2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)

Usability and Performance Improvements in Hatchet

Stephanie Brink*, Ian Lumsden‡, Connor Scully-Allison§, Katy Williams§, Olga Pearce*, Todd Gamblin*, Michela Taufer‡, Katherine E. Isaacs§, Abhinav Bhatele†

*Lawrence Livermore National Laboratory, Livermore, CA, USA
‡Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA
§Department of Computer Science, University of Arizona, Tucson, AZ, USA
†Department of Computer Science, University of Maryland, College Park, MD, USA

***Abstract*—Performance analysis is critical for pinpointing bottlenecks in parallel applications. Several profilers exist to instrument parallel programs on HPC systems and gather performance data. Hatchet is an open-source Python library that can read profiling output of several tools, and enables the user to perform a variety of programmatic analyses on hierarchical performance profiles. In this paper, we augment Hatchet to support new features: a query language for representing call path patterns that can be used to filter a calling context tree, visualization support for displaying and interacting with performance profiles, and new operations for performing analyses on multiple datasets. Additionally, we present performance optimizations in Hatchet's HPCToolkit reader and the unify operation to enable scalable analysis of large datasets.**

***Index Terms*—performance analysis tools, parallel profiles, calling context tree, call graph, graph analytics**

I. Introduction

Profilers [1]–[5] measure code performance on HPC systems, allowing users to identify performance and scalability bottlenecks. Most profilers use their own unique file formats for storing profiling data. These profilers typically provide graphical user interfaces (GUIs) to visualize the data. However, there is limited functionality available to the user to analyze the data programmatically. This ultimately limits the kinds of analyses users can perform on their data.

One challenge of parallel performance analysis is attributing execution time to the code. Simple profilers collect the execution time of individual functions or statements in the code. More advanced profilers can distinguish the time spent in a function when called from different calling contexts, such as an `MPI_Bcast` called by a physics routine versus an `MPI_Bcast` called by a solver library. Other profilers may attribute time to nodes in a call graph, which aggregates the time spent in a function across all occurrences. In all these cases, profile data may represent code in different ways, and analyzing performance data can be a tedious process.

Hatchet [6] is an open-source Python library that overcomes these analysis challenges by enabling users to read the hierarchical profile data generated by different HPC profilers into a canonical data model. Hatchet uses the pandas library [7], [8] and combines graph data with pandas' DataFrames. Using Hatchet, users can perform a variety of operations on profiling data either via the Hatchet API or their own analysis in Python. In this paper, we present several recent changes in Hatchet that improve its usability and performance.

We have developed a query language for representing call path patterns that can be used to filter a calling context tree (CCT). We have also added visualization support for displaying and interacting with performance profiles, in particular, in Jupyter notebooks. We also present new operations added to Hatchet for performing analyses on multiple datasets. And finally, we present performance optimizations in Hatchet's HPCToolkit reader and the unify operation to enable scalable analysis of large datasets.

The main contributions of this paper are as follows:

- a query language to specify call path patterns and a demonstration of its use in analyzing performance variations across MPI implementations;
- enhancements to Hatchet's existing tree-to-text renderer, and a new interactive tree visualization for Jupyter notebooks;
- additions to the Hatchet API to facilitate comparing multiple datasets; and
- optimization of some Hatchet operations and a study of the performance impact of these optimizations.

II. Background

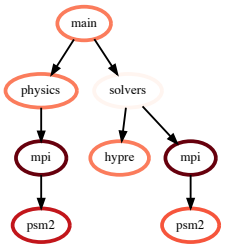
In this section, we provide a brief overview of some common profiling tools, the data they collect, and Hatchet's data model.

A. Profiling Tools and Performance Data

Two common methods for collecting execution profiles of a program are: sampling and source code instrumentation. A sampling-based tool such as HPCToolkit collects data at a regular sampling frequency as the program is executing. With source code instrumentation as used in Caliper, the user annotates their code to specify annotation regions and the tool collects data at each user annotation.

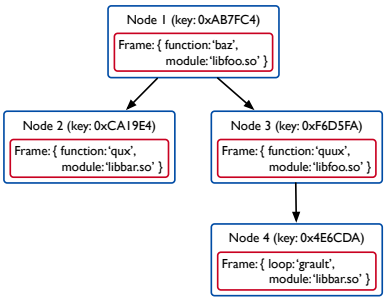
HPCToolkit: HPCToolkit [5] is a suite of tools for performance measurement, analysis, and visualization. HPCToolkit uses thread- and process-level sampling to measure different performance metrics, and attributes their values to the full calling context in which they occur (recorded as a CCT).

978-1-6654-2280-2/20/$31.00 ©2020 IEEE
DOI 10.1109/HUSTProtools51951.2020.00013



node	name	nid	time	time (inc)
{'name': 'main', 'type': 'function'}	main	0	40.0	200.0
{'name': 'physics', 'type': 'function'}	physics	1	40.0	60.0
{'name': 'mpi', 'type': 'function'}	mpi	2	5.0	20.0
{'name': 'psm2', 'type': 'function'}	psm2	3	15.0	15.0
{'name': 'solvers', 'type': 'function'}	solvers	4	100.0	0.0
{'name': 'hypre', 'type': 'function'}	hypre	5	65.0	65.0
{'name': 'mpi', 'type': 'function'}	mpi	6	10.0	35.0
{'name': 'psm2', 'type': 'function'}	psm2	7	25.0	25.0

(a) Hatchet's GraphFrame data structure consists of a Graph object (left) and a pandas DataFrame object (right).



