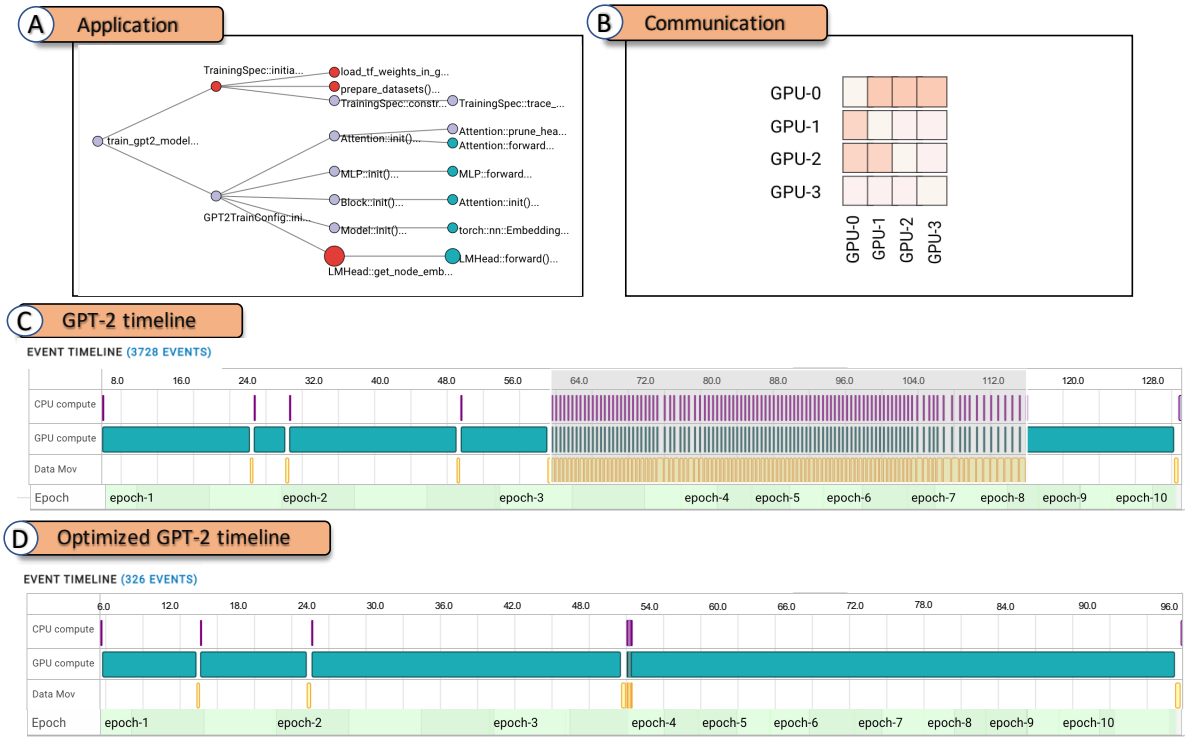


Figure 7.8: **GPT-2 model training**: Using the timeline view, I analyze the recurring events involving CPU, GPU, and data movement during GPT-2 training. Subsequently, I select specific regions in the timeline to establish correlations with the HAC domains (refer to A and B). This analysis enables us to optimize the application code effectively, leading to a reduction in repeated transfers between different contexts (as depicted in D).

interested in understanding the improvements with JIT tracing of publicly available GPT-2 model [207], which has approximately 1.5 billion parameters and fitting the entire model in a single GPU is not practically possible.

GPT-2 is a large transformer-based language model to predict the next word using a prompt provided by the user. The model comprises several attention heads (H_{att}), an embedding module, and a layer-mapping module (stores all the hidden states). GPT-2 abstracts model parallelism by distributing the attention heads across the GPUs, which can involve significant data movement while updating the model. Previous research has also exposed that transformer models do not fully utilize the GPU, and data movement is a key bottleneck in the training phase [102]. To track data movement with *DMTracker*, I write python bindings to the "`DMV` object" and place instrumentation in the TorchScript tracing module to record the start and end of CPU- and GPU-compute using the `device`

field in each tensor operation. Additionally tags were placed to mark the beginning and end of each epoch (in this case study, we train for 10 epochs) and use four GPUs for training with the following parameters (H_{att}=12).

From the execution timeline, we notice that the *DMTracker* recorded 3728 events. For this case study, we cluster the events based on data locality — CPU or GPU, where the data resides — to reduce the number of events shown in the timeline view. For each epoch, the JIT execution begins with JIT tracing (in purple), followed by the execution of tensors on the GPU-compute (in green), and finally, there is data movement to exchange the hyperparameters (in yellow). From the timeline view(see Figure 7.8(C)), we notice this pattern until epoch-4; after that, there are a lot of exchanges between the execution timelines (from 60s to 114s). From this, we can identify that data exchanges (as highlighted by grey selection) are a key bottleneck.

To identify the root cause of the bottleneck, we use the HAC views by using a brush operation to summarize the performance. From the application view (see Figure 7.8(A)), we notice the function `GPT2TrainLayerMapping::get_output_embeddings()` with the biggest node size in the CCT. From the communication view (see Figure 7.8(B)), we can see each communication event was alternating between the GPU fetch the output embeddings from the layer-mapping head (aka `LMHead`). To avoid this alternating pattern, we cache the output embeddings from the `LMHead` and only update the embeddings if the model changes the attention mask. After optimizing the training (see Figure 7.8(D)), we can notice that this reduces the communication between the CPU-GPU and improves runtime by 28% (with total training time of 98s for 10 epochs). The number of recorded events has also reduced to 328 events. Overall, this case study demonstrates that *DMVis* is able to identify communication patterns from the timeline view and relate them to the HAC domains, and facilitate optimization strategies.

7.6 Overhead Evaluation

Evaluating overhead is essential when developing *DMTracker* because it helps gauge the impact of the tracking and monitoring mechanisms on the application's performance.

By carefully measuring and analyzing the overhead introduced by *DMTracker*, we can ensure that its data collection and abstraction do not significantly affect the application's execution time or resource utilization. This assessment is vital to maintain the tool's minimally invasive nature, enabling developers to trust the accuracy of the captured performance metrics and effectively optimize their codes for better efficiency and performance.

To quantify the cost of using *DMV*, we measure the cost of executing different applications with/without using *DMTracker* (see Table 7.2). We conducted our analysis on popular CUDA kernels from the Rodinia benchmark [208] (*i.e.,* streamcluster, backprop, pathfinder, and srad) and GEMM. We selected kernels and problem sizes that produce modest-to-large memory footprints and involve significant memory-copy, representative of modern workloads for current heterogeneous architectures. We ran each kernel using both single and multi-GPU environments, collected the runtime overheads and averaged the costs over 100 iterations. To compare against a popular baseline, we profiled the application using Nvidia's open-source profiling tool, `nsys`. Finally, we varied the number of metrics recorded during experiments (25 vs 100 metrics) to evaluate the impact of tracing. For the multi-GPU experiments, we employ data parallelism where the input data is batched across the GPUs.

Comparison with `nsys`: We notice that applications have an overhead of 10-25% when using `nsys` tool. In comparison, *DMTracker* had a reduced overhead of 3-8% for 25 metrics. This is primarily attributed to *DMTracker*'s requirement to summarize the execution timeline across HAC domains, while `nsys` enables tracing at finer granularity (*i.e.,* with a higher sampling rate). However, this trade-off between overhead and trace granularity may be acceptable for many use cases, and the reduced overhead of *DMTracker* makes it a promising tool for profiling and optimizing CUDA applications.

***DMTracker* with 25 and 100 metrics**: When we increase the number of metrics, we observe that the overhead doubles from 3-8% to 8-15% across all applications. This is primarily because of the added tracing of GPU metrics. However, we also notice an increased variability across applications. For example, `srad` and `streamcluster` have the

Application	GPUs	runtime	w-nsys	O-nsys	w-dmv(25)	O-dmv(25)	w-dmv(100)	O-dmv(100)
Backprop	1	0.641	0.712	11.1	0.661	3.1	0.680	6.0
	4	0.41	0.487	18.7	0.441	7.5	0.454	10.7
PathFinder	1	0.63	0.703	11.6	0.653	3.4	0.684	7.8
	4	0.542	0.612	12.9	0.579	6.8	0.593	9.4
Srad	1	3.154	3.462	9.7	3.35	6.1	3.52	11.6
	4	0.737	0.871	18.1	0.79	8.2	0.842	14.2
Streamcluster	1	10.83	12.73	17.5	11.44	5.6	11.82	11.9
	4	3.92	4.9	25.1	4.23	7.4	4.45	13.5
GEMM	1	24.46	29.05	18.7	25.74	5.2	26.94	8.5
	4	6.48	7.67	18.3	6.83	5.4	7.23	11.5

Table 7.2: Performance overhead with different applications. All runtimes are presented in seconds (s). The overhead (O) is calculated by *((w DMV - w/0 DMV)/ (w/o DMV)) $\times$ 100)*.

highest overhead since they have a low computation-to-communication ratio and therefore have increased data movement.

Single- vs multi-GPU profiling: In our experiments, we observe that the overhead consistently (4.4% for `backprop`, 3.4% for `pathfinder`, 3.6% for `srad`, 1.6% for `streamcluster`, 3.0% for `GEMM`) increases for multi-GPU executions across all applications. This is because, in multi-GPU executions, *DMTracker* needs to synchronize the worker threads across the four GPUs, which introduces additional communication and synchronization overhead. Moreover, the degree of overhead increase varies depending on the specific application and input data size, but in general, multi-GPU executions seem to experience higher overhead than single-GPU executions even with `nsys`.

7.7 Summary

In this chapter, I presented Data Movement Visualized (DMV), a framework that tracks data movement across heterogeneous interfaces and analyzes its impact on performance trade-offs. The framework includes two tools: DMTracker, a unified data collection interface, and DMVis, a visual analytic interface. DMTracker facilitates performance data collection across CPU-GPU and GPU-GPU interfaces, while DMVis offers interactive performance analysis across the HAC domains. The significance of DMV lies in its ability to bridge performance analysis and data movement understanding, providing developers with valuable insights to optimize memory usage and improve overall

performance. Through case studies on the GEMM kernel and GPT-2 model training, DMV demonstrated effective performance optimizations achieved through careful data management, resulting in substantial runtime reductions. Beyond specific case studies, DMV's impact extends to various applications and domains, reducing overhead in data movement summarization by at least 20%. It opens up opportunities for further research and development, including support for additional heterogeneous architectures and interface types. Future improvements in DMVis will enhance usability and efficiency. In conclusion, DMV offers a valuable approach to understand data movement's effects on performance, with the potential to revolutionize performance analysis and optimization. With its insights, reduced overhead, and memory management benefits, DMV can advance high-performance computing and data-intensive applications.

Chapter 8
Conclusion

This dissertation tackles the difficulties associated with analyzing and establishing connections among performance data gathered in three crucial domains of high-performance computing (HPC): — *Hardware*, *Application*, *Communciation*. The research culminates in the development of visual analytic frameworks to significantly enhance the capabilities of HPC experts in comprehending and optimizing the performance of resource-intensive supercomputing applications.

First, I develop CallFlow, an interactive visual analytic tool that utilizes semantic operations to explore the *sampled profiles* representing the caller-callee relationships from the application domain. Modern scientific applications are built on top of rich frameworks and libraries that add layers of abstraction, leading to large-sized Calling Context Trees (CCT). Multiple layers of abstraction make *attribution* of performance bottlenecks associate with a CCT challenging. Additionally, domain experts conduct a variety of test configurations to identify optimal performance, resulting in an ensemble of sampled profiles. To address the attribution challenge, CallFlow introduces a new representation called *"super graphs"*, that can aggregate and split a CCT at user-controllable levels of abstraction to gain insights into expensive call sites. To *scale* the analysis for an ensemble of CCTs, CallFlow constructs *"ensemble super graphs"* that aggregate call paths across multiple CCTs and encode the differences between the CCTs to study performance variability.