(b) The nodes in Hatchet's Graph object contain a Frame object, which identifies the code construct it represents.

Fig. 1: Hatchet's central data structure and data model.

Caliper: Caliper [2] is a general-purpose instrumentation and profiling library for performance analysis. It provides an API for annotating the application's source code as well as a flexible data aggregation model [9] for online or offline analysis. Caliper generates a hierarchical annotation profile or a CCT, depending on the use of source code annotations or enabling the call path service.

Profiling data typically contains both *contextual information*, such as the filename, line number, and call path for a callsite, and *performance metrics*, such as exclusive and inclusive time, number of instructions retired, or the number of cache misses since the previous sample. The hierarchy might represent a calling context tree (CCT) or call graph depending on how the data is aggregated.

Calling Context Trees: A CCT represents the prefix tree of all call paths in the execution of a parallel program. Each unique function call based on its context becomes a node in the CCT, and a path from the root to any node in the tree represents the calling context that led to that particular function.

Call Graphs: A call graph is created when all nodes representing the same function in a CCT are merged and their performance metrics are aggregated. Call graphs contain all the calling contexts for each node, but the performance metrics are not stored per calling context.

B. Overview of the Hatchet Library

The primary data structure in Hatchet is called a `GraphFrame`, which consists of two components: a `Graph` defining the caller-callee relationships and a pandas `DataFrame` storing the categorical and numerical data associated with each node. Fig. 1a shows the two objects of a GraphFrame. Pandas [8], [10] is an open-source Python library that provides data structures and manipulation tools for data analysis. Hatchet introduces a canonical data model for representing and indexing the performance data from execution profiles. This structured index enables nodes in the structured graph to be used as an index in the pandas DataFrame. Fig. 1b illustrates that each node in the graph contains a `Frame`, which identifies the code construct that the node represents.

Hatchet provides readers for data gathered from several popular profiling tools, such as HPCToolkit, Caliper, gprof, callgrind, and cprofile. Once the data has been read into Hatchet, a user can perform operations to filter or squash the GraphFrame, for example. Some operations are applied to a single GraphFrame (*e.g.,* filter), while others are meant to compare across GraphFrames (*e.g.,* add). In the following sections, we describe the new features that have been added to Hatchet since it was initially introduced in [6].

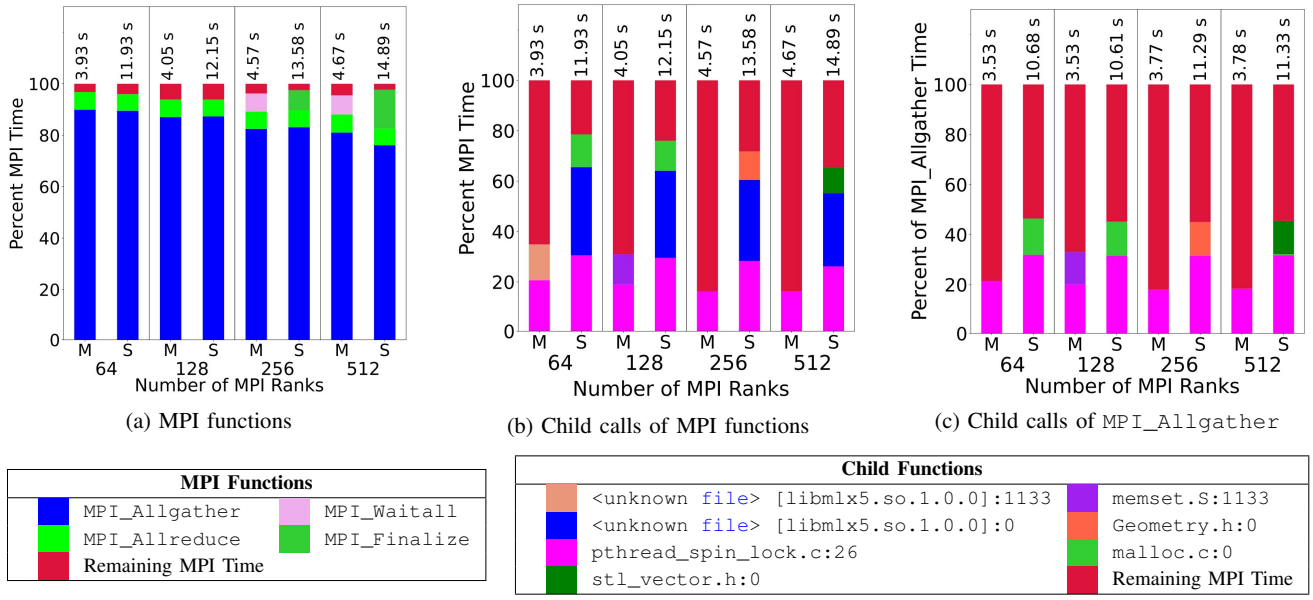
III. Call Path Query Language

We now present Hatchet's call path query language and demonstrate how it can be used to filter the GraphFrame in the following subsections.

A. Design and Implementation

Previously, Hatchet did not provide a way for the user to specify call path patterns to filter the graph. To enable this, we design a call path query language based on Cypher [11] and GQL [12] to filter performance data based on call path patterns. In our query language, users provide a *query path* in the form of a list of *abstract graph nodes*. A node consists of two elements: (1) a wildcard specifying the number of real graph nodes to match to the *abstract graph node*, and (2) a filter determining whether a real graph node matches the *abstract graph node*. We filter the nodes in the real graph in three steps. First, we match all real nodes in the graph to the *abstract nodes* in the user-provided *query path*. Second, we collect an exhaustive list of paths in the graph that match the entire *query path*. Finally, we use the exhaustive list to create a list of real nodes found in the list of matched paths. An example graph and query is shown in Fig. 3.