Next, I design a visual analytic framework to support performance analysis in the communication domain by recording *event traces* at different time points of code execution. Grouping computing nodes based on the communication behaviors can reveal bottlenecks while studying different network topologies. However, a variety

of performance metrics require *correlation* for detecting bottlenecks from large-scale applications. To study communication behaviors, I integrate a analysis workflow to study multivariate performance metrics with coordinated communication views to reveal temporal behaviors using clustering algorithms and detect change points using time-series analysis. Finally, to reduce the high computational cost of analyzing large-scale applications, I extend the framework to support real-time analysis and monitor streaming data by adopting an in situ analysis workflow. Online algorithms provide scalable analysis by handling large data *velocities*, and progressive visualization techniques enable active user-engagement, perception, comprehension of performance behaviors.

Finally, I develop Data Movement Visualized (DMV) — a framework to track data movement across heterogeneous interfaces and understand their implication in the hardware domain. Handling data movement has become vital with the introduction of GPUs and FPGAs as compute-intensive operations demand expensive data movement across devices. To achieve good scalability and performance, one must minimize unnecessary data movement operations and data transfer volume between devices. *DMV* provides an unified data collection interface (*DMTracker*) for tracking the data movement across CPU–GPU and GPU–GPU interfaces and integrates a visual analytic interface (*DMVis*) to provide a holistic exploration of data movement across the HAC domains. I expose critical data access patterns and anti-patterns from the CPU-GPU interface by investigating the GEMM kernel under different memory allocation strategies and the training of the GPT-2 model. *DMV* helps identify performance optimizations through careful data management for code developers to better understand the resource utilization and derive strategies to improve performance with upcoming heterogeneous supercomputers.

As the future of performance analysis in HPC unfolds, developers will face the challenge of adapting their applications to leverage the latest hardware upgrades in supercomputers. Moreover, evolving software frameworks and libraries will add complexity, rendering manual performance analysis impractical. The transition to exascale computing will further amplify the volume and intricacy of performance

data, necessitating advanced visual analytic frameworks that will prove crucial in comprehending intricate performance patterns across the HAC domains. Successful progress in this will hinge upon collaborative efforts among computer scientists, domain experts, and tool developers to design sophisticated solutions capable of managing data scale and empowering users to effectively optimize HPC applications in the exascale era and beyond.

Bibliography

[1] G. Flato, J. Marotzke, B. Abiodun, P. Braconnot, S. C. Chou, W. Collins, P. Cox, F. Driouech, S. Emori, V. Eyring *et al.*, "Evaluation of climate models," in *Climate change: the physical science basis.* Cambridge University Press, 2014, pp. 741–866.

[2] Y. Cui, E. Poyraz, K. B. Olsen, J. Zhou, K. Withers, S. Callaghan, J. Larkin, C. Guest, D. Choi, A. Chourasia *et al.*, "Physics-based seismic hazard analysis on petascale heterogeneous supercomputers," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 2013, pp. 1–12.

[3] D. Günther, R. A. Boto, J. Contreras-Garcia, J.-P. Piquemal, and J. Tierny, "Characterizing molecular interactions in chemical systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2476–2485, 2014.

[4] M. Smith and J. C. Smith, "Repurposing therapeutics for COVID-19: Supercomputer-based docking to the SARS-CoV-2 viral spike protein and viral spike protein-human ACE2 interface," *ChemRxiv*, 2020.

[5] "Top 500 supercomputer list," https://www.top500.org/lists/top500/2020/11/, accessed: 01/2021.

[6] M. Schulz, J. A. Levine, P.-T. Bremer, T. Gamblin, and V. Pascucci, "Interpreting performance data across intuitive domains," in *International Conference on Parallel Processing.* IEEE, 2011, pp. 206–215.

[7] 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.

[8] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[9] 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." in *EuroVis (STARs)*, 2014.

[10] 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, 2008, pp. 139–155.

[11] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* IEEE Press, 2016, p. 47.

[12] G. T. Chetsa, L. Lefèvre, J.-M. Pierson, P. Stolf, and G. Da Costa, "Exploiting performance counters to predict and improve energy performance of HPC systems," *Future Generation Computer Systems*, vol. 36, pp. 287–298, 2014.

[13] 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.

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

[15] 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.

[16] M. Geimer, F. Wolf, B. J. Wylie, E. Ábrahám, D. Becker, and B. Mohr, "The scalasca performance toolset architecture," *Concurrency and computation: Practice and experience*, vol. 22, no. 6, pp. 702–719, 2010.

[17] 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.

[18] J. Abello, F. Van Ham, and N. Krishnan, "ASK-GraphView: A large scale graph visualization system," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 669–676, 2006.

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

[20] Q. V. Nguyen and M. L. Huang, "A space-optimized tree visualization," in *IEEE Symposium on Information Visualization*, 2002, pp. 85–92.

[21] C. Plaisant, J. Grosjean, and B. B. Bederson, "Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation," in *Proceedings of the IEEE Symposium on Information Visualization*, 2002, pp. 57–64.

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

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

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

[25] 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," *Information Visualization*, vol. 4, no. 2, pp. 114–135, 2005.

[26] 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," *Computer Graphics Forum*, vol. 37, no. 3, pp. 561–572, 2018.

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

[28] 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 *Proceedings of the 3rd International Workshop on Visual Performance Analysis*. IEEE Press, 2016, pp. 25–28.

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

[30] 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 Transactions on Visualization and Computer Graphics*, 2019.

[31] 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.

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

[33] B. Shneiderman and M. Wattenberg, "Ordered treemap layouts," in *Proceedings of the IEEE Symposium on Information Visualization*, 2001, pp. 73–78.

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

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

[36] F. Balmas, "Displaying dependence graphs: a hierarchical approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 3, pp. 151–185, 2004.