Our query language consists of two API levels. The "high-level" API represents the *query path* as a Python list in which each element is an *abstract graph node*. Filters in the high-level API are represented as Python dictionaries keyed on the



(a) MPI functions

(b) Child calls of MPI functions

(c) Child calls of `MPI_Allgather`

Fig. 2: Analysis of communication in AMG: (a) The percent of total MPI time spent in MPI functions, (b) The percent of total MPI time spent in the children calls of MPI functions, and (c) The percent of total `MPI_Allgather` time spent in children calls. We focus on those function calls that contribute 10% or more of the total MPI or `MPI_Allgather` time, and combine the time of all remaining function calls into *Remaining MPI Time*. MVAPICH and Spectrum MPI libraries are denoted by by *M* and *S*, respectively. The functions defined in the legends are listed as provided by the HPCToolkit profiles.

attribute names of real nodes in the graph. The "low-level" API represents the *query path* as a set of chained function calls in which each function call represents a single *abstract graph node*. Filters in the low-level API are represented by Python callables that accept a pandas Series representing a row and return a boolean. In both API levels, we represent wildcards as either a number or a regex-style wildcard string.

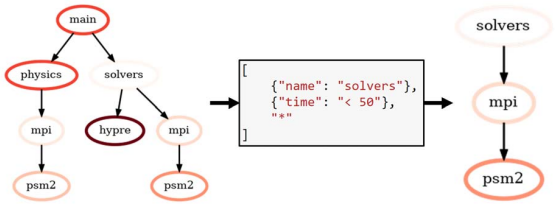


Fig. 3: Example using Hatchet's new call path query language to filter a graph. Here, our query specifies a call path rooted at a node named "solvers", followed by a node with a time metric value less than 50, followed by any number of children nodes. The result is a subtree containing three nodes.

B. Case Study: Identifying Sources of Performance Losses

We demonstrate the effectiveness of Hatchet once augmented with our query language by analyzing the CPU performance of MPI routines when using MVAPICH as compared to IBM's Spectrum MPI in two proxy applications and one production application. We use the call path query language to identify sources of performance losses associated with MPI functions in AMG, Kripke, and LAMMPS. AMG [13] is a parallel algebraic multigrid solver for linear systems derived from the BoomerAMG [14] solver in the hypre library [15]. Kripke is a proxy application for a fully functional discrete-ordinates transport code [16]. LAMMPS is a classical molecular dynamics code with a focus on materials modeling [17]. We use two different MPI libraries (*i.e.,* MVAPICH and Spectrum MPI) with 64, 128, 256, and 512 processes on LLNL's Lassen supercomputer, where each node contains two IBM Power9 CPUs and four NVIDIA Volta V100 (though only the CPUs were used in our study). We profile all the applications using HPCToolkit [5].

Using our query language, we extract the subgraphs rooted at standard MPI function calls from the generated profiles. Using the subgraphs we obtain, we examine the percentage of the total MPI time spent in each MPI function call. We also examine the percentage of total MPI time spent in each child call of the MPI functions. Using this data, we determine the MPI routines and their children calls that are most important to the performance of the application running with a particular MPI library.

In this paper, we only show results related to the `MPI_Allgather` function in AMG due to space constraints. Our reasons for this are two-fold. First, as shown in Fig. 2a, `MPI_Allgather` clearly comprises the majority of the MPI time spent in the AMG benchmark. Second, the AMG benchmark has the largest performance difference between MVAPICH and Spectrum MPI. This suggests that, if we can determine a