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

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

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

[40] H. Alemasoom, F. Samavati, J. Brosz, and D. Layzell, "Energyviz: An interactive system for visualization of energy systems," *The Visual Computer*, vol. 32, pp. 403–413, 2016.

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

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

[43] 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 International Conference on Bioinformatics and Biomedicine*, 2014, pp. 521–528.

[44] J. Zhao, F. Chevalier, C. Collins, and R. Balakrishnan, "Facilitating discourse analysis with interactive visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2639–2648, 2012.

[45] H.-J. Schulz, T. Nocke, M. Heitzler, and H. Schumann, "A design space of visualization tasks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2366–2375, 2013.

[46] T. Munzner, F. Guimbretière, S. Tasiran, L. Zhang, and Y. Zhou, "TreeJuxtaposer: Scalable tree comparison using Focus+ Context with guaranteed visibility," in *ACM Transactions on Graphics*, vol. 22, no. 3. ACM, 2003, pp. 453–462.

[47] M. Meyer, T. Munzner, and H. Pfister, "MizBee: A multiscale synteny browser," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 897–904, 2009.

[48] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.

[49] M. Graham and J. Kennedy, "A survey of multiple tree visualisation," *Information Visualization*, vol. 9, no. 4, pp. 235–252, 2010.

[50] M. Gleicher, D. Albers, R. Walker, I. Jusufi, C. D. Hansen, and J. C. Roberts, "Visual comparison for information visualization," *Information Visualization*, vol. 10, no. 4, pp. 289–309, 2011.

[51] D. Holten and J. J. Van Wijk, "Visual comparison of hierarchically organized data," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 759–766.

[52] M. M. Malik, C. Heinzl, and M. E. Groeller, "Comparative visualization for parameter studies of dataset series," *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 5, pp. 829–840, 2010.

[53] N. Amenta and J. Klingner, "Case study: Visualizing sets of evolutionary trees," in *Symposium on Information Visualization*. IEEE, 2002, pp. 71–74.

[54] J. Y. Hong, J. D'Andries, M. Richman, and M. Westfall, "Zoomology: comparing two large hierarchical trees," *Proceedings of the IEEE InfoVis Poster Compendium*, pp. 120–121, 2003.

[55] S. Bremm, T. von Landesberger, M. Heß, T. Schreck, P. Weil, and K. Hamacherk, "Interactive visual comparison of multiple trees," in *Conference on Visual Analytic Science and Technology*. IEEE, 2011, pp. 31–40.

[56] S. Fu, H. Dong, W. Cui, J. Zhao, and H. Qu, "How do ancestral traits shape family trees over generations?" *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 205–214, 2017.

[57] Z. Liu, S. H. Zhan, and T. Munzner, "Aggregated dendrograms for visual comparison between many phylogenetic trees," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 9, pp. 2732–2747, 2019.

[58] G. Li, Y. Zhang, Y. Dong, J. Liang, J. Zhang, J. Wang, M. J. McGuffin, and X. Yuan, "Barcodetree: Scalable comparison of multiple hierarchies," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 1, pp. 1022–1032, 2019.

[59] Z. Vosough, D. Kammer, M. Keck, and R. Groh, "Visualization approaches for understanding uncertainty in flow diagrams," *Journal of Computer Languages*, vol. 52, pp. 44–54, 2019.

[60] K. Williams, A. Bigelow, and K. Isaacs, "Visualizing a moving target: A design study on task parallel programs in the presence of evolving data and concerns," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 1118–1128, 2019.

[61] K. Andrews, M. Wohlfahrt, and G. Wurzinger, "Visual graph comparison," in *Internations Conference in Information Visualization*. IEEE, 2009, pp. 62–67.

[62] J. Schmidt, "Scalable comparative visualization - visual analysis of local features in different dataset ensembles," Ph.D. dissertation, TU Delft, 2016.

[63] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[64] C. E. Leiserson, "Fat-trees: universal networks for hardware-efficient supercomputing," *IEEE Transactions on Computers*, vol. 100, no. 10, pp. 892–901, 1985.

[65] F. Petrini and M. Vanneschi, "k-ary n-trees: High performance networks for massively parallel architectures," in *Proceedings of the 11th International Parallel Processing Symposium*. IEEE, 1997, pp. 87–93.

[66] K. M. Iftekharuddin and M. A. Karim, "Butterfly interconnection network: design of multiplier, flip-flop, and shift register," *Applied optics*, vol. 33, no. 8, pp. 1457–1462, 1994.

[67] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proceedings of the International Symposium on Computer Architecture*. IEEE, 2008, pp. 77–88.

[68] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken *et al.*, "Blue gene/l torus interconnection network," *IBM Journal of Research and Development*, vol. 49, no. 2.3, pp. 265–276, 2005.

[69] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff, "MPI on a million processors," in *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 2009, pp. 20–30.

[70] B. Van Straalen, J. Shalf, T. Ligocki, N. Keen, and W.-S. Yang, "Scalability challenges for massively parallel AMR applications," in *IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–12.

[71] K. Mohror and K. L. Karavanic, "Evaluating similarity-based trace reduction techniques for scalable performance analysis," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–12.

[72] K. E. Isaacs, P.-T. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann, "Combing the communication hairball: Visualizing parallel execution traces using logical time," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2349–2358, 2014.

[73] T. Fujiwara, P. Malakar, K. Reda, V. Vishwanath, M. E. Papka, and K.-L. Ma, "A visual analytics system for optimizing communications in massively parallel applications," in *Proceedings of the IEEE Conference on Visual Analytics Science and Technology*, 2017, pp. 59–70.

[74] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, 1990.

[75] C. D. Carothers, D. Bauer, and S. Pearce, "ROSS: A high-performance, low-memory, modular time warp system," *Journal of Parallel and Distributed Computing*, vol. 62, no. 11, pp. 1648–1669, 2002.

[76] C. Ross, C. D. Carothers, M. Mubarak, P. Carns, R. Ross, J. K. Li, and K.-L. Ma, "Visual data-analytics of large-scale parallel discrete-event simulations," in *Proceedings of the International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. IEEE, 2016, pp. 87–97.

[77] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 3, pp. 404–425, 1985.

[78] C. Sigovan, C. W. Muelder, and K.-L. Ma, "Visualizing large-scale parallel communication traces using a particle animation technique," *Computer Graphics Forum*, vol. 32, no. 3pt2, pp. 141–150, 2013.

[79] C. Muelder, B. Zhu, W. Chen, H. Zhang, and K.-L. Ma, "Visual analysis of cloud computing performance using behavioral lines," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 6, pp. 1694–1704, 2016.

[80] T. Fujiwara, J. K. Li, M. Mubarak, C. Ross, C. D. Carothers, R. B. Ross, and K.-L. Ma, "A visual analytics system for optimizing the performance of large-scale networks in supercomputing systems," *Visual Informatics*, vol. 2, no. 1, pp. 98–110, 2018.

[81] A. Dasgupta, D. L. Arendt, L. R. Franklin, P. C. Wong, and K. A. Cook, "Human factors in streaming data analysis: Challenges and opportunities for information visualization," *Computer Graphics Forum*, vol. 37, no. 1, pp. 254–272, 2018.

[82] M. Krstajic and D. A. Keim, "Visualization of streaming data: Observing change and context in information visualization techniques," in *Proceedings of the IEEE International Conference in Big Data*, 2013, pp. 41–47.

[83] P. Xu, H. Mei, L. Ren, and W. Chen, "ViDX: Visual diagnostics of assembly line performance in smart factories," *IEEE Transactions of Visualization and Computer Graphics*, vol. 23, no. 1, pp. 291–300, 2017.

[84] S. Cheng, K. Mueller, and W. Xu, "A framework to visualize temporal behavioral relationships in streaming multivariate data," in *Proceedings of the New York Scientific Data Summit*. IEEE, 2016, pp. 1–10.

[85] K. Webga and A. Lu, "Discovery of rating fraud with real-time streaming visual analytics," in *Proceedings of the IEEE VizSec*, 2015, pp. 1–8.

[86] C. C. Aggarwal, "A survey of stream clustering algorithms," in *Data Clustering*. Chapman and Hall/CRC, 2013, pp. 231–258.

[87] C. Turkay, N. Pezzotti, C. Binnig, H. Strobelt, B. Hammer, D. A. Keim *et al.*, "Progressive data science: Potential and challenges," *arXiv preprint:1812.08032*, 2018.

[88] T. Mühlbacher, H. Piringer, S. Gratzl, M. Sedlmair, and M. Streit, "Opening the black box: Strategies for increased user involvement in existing algorithm implementations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1643–1652, 2014.

[89] C. D. Stolper, A. Perer, and D. Gotz, "Progressive visual analytics: User-driven visual exploration of in-progress analytics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 1653–1662, 2014.

[90] C. Turkay, E. Kaya, S. Balcisoy, and H. Hauser, "Designing progressive and interactive analytics processes for high-dimensional data analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 131–140, 2017.

[91] H. Strange and R. Zwiggelaar, *Open Problems in Spectral Dimensionality Reduction*. Springer, 2014.

[92] N. Pezzotti, B. P. Lelieveldt, L. van der Maaten, T. Höllt, E. Eisemann, and A. Vilanova, "Approximated and user steerable tSNE for progressive visual analytics," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 7, pp. 1739–1752, 2017.

[93] L. van der Maaten and G. Hinton, "Visualizing data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[94] T. Fujiwara, J.-K. Chou, S. Shilpika, P. Xu, L. Ren, and K.-L. Ma, "An incremental dimensionality reduction method for visualizing streaming multidimensional data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 26, no. 1, pp. 418–428, 2019.

[95] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang, "Incremental learning for robust visual tracking," *Int. Journal of Computer Vision*, vol. 77, no. 1-3, pp. 125–141, 2008.

[96] T. Gysi, T. Grosser, and T. Hoefler, "Modesto: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, 2015, pp. 177–186.

[97] D. Unat, A. Dubey, T. Hoefler, J. Shalf, M. Abraham, M. Bianco, B. L. Chamberlain, R. Cledat, H. C. Edwards, H. Finkel *et al.*, "Trends in data locality abstractions for HPC systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 10, 2017.

[98] P. Cicotti, S. Oral, G. Kestor, R. Gioiosa, S. Strande, M. Taufer, J. H. Rogers, H. Abbasi, J. Hill, and L. Carrington, "Data movement in data-intensive high performance computing," in *Conquering Big Data with High Performance Computation*. Springer, 2016.

[99] M. E. Belviranli, F. Khorasani, L. N. Bhuyan, and R. Gupta, "CUMAS: Data transfer aware multi-application scheduling for shared GPUs," in *Proceedings of the International Conference on Supercomputing*, 2016.

[100] T. Gysi, J. Bär, and T. Hoefler, "dCUDA: hardware supported overlap of computation and communication," in *IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016.

[101] T. Ben-Nun, J. de Fine Licht, A. N. Ziogas, T. Schneider, and T. Hoefler, "Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019, pp. 1–14.

[102] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proceedings of the Machine Learning and Systems*, vol. 3, pp. 711–732, 2021.

[103] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[104] C. Phuong, N. Saied, and C. Tanis, "Assessing kokkos performance on selected architectures," in *Latin American High Performance Computing Conference*. Springer, 2019, pp. 170–184.

[105] D. A. Beckingsale, J. Burmark, R. Hornung, H. Jones, W. Killian, A. J. Kunen, O. Pearce, P. Robinson, B. S. Ryujin, and T. R. Scogland, "RAJA: Portable performance for large-scale scientific applications," in *IEEE/ACM International workshop on Performance, Portability and Productivity in HPC (p3hpc)*. IEEE, 2019, pp. 71–81.