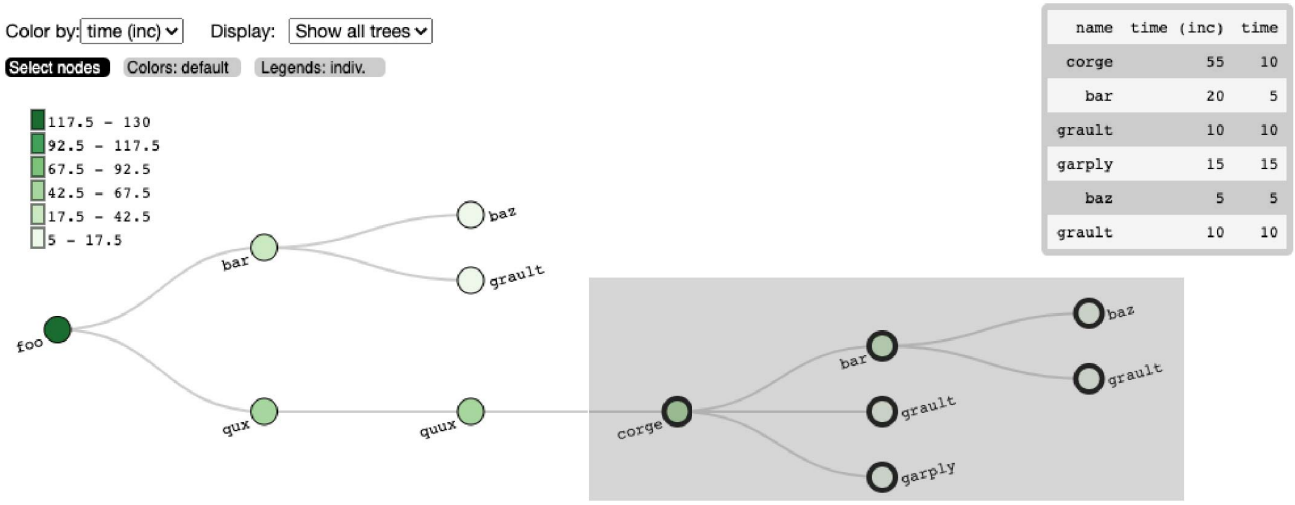


Fig. 4: An example tree rendered using Hatchet's interactive tree visualization in Jupyter. Using the "Select nodes" feature, the user brushes over the `corge` subtree (shown in the grey box). The metrics of the selected nodes are shown in the table in the upper right. This selection can be accessed in other Jupyter cells using the `fetchData` function. Additional controls allow for adjusting the color scale, changing which trees (in a forest of trees) are displayed, and changing what metric is displayed.

likely cause for the performance difference in `MPI_Allgather` between the two MPI libraries, we can also determine the likely primary cause for the overall performance difference between the application versions using these two libraries.

To determine a likely cause for the performance difference in `MPI_Allgather`, we first use the query language to obtain the subgraphs of the AMG data rooted at `MPI_Allgather` calls. We further reduce the data to consider only the children calls of this MPI function that we previously identified in Fig. 2b as most important to the performance of the program. The results of this reduction are shown in Fig. 2c.

In our tests with MVAPICH and Spectrum MPI, we determine that the `pthread_spin_lock` function is consistently a major contributor to MPI runtime (*i.e.,* 10% or more of the MPI time, usually 20% or more). Additionally, when considering `MPI_Allgather`, we conclude that the worse performance of Spectrum MPI may be due to differences in its use of `pthread_spin_lock` compared to MVAPICH.

Overall, Hatchet augmented with the call path query language supports these new analysis capabilities: extracting all call paths specific to a given library; determining the performance contributions of function calls used internally in a library; correlating children function calls to specific important library API calls in an application; using this correlation to determine children function calls that contribute the most to the performance of the targeted library API call; and comparing the correlation of children and API calls across libraries to determine possible causes for performance differences in these libraries.

IV. Visualization Enhancements

We present a new interactive visualization for representing the Hatchet tree, and interacting with it within a Jupyter notebook. We also improve the existing tree-to-text renderer in Hatchet. Both visualizations have refined designs for readability.

A. Interactive Tree Visualization in Jupyter

A central design goal of Hatchet is easing analysis on calling context trees and other similar performance data. While programmatic analysis is the main focus of Hatchet, some operations may be easier to perform in an interactive visual environment. We introduce an interactive tree visualization for Jupyter, shown in Fig. 4. The visualization is built using D3.js [18] and Roundtrip [19]. Our Jupyter visualization has several features that can be directly manipulated, with a key addition of being able to select nodes visually and pass them back to the scripting context.

The interactive tree permits selection of a single node on-click or multiple nodes by brush (gray box drawn around selected nodes). Selection of one or many nodes populates a table in the visualization as shown in the upper right of Fig. 4. The dynamic table lists all selected nodes and their associated metric values. Selected nodes are outlined with a thick black line. Once a user has drawn a selection over multiple nodes, the corresponding query can be accessed in any other Jupyter cell using the Roundtrip `fetchData` function. As shown in Fig. 6, calling `%fetchData(mySelection)` returns the corresponding call path query based on the selection, and stores the query into the `mySelection` Python variable. The resulting query can then be passed to the Hatchet filter function to extract the same subtree programmatically that was selected in the interactive visualization. With the interactive tree, users can visually select nodes that can later be manipulated programmatically, allowing users to combine interactive visual selection with scripting.

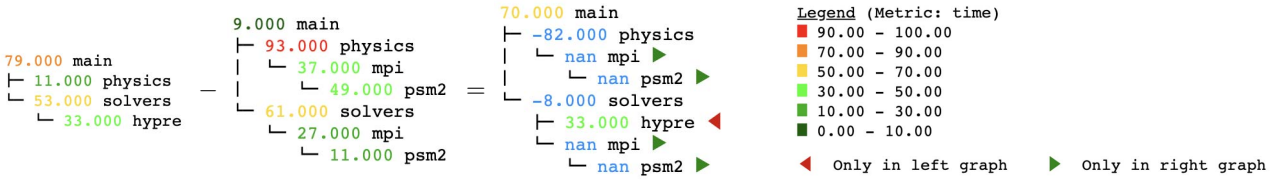
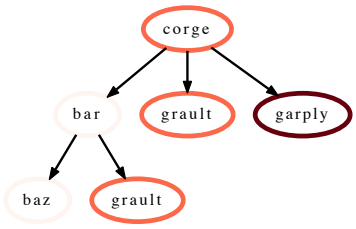


Fig. 5: Tree-to-text rendering for a subtraction of two GraphFrames in Hatchet (resulting tree on the right). Any node that only exists in one of the two trees is annotated with a green or red arrow in the result tree. Here, the `mpi` and `psm2` nodes exist only in the right tree, and are annotated with a green arrow. The `hypre` node exists only in the left tree, and is annotated with a red arrow.




```
%fetchData mySelection
filter_gf = gf.filter(mySelection)
```

Fig. 6: Example demonstrating how the query (translated from the user's selection in the interactive tree in Fig. 4) can be applied programmatically to filter the tree.

Additionally, the Jupyter tree visualization has several interactive controls for adjusting the display. The coloring of the nodes is set by the selected metric in the `Color by` field. Other metrics can be selected using the drop down menu. In Fig. 4, nodes are colored by inclusive time. When multiple trees are present in a Hatchet graph (forest of trees), users can choose to display them all or just one, using a unified colormap, or separate ones. Users can also choose to invert the colormap. For the specific legend used in Fig. 4, dark hues are associated with high metric values and light hues are associated with low metric values.

B. Updates to Tree-to-text Renderer

Hatchet provides a visualization of its graph when the graph is a tree via the tree-to-text renderer, inspired by pyinstrument's text renderer. We have redesigned the output and extended its functionality to provide users with more customization over their visualizations. Examples of the current tree-to-text output can be seen in Fig. 5. By default, node names are printed alongside the specified metric, such as exclusive or inclusive time. Users can specify `depth` or `precision` to Hatchet's tree renderer to control the number of tree levels to output and the number of decimal places to output for the displayed metric values.

We also provide users with increased control over the colormap. By default, the colormap annotates nodes with the highest metric values in red and those with the lowest metric values in green. In the case where a user computes the division (or speedup) of two GraphFrames, a user may want to invert the colormap, so that nodes with high speedup are annotated in green, while nodes with low speedup are annotated in red. Users can invert the default colormap by specifying `invert_colormap=True`. Additionally, the tree renderer now annotates nodes that only exist in one of the two GraphFrames in the right hand side of an algebraic operation, such as subtraction. As shown in Fig. 5, the graphs of the two input GraphFrames are structurally different, so we first unify the graphs before computing the difference. Unifying the graphs means that some nodes exist in one GraphFrame but not the other, and vice versa. In the output GraphFrame, we annotate those nodes with a red left arrow to indicate nodes that exist only in the left input graph, or a green right arrow to indicate nodes that only exist in the right input graph.

V. Improvements in the Hatchet API

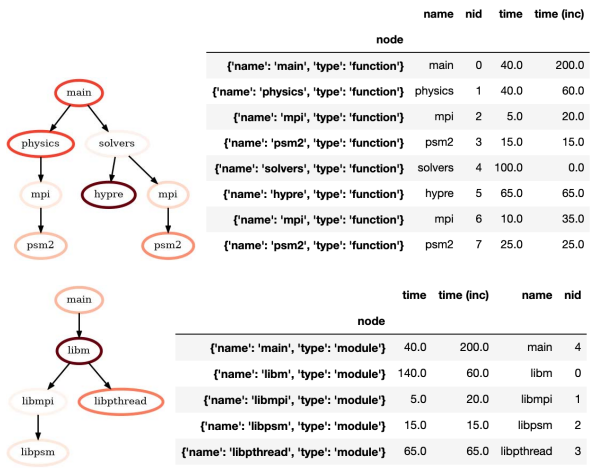
In this section, we describe an extension to the filter function and three new GraphFrame operations that have been added to Hatchet – groupby_aggregate, mul, and div. Filter and groupby_aggregate operate on a single GraphFrame object, while mul and div operate on two GraphFrames objects.

Extension to filter: The existing filter operation takes a user-supplied function and applies that to all rows in the DataFrame. As an example, the filter function may keep all rows in the DataFrame that have a particular column's values greater than some threshold. With the newly added query language, we have extended the filter function to support taking a user-defined query as input to filter the graph based on call path patterns. Hatchet's new call path query language is described in detail in Section III. The Series or DataFrame represented by the query is used to filter the GraphFrame's DataFrame to only match rows that are true. By default, filter performs a `squash` on the graph to remove nodes that are no longer in the DataFrame after applying the filter. Squash rewires the graph such that the nearest remaining ancestor is connected to the nearest remaining child on all call paths.

groupby_aggregate: The groupby and aggregate operation takes as input a column to use for the pandas groupby operation and an aggregation function to apply on the members of each group. It produces a new DataFrame that has number

of rows equal to the number of groups (rows in each group are aggregated). This operation is useful for aggregating the data into alternative groups based on hierarchies other than the column that denotes the calling context. As an example, users may want to look at the performance attributed to different modules or file names (instead of the default, which is typically function or region names), and aggregate the data values accordingly.

As part of the groupby and aggregate operation, the graph is reorganized to be able to index the new rows in the DataFrame. For each group, the graph reorganization merges all nodes belonging to a group into a single "supernode" (and aggregates corresponding rows in the DataFrame to a single row). When a node is merged into a supernode, any of its edges to a parent or child are added as edges to the new supernode. Each supernode is represented by a new node in the GraphFrame, and these new nodes are used to index the DataFrame. The groupby_aggregate returns a new GraphFrame with a reorganized graph and a groupby-aggregated DataFrame. Fig. 7 shows the graph before and after a groupby-aggregate is performed, specifying *module* as the new column to use for the groupby operation.