[106] D. A. Beckingsale, M. J. Mcfadden, J. P. Dahm, R. Pankajakshan, and R. D. Hornung, "Umpire: Application-focused management and coordination of complex

hierarchical memory," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 00–1, 2019.

[107] T. Allen and R. Ge, "In-depth analyses of unified virtual memory system for GPU accelerated computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021.

[108] C. Shao, J. Guo, P. Wang, J. Wang, C. Li, and M. Guo, "Oversubscribing GPU unified virtual memory: Implications and suggestions," in *Proceedings of the ACM/SPEC on International Conference on Performance Engineering*, 2022, pp. 67–75.

[109] S. Chien, I. Peng, and S. Markidis, "Performance evaluation of advanced features in cuda unified memory," in *IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. IEEE, 2019, pp. 50–57.

[110] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched gemm for GPUs," in *International Conference on High Performance Computing*, 2016.

[111] T. Allen and R. Ge, "Demystifying GPU UVM cost with deep runtime and workload analysis," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 141–150.

[112] Z. Jin and J. S. Vetter, "Evaluating unified memory performance in hip," in *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2022, pp. 562–568.

[113] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: a GPU memory manager with application-transparent support for multiple page sizes," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 136–150.

[114] C.-C. Huang, G. Jin, and J. Li, "Swapadvisor: Pushing deep learning beyond the GPU memory limit via smart swapping," in *Proceedings of the 25th International Conference on Architecture Support for Programming Languages and Operating Systems*, 2020.

[115] H. Xu, P.-H. Lin, M. Emani, L. Hu, and C. Liao, "Xunified: A framework for guiding optimal use of GPU unified memory," *IEEE Access*, vol. 10, pp. 82 614–82 625, 2022.

[116] S. Go, H. Lee, J. Kim, J. Lee, M. K. Yoon, and W. W. Ro, "Early-adaptor: An adaptive framework forproactive UVM memory management," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 248–258.

[117] T. Mu, J. Tao, M. Schulz, and S. A. McKee, "Interactive locality optimization on NUMA architectures," in *Proceedings of the ACM symposium on Software visualization*, 2003, pp. 133–ff.

[118] A. Giménez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P.-T. Bremer, and B. Hamann, "Memaxes: Visualization and analytics for characterizing complex memory performance behaviors," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 7, pp. 2180–2193, 2017.

[119] A. F. Blanco, A. Bergel, and J. P. S. Alcocer, "Software visualizations to analyze memory consumption: A literature review," *ACM Computing Surveys (CSUR)*, vol. 55, no. 1, pp. 1–34, 2022.

[120] C. Sigovan, C. Muelder, K.-L. Ma, J. Cope, K. Iskra, and R. Ross, "A visual network analysis method for large-scale parallel I/O systems," in *IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 308–319.

[121] A. G. Landge, J. A. Levine, A. Bhatele, K. E. Isaacs, T. Gamblin, M. Schulz, S. H. Langer, P.-T. Bremer, and V. Pascucci, "Visualizing network traffic to understand the performance of massively parallel simulations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 12, pp. 2467–2476, 2012.

[122] E. E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. L. Su, and S. Z. Guyer, "Heapviz: interactive heap visualization for program understanding and debugging," in *Proceedings of the 5th International Symposium on Software Visualization*, 2010, pp. 53–62.

[123] B. Quaing, J. Tao, and W. Karl, "Yaco: A user conducted visualization tool for supporting cache optimization," in *High Performance Computing and Communications (HPCC)*. Springer, 2005, pp. 694–703.

[124] S. Moreta and A. Telea, "Visualizing dynamic memory allocations," in *Proceedings of the 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2007, pp. 31–38.

[125] P. Rosen, "A visual approach to investigating shared and global memory behavior of cuda kernels," in *Computer Graphics Forum*, vol. 32, no. 3pt2. Wiley Online Library, 2013, pp. 161–170.

[126] A. Ariel, W. W. Fung, A. E. Turner, and T. M. Aamodt, "Visualizing complex dynamics in many-core accelerator architectures," in *IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*. IEEE, 2010, pp. 164–174.

[127] NVIDIA, "NSYS: Nvidia nsight systems," https://developer.nvidia.com/nsight-systems.

[128] AMD, "Rocprofiler: a low-level performance analysis api for profiling GPU compute applications," https://docs.amd.com/bundle/AMD-ROCProfiler-User-Guide-v5.1/page/ROCProfiler_Profiling_API.html.

[129] P. Kousha, B. Ramesh, K. Kandadi Suresh, C.-H. Chu, A. Jain, N. Sarkauskas, H. Subramoni, and D. K. Panda, "Designing a profiling and visualization tool for scalable and in-depth analysis of high-performance GPU clusters," in *Proceedings of the IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, 2019, pp. 93–102.

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

[131] 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.

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

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

[134] S. Tilkov and S. Vinoski, "Node.js: Using JavaScript to build high-performance network programs," *IEEE Internet Computing*, vol. 14, no. 6, pp. 80–83, 2010.

[135] 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.

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

[137] 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.

[138] 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 Transactions on Visualization and Computer Graphics*, pp. 1–1, 2019.

[139] P. Riehmann, M. Hanfler, and B. Froehlich, "Interactive Sankey diagrams," *IEEE Symposium on Information Visualization*, pp. 233–240, 2005.

[140] M. Gleicher, "Considerations for visualizing comparison," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 413–423, 2017.

[141] O. Pearce, T. Gamblin, B. R. De Supinski, M. Schulz, and N. M. Amato, "Quantifying the effectiveness of load balance algorithms," in *Proceedings of the 26th ACM International Conference on Supercomputing*, 2012, pp. 185–194.

[142] N. J. Wright, S. Smallen, C. M. Olschanowsky, J. Hayes, and A. Snavely, "Measuring and understanding variation in benchmark performance," in *DoD High Performance Computing Modernization Program Users Group Conference*. IEEE, 2009, pp. 438–443.