node	name	nid	time	time (inc)
{'name': 'main', 'type': 'function'}	main	0	40.0	200.0
{'name': 'physics', 'type': 'function'}	physics	1	40.0	60.0
{'name': 'mpi', 'type': 'function'}	mpi	2	5.0	20.0
{'name': 'psm2', 'type': 'function'}	psm2	3	15.0	15.0
{'name': 'solvers', 'type': 'function'}	solvers	4	100.0	0.0
{'name': 'hypre', 'type': 'function'}	hypre	5	65.0	65.0
{'name': 'mpi', 'type': 'function'}	mpi	6	10.0	35.0
{'name': 'psm2', 'type': 'function'}	psm2	7	25.0	25.0

node	time	time (inc)	name	nid
{'name': 'main', 'type': 'module'}	40.0	200.0	main	4
{'name': 'libm', 'type': 'module'}	140.0	60.0	libm	0
{'name': 'libmpi', 'type': 'module'}	5.0	20.0	libmpi	1
{'name': 'libpsm', 'type': 'module'}	15.0	15.0	libpsm	2
{'name': 'libpthread', 'type': 'module'}	65.0	65.0	libpthread	3

```
gf = GraphFrame( ... )
groupby_col = ["module"]
agg_func = {"time": np.sum, "time (inc)": np.sum}
gf2 = gf.groupby_aggregate(groupby_col, agg_func)
```

Fig. 7: Groupby-aggregate operation applied to a single GraphFrame. The bottom figures show the resulting module-level graph and associated DataFrame.

mul: The multiplication (*) operation computes the multiplication of the corresponding DataFrames in two GraphFrames. If the graphs of the operands are not the same, then unify is applied first to create a single unified graph. Then the DataFrames are reindexed by the unified graph. The multiplication operation returns a new GraphFrame with the unified graph and the result of multiplying the DataFrames. The multiplication operator can also be used in-place ($a* = b$) to update an existing GraphFrame.

div: The division ($/$) operation is similar to the other algebraic operations in Hatchet, and computes the division of two DataFrames. Similar to mul, if the graphs of the operands are not the same, it first unifies the graphs and reindexes the DataFrames before performing the division. The division operation either returns a new GraphFrame or updates the GraphFrame in-place if the in-place division operator ($a/ = b$) is used.

VI. Performance Improvements

Performance improvements in Hatchet aim to support large profiles collected from executions of parallel programs on a large number of processes. These efforts target two critical functions in Hatchet: the HPCToolkit reader and `unify`. We detail the optimization process and the results of these efforts in the following subsections.

A. Performance Analysis Infrastructure

To enable performance analysis of Hatchet, we developed a custom-made cProfile [20] wrapper class. This class provides simple annotations for starting, stopping, and resetting the profiler in Python code that uses Hatchet. Furthermore, we created several interfaces for aggregating and exporting the measured performance data for post-mortem analysis. This minimal profiling infrastructure provides Hatchet developers and users with a framework for quickly identifying bottlenecks within Hatchet workflows.

B. Optimizing the HPCToolkit Reader

The detailed calling contexts and per process metrics gathered by HPCToolkit can result in large and complex datasets. As an example, an HPCToolkit profile of LAMMPS collected on 512 processes (13 nodes) on LLNL's Lassen system produces a calling context tree (CCT) of over 34,000 call sites and approximately 50,000,000 performance data records. The size of HPCToolkit's profiles made Hatchet's HPCToolkit reader an obvious first step toward extending Hatchet's support for big data.

We identified the critical bottleneck of the HPCToolkit reader inside of a recursive, tree-traversing function call that constructs Hatchet's graph nodes from HPCToolkit's XML representation of call sites in a profile. In addition to constructing nodes for the Hatchet graph, when this function arrives at leaf nodes, it subtracts the exclusive time of leaf nodes from their parents (HPCToolkit stores the exclusive time of statement nodes in both the leaf nodes and their parents). The few lines of Python code dedicated to this computation dominate the bottleneck found in this function.

The procedure itself uses pandas' conditional indexing functionality to find all rows containing the current node's ID and its parent's ID. The two resulting lists are then subtracted as vectors and re-inserted into Hatchet's DataFrame. In a sequential profile, this would be only two rows in the DataFrame, accessible directly by the structured index. However, HPCToolkit metrics are collected per execution thread

and process for each call site. This means that a given node ID appears in the DataFrame n times, where n is the product of the total number of execution threads $\times$ processes the profiled application was running on. For highly parallel programs, this can mean searching for tens of thousands of rows containing the same parent and child ID.

This significant number of rows, with slight variations in the metric data across threads and processes, causes Hatchet's DataFrame to explode in size compared to its corresponding CCT. Since the pandas DataFrame is not optimized to handle operations on very large datasets, operations such as conditional indexing are the primary bottlenecks in Hatchet. For each statement node in HPCToolkit's XML data, the conditional indexing executes twice: once for the parent and again for the child to get the two vectors of exclusive metrics, increasing the time spent in this slow operation.

We optimized the conditional index operation by leveraging the structure of this data as well as opportunities to speedup array-based operations provided by the C/Python hybrid language, Cython [21]. We first extract the relevant columns (*i.e.,* exclusive metrics) from the DataFrame, pass them into a Cython function, and exploit the structure of the data to stride over millions of rows of data in a few iterations, locating and updating only those rows of interest.

Hatchet's DataFrame can be decomposed into t equal sized sub-frames of length m, where t is the number of execution threads $\times$ processes used to run the application and m is the number of call sites. Since each sub-frame is sorted by node ID, there is no need to iterate over the entire DataFrame row-by-row. Instead, we make t strides of length m over the metric values and subtract the children metrics from the parent metrics at each iteration. For our largest dataset, we reduced the number of iterations over our DataFrame (per function call) from more than 100,000,000 to just above 30,000.