[143] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Nov. 2019.

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

[145] H. D. F. Group., "HDF5 Reference Manual September, National Center for Supercomputing Application (NCSA), University of Illinois at Urbana-Champaign," https://www.hdfgroup.org/solutions/hdf5.

[146] B. Gregg, "The flame graph," *Communications of the ACM*, vol. 59, no. 6, pp. 48–57, 2016.

[147] N. Henry, J.-D. Fekete, and M. J. McGuffin, "Nodetrix: a hybrid visualization of social networks," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1302–1309, 2007.

[148] J. B. Kruskal, "Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis," *Psychometrika*, vol. 29, no. 1, 1964.

[149] H. Bhatia, D. Hoang, N. Morrical, V. Pascucci, P.-T. Bremer, and P. Lindstrom, "AMM: Adaptive multilinear meshes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 6, pp. 2350–2363, 2022.

[150] D. Hoang, P. Klacansky, H. Bhatia, P.-T. Bremer, P. Lindstrom, and V. Pascucci., "A study of the trade-off between reducing precision and reducing resolution for data analysis and visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 1, pp. 1193–1203, 2019.

[151] I. Karlin, "Lulesh programming model and performance ports overview," Lawrence Livermore National Laboratory (LLNL), Livermore, CA (United States), Tech. Rep., 2012.

[152] J. K. Li, T. Fujiwara, S. P. Kesavan, C. Ross, M. Mubarak, C. D. Carothers, R. B. Ross, and K.-L. Ma, "A visual analytics framework for analyzing parallel and distributed computing applications," in *IEEE Visualization in Data Science (VDS)*. IEEE, 2019, pp. 1–9.

[153] T.-c. Fu, "A review on time series data mining," *Engineering Applications of Artificial Intelligence*, vol. 24, no. 1, pp. 164–181, 2011.

[154] J. A. Hartigan and M. A. Wong, "A k-means clustering algorithm," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 28, no. 1, pp. 100–108, 1979.

[155] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, 2009, vol. 344.

[156] R. Xu and D. C. Wunsch, "Survey of clustering algorithms," *IEEE Transactions on Neural Networks*, vol. 16, no. 3, p. 645, 2005.

[157] D. J. Berndt and J. Clifford, "Using dynamic time warping to find patterns in time series," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1994, pp. 359–370.

[158] P.-F. Marteau, "Time warp edit distance with stiffness adjustment for time series matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 31, no. 2, pp. 306–318, 2009.

[159] J. Serra and J. L. Arcos, "An empirical evaluation of similarity measures for time series classification," *Knowledge-Based Systems*, vol. 67, pp. 305–314, 2014.

[160] W. S. Torgerson, "Multidimensional scaling: I. theory and method," *Psychometrika*, vol. 17, no. 4, pp. 401–419, 1952.

[161] L. Van Der Maaten, "Accelerating t-SNE using tree-based algorithms," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 3221–3245, 2014.

[162] C. Bryan, K.-L. Ma, and J. Woodring, "Temporal summary images: An approach to narrative visualization via interactive annotation generation and placement," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 511–520, 2017.

[163] S. Aminikhanghahi and D. J. Cook, "A survey of methods for time series change point detection," *Knowledge and Information Systems*, vol. 51, no. 2, pp. 339–367, 2017.

[164] N. A. James and D. S. Matteson, "ecp: An R package for nonparametric multiple change point analysis of multivariate data," *Journal of Statistical Software*, vol. 62, no. 07, 2015.

[165] M. Mubarak, C. D. Carothers, R. B. Ross, and P. Carns, "Enabling parallel simulation of large-scale HPC network systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 87–100, 2017.

[166] Argonne Leadership Computing Facility, "Theta," https://www.alcf.anl.gov/theta, accessed: 2017-12-11.

[167] C. design at Lawrence Livermore National Laboratory, "Algebraic multigrid solver (AMG)," https://computation.llnl.gov/projects/co-design/amg2013, accessed: 2019-3-8.

[168] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. De Carvalho, and J. Gama, "Data stream clustering: A survey," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 13, 2013.

[169] R. Bro, E. Acar, and T. G. Kolda, "Resolving the sign ambiguity in the singular value decomposition," *Journal of Chemometrics*, vol. 22, no. 2, pp. 135–140, 2008.

[170] K.-L. Ma, C. Wang, H. Yu, and A. Tikhonova, "In-situ processing and visualization for ultrascale simulations," in *Journal of Physics: Conference Series*, vol. 78, no. 1. IOP Publishing, 2007, p. 012043.

[171] A. Bifet, R. Gavaldà, G. Holmes, and B. Pfahringer, *Machine learning for data streams: with practical examples in MOA*. MIT Press, 2018.

[172] E. W. Bethel, H. Childs, and C. Hansen, *High performance visualization: Enabling extreme-scale scientific insight*. CRC Press, 2012.

[173] K. D. Moreland, D. Pugmire, D. Rogers, H. Childs, K.-L. Ma, and B. Geveci, "Xvis: Visualization for the extreme-scale scientific-computation ecosystem: Year-end report." Office of Scientific and Technical Information U.S. Department of Energy, Tech. Rep. DOE-UOREGON-0012380, 2019. [Online]. Available: https://www.osti.gov/biblio/1547341

[174] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The ALPINE In Situ Infrastructure: Ascending from the Ashes of Strawman," in *Proceedings of the ISAV 2017*, pp. 42–46.

[175] J. K. Li, M. Mubarak, R. B. Ross, C. D. Carothers, and K.-L. Ma, "Visual analytics techniques for exploring the design space of large-scale high-radix networks," in *Proceedings of the IEEE Cluster*, 2017, pp. 193–203.

[176] S. Kesavan, H. Bhatia, A. Bhatele, S. Brink, O. Pearce, T. Gamblin, P.-T. Bremer, and K.-L. Ma, "Scalable comparative visualization of ensembles of call graphs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 29, no. 03, pp. 1691–1704, 2021.