C. Evaluation of Performance Improvements

To examine the impact of our optimizations to the HPCToolkit reader, we measured the runtime of Hatchet's HPCToolkit reader on a series of profiles varying in size from 999 call graph nodes and 191,808 rows in the DataFrame to 34,855 nodes and 53,537,280 rows. The HPCToolkit profiles used in our performance study came from the case study described in Section III-B. The smallest profile was for a Kripke execution on 64 processes (2 nodes) and the largest was for a LAMMPS run on 512 processes (13 nodes). The results of these trials are presented in Fig. 8. for each read first on the unoptimized code and again on the optimized code. Each point represents the average performance over five trials to read in a HPCToolkit profile of a given size (*i.e.,* number of DataFrame rows).

Our optimizations significantly improve the performance of Hatchet's HPCToolkit reader. As shown in Fig. 8, the slowdown of the pre-optimized implementation is more pronounced with larger DataFrames, while the post-optimized code scales linearly. For even larger datasets, the relative speedup of the optimized HPCToolkit reader will continue to increase. For a DataFrame containing 50,000,000 rows, the HPCToolkit reader went down from six hours and fifteen minutes to two minutes and twenty-four seconds, a reduction of two orders of magnitude.

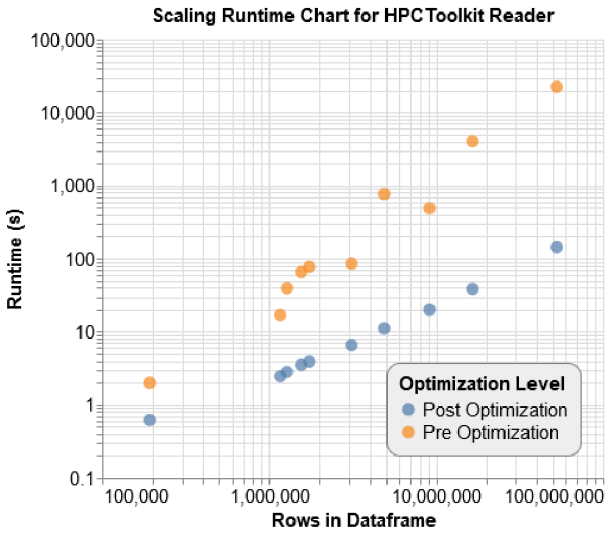


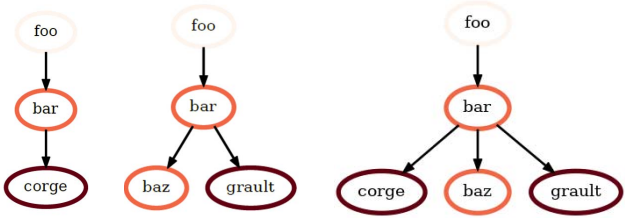
Fig. 8: Log-log plot showing performance before and after optimization of the HPCToolkit reader as the size of the Hatchet DataFrame increases. The optimized HPCToolkit reader scales significantly better compared to its unoptimized predecessor.

D. Optimizing the Unify Operation

The unify operation takes two GraphFrame objects, unifies the graphs in them, and reindexes the DataFrames by the nodes in the unified graph. The updated GraphFrames contain the new unified graph (as shown in Fig. 9) and reindexed DataFrames. The DataFrame of the GraphFrame object calling unify contains all the nodes from both DataFrames and also stores metadata about the origin of nodes with a column `_missing_node`, which denotes that a particular node existed only in its DataFrame or in the DataFrame of the other GraphFrame. If a node existed in both GraphFrames, then this column is left empty.

We chose to optimize unify since it is a primary operation in most of Hatchet's algebraic operations, such as multiply or add. Since these algebraic operations are critical to the unique profiling workflow offered by this library, it is essential that they be performant. The initial performance analysis of unify, executed with the same profiling infrastructure introduced in Section VI-B, reveals merit in targeting unify as a potential bottleneck. Unifying a LAMMPS dataset with 50,000,000 rows and 34,000 nodes with another dataset of roughly equivalent size takes one hour and 38 minutes. Even when unifying smaller datasets (100,000 rows and 1,000 nodes in the CCT), the unify operation is notably slow, consuming 30 seconds.

Unify's runtime is dominated by the time spent updating the DataFrames. However, in contrast to HPCToolkit, slowdowns are spread out among several pandas library operations in Hatchet's internal DataFrame management function,



node	_missing_node	name	time	time (inc)
{'name': 'foo', 'type': 'function'}		foo	0.0	15.0
{'name': 'bar', 'type': 'function'}		bar	5.0	15.0
{'name': 'baz', 'type': 'function'}	R	baz	0.0	0.0
{'name': 'corge', 'type': 'function'}	L	corge	10.0	10.0
{'name': 'grault', 'type': 'function'}	R	grault	0.0	0.0

```
gf1 = GraphFrame( ... )
gf2 = GraphFrame( ... )
gf1.unify(gf2)
```

Fig. 9: An example of Hatchet's unify operation. The left graph and middle graph are unified by traversing both graphs and adding any nodes that exist in one graph but not the other to the result. The result is a single unified graph shown on the right. The DataFrame of the GraphFrame object calling unify contains a new `_missing_node` column identifying which nodes were exclusive to each graph (denoted by an L or R).

`_insert_missing_rows`. Our initial performance enhancements involved tweaking existing code to follow pandas conventions. We replaced assignment of values in DataFrames in C-like loops for Python lists or NumPy arrays that are then assigned to a pandas DataFrame column. Both native Python and NumPy handle single-element array access and their array/list creation significantly better than pandas. This optimization provides some marginal speedup and reveals a need to integrate Cython for more substantial performance gains.

Another bottleneck in unify is the pandas `isin` method, called by the `_insert_missing_rows` function. This particular method takes an argument of a list, NumPy array or pandas Series, and returns a boolean mask with a true or false for each element in the passed array. This mask indicates each element's presence in the calling DataFrame. For DataFrames of over 1,000,000 rows, common for most of our test datasets, pandas falls back to NumPy's `isin` functionality, which combines `np.unique` for sorting, and a binary search for determining membership. The `isin` operation does not perform especially well with lists of complex objects, such as the nodes used by Hatchet.

To improve the slowdown in pandas' `isin`, we implemented a more specialized function in Cython. Hatchet's specific `isin` function is directly optimized for Hatchet's data and designed to use as few columns as possible. By using Cython, we reduced overhead introduced by superfluous calls through pandas to NumPy, and into NumPy's various libraries. Furthermore, we pre-process our complex array of "node" objects into a sorted array of integer node IDs. This fundamental array can be quickly iterated over and with lower memory overhead. We implemented the `isin` function itself as a binary search inside of a loop over the searched-for array of elements. We further sped up this binary search by adding an early stopping condition: if we have previously visited the current node ID, then copy the prior results to this one and forego the search. This optimization ensures that the number of binary searches performed is bound by the number of nodes and not the number of rows.

There is very little opportunity for a critical spot optimization like Hatchet's custom `isin` function. However, removing multi-indexes resulted in another 25% speedup over the order-of-magnitude speedup gained from prior optimizations. Hatchet leverages multi-indexes comprised of nodes, processes, and threads to provide a meaningful and unique index for each row in its DataFrame. Compared to single indexes, multi-indexes introduce a noticeable overhead to standard pandas operations like `concat` or even assignment of a new column. Although the source of this overhead is not abundantly clear, the use of multi-indexes apparently introduces a layer of calls through the "multi" library in pandas. By removing the multi-index, we eliminated this layer of calls to "multi" and reduce the slowdown observed with many pandas methods.

E. Evaluation of Performance Improvements

To measure the impact of our optimizations to the `unify` method, we collected the runtimes produced by unifying two similar GraphFrames produced from the same set of HPCToolkit profiles we used to profile Hatchet's HPCToolkit reader. For our performance study, we unify two HPCToolkit profiles for the same application, each using a different MPI implementation. The performance results, averaged over five trials are shown in Fig. 10. Because this experiment required pairs of datasets per run (*i.e.,* one profile using MVAPICH and the other using Spectrum MPI), there are two fewer data points in this figure as compared to Fig. 8. The two missing profiles were only collected with MVAPICH or Spectrum MPI but not both, so they were omitted from our performance study.

For smaller datasets, we saw a reduction in runtime from 30 seconds to approximately 1 second. For mid-sized DataFrames containing millions of rows, `unify`'s execution time went from minutes down to tens of seconds. For very large datasets, `unify`'s runtime went down from greater than an hour to only a few minutes. The primary contributor to the order-of-magnitude speedup between our pre- and post-optimization is Hatchet's Cythonized, custom `isin` function. The similarity in the trends of our pre- and post-optimization runtime measurements speaks to the similarity of the underlying functionality which drives NumPy's `isin` method and Hatchet's. We attribute the substantial speedup to a reduction in overhead from Python function calls, memory management, bounds checking, and reduced space requirements. However,

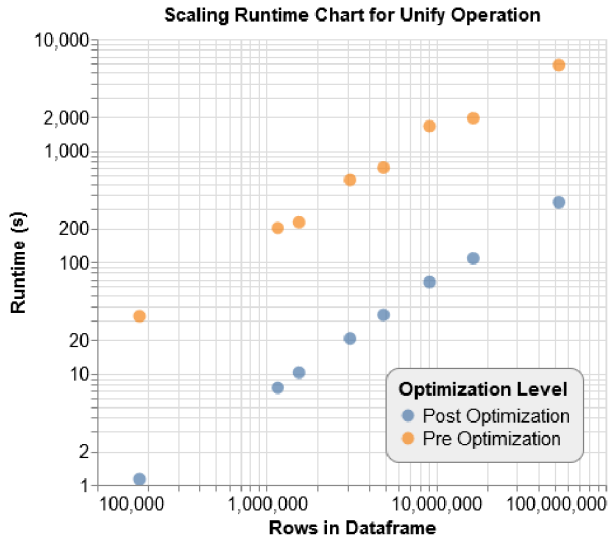


Fig. 10: Log-log plot showing the pre- and post-optimization performance of the `unify` operation as the size of the dataset increases.

no algorithmic advantage exists like that produced by our HPCToolkit reader optimizations.

F. Performance Improvement of Common Workflows

A simple workflow for Hatchet is depicted in Fig. 11. In this workflow, a user reads in two HPCToolkit profiles, perhaps collected at two different levels of concurrency or varying the underlying MPI implementation. A user then uses one of Hatchet's algebraic operators to make a quick comparison between the two runs, storing the result in a new GraphFrame. Before doing the optimizations discussed in this section, this program would take fourteen hours to compare two HPCToolkit profiles collected from a large program run on 512 ranks. After integrating the optimizations detailed in this section, the overall runtime for analyzing large datasets has been significantly reduced. The runtime for this workflow has been reduced to ten minutes and thirty seconds for the same large profiles (80× improvement).

```
gf1 = GraphFrame.from_hpctoolkit( ... )
gf2 = GraphFrame.from_hpctoolkit( ... )
gf3 = gf1 - gf2
```

Fig. 11: Simple workflow using the Hatchet library. Two similar HPCToolkit