[177] S. Albers, "Online algorithms: a survey," *Mathematical Programming*, vol. 97, no. 1-2, pp. 3–26, 2003.

[178] J. Biddiscombe, B. Geveci, K. Martin, K. Moreland, and D. Thompson, "Time dependent processing in a parallel pipeline architecture," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1376–1383, 2007.

[179] A. A. Qahtan, B. Alharbi, S. Wang, and X. Zhang, "A PCA-based change detection framework for multidimensional data streams: Change detection in multidimensional data streams," in *Proceedings of the ACM SIGKDD International Conference in Knowledge Discovery and Data Mining*, 2015, pp. 935–944.

[180] I. T. Jolliffe, *Principal Component Analysis and Factor Analysis*. Springer Series in Statistics, 1986, pp. 115–128.

[181] D. H. Jeong, C. Ziemkiewicz, W. Ribarsky, R. Chang, and C. V. Center, "Understanding principal component analysis using a visual analytics tool," *Charlotte Visualization Center, UNC Charlotte*, vol. 19, 2009.

[182] D. A. Bodenham and N. M. Adams, "Continuous monitoring for changepoints in data streams using adaptive estimation," *Statistics and Computing*, vol. 27, no. 5, pp. 1257–1270, 2017.

[183] M. Carnein and H. Trautmann, "Optimizing data stream representation: An extensive survey on stream clustering algorithms," *Business & Information Systems Engineering*, pp. 1–21, 2019.

[184] D. Sculley, "Web-scale k-means clustering," in *Proceedings of the Conference in World Wide Web*. ACM, 2010, pp. 1177–1178.

[185] J. C. Gower, G. B. Dijksterhuis *et al.*, *Procrustes problems*. Oxford University Press on Demand, 2004, vol. 30.

[186] J. D. Hamilton, *Time series analysis*. Princeton University Press, 1994, vol. 2.

[187] D. A. Pierce and L. D. Haugh, "Causality in temporal systems: Characterization and a survey," *Journal of Econometrics*, vol. 5, no. 3, pp. 265–293, 1977.

[188] M. Mubarak, C. D. Carothers, R. Ross, and P. Carns, "Modeling a million-node dragonfly network using massively parallel discrete-event simulation," in *Proceedings of the 2012 SC Companion*. IEEE, pp. 366–376.

[189] D. W. Bauer Jr, C. D. Carothers, and A. Holder, "Scalable time warp on blue gene supercomputers," in *Proceedings of the ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, 2009, pp. 35–44.

[190] L. Eeckhout, "Is moore's law slowing down? what's next?" *IEEE Micro*, vol. 37, no. 4, pp. 4–5, 2017.

[191] J. P. Dahm, D. F. Richards, A. Black, A. D. Bertsch, L. Grinberg, I. Karlin, S. Kokkila-Schumacher, E. A. León, J. R. Neely, R. Pankajakshan *et al.*, "Sierra center of excellence: Lessons learned," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 2–1, 2019.

[192] S. Mittal and J. S. Vetter, "A survey of CPU-GPU heterogeneous computing techniques," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–35, 2015.

[193] R. W. Wisniewski, X. Tian, P. Thierry, S. Sury, and J. Pennycook, "A holistic systems approach to leveraging heterogeneity," in *2021 IEEE/ACM Programming Environments for Heterogeneous Computing (PEHC)*. IEEE, 2021, pp. 27–33.

[194] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2766–2780, 2015.

[195] N. Dryden, N. Maruyama, T. Benson, T. Moon, M. Snir, and B. Van Essen, "Improving strong-scaling of cnn training by exploiting finer-grained parallelism," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 210–220.

[196] M. J. Abraham, T. Murtola, R. Schulz, S. Páll, J. C. Smith, B. Hess, and E. Lindahl, "Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers," *SoftwareX*, vol. 1, pp. 19–25, 2015.

[197] P. Sinha, A. Guliani, R. Jain, B. Tran, M. D. Sinclair, and S. Venkataraman, "Not all GPUs are created equal: characterizing variability in large-scale, accelerator-rich systems," in *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2022, pp. 01–15.

[198] Y. Ma, F. Rusu, K. Wu, and A. Sim, "Adaptive stochastic gradient descent for deep learning on heterogeneous CPU+GPU architectures," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2021, pp. 6–15.

[199] J. R. Madsen, M. G. Awan, H. Brunie, J. Deslippe, R. Gayatri, L. Oliker, Y. Wang, C. Yang, and S. Williams, "Timemory: modular performance analysis for HPC," in *High Performance Computing: 35th International Conference, ISC High Performance*. Springer, 2020, pp. 434–452.

[200] Y. Sun, Y. Zhang, A. Mosallaei, M. D. Shah, C. Dunne, and D. Kaeli, "Daisen: a framework for visualizing detailed gpu execution," in *Computer Graphics Forum*, vol. 40, no. 3. Wiley Online Library, 2021, pp. 239–250.

[201] J. Lai and A. Seznec, "Performance upper bound analysis and optimization of SGEMM on fermi and kepler GPUs," in *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2013, pp. 1–10.

[202] NVIDIA, "CUDA," https://developer.nvidia.com/cuda-toolkit, 2022.

[203] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in HPC applications," in *18th Euromicro Conference on Parallel, Distributed and Network-based Processing*. IEEE, 2010, pp. 180–186.

[204] F. G. Van Zee and R. A. Van De Geijn, "Blis: A framework for rapidly instantiating blas functionality," *ACM Transactions on Mathematical Software (TOMS)*, vol. 41, no. 3, pp. 1–33, 2015.

[205] D. Yan, W. Wang, and X. Chu, "Demystifying tensor cores to optimize half-precision matrix multiply," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 634–643.

[206] T. Faingnaert, T. Besard, and B. De Sutter, "Flexible performant gemm kernels on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[207] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever *et al.*, "Language models are unsupervised multitask learners," *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.

[208] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2009, pp. 44–54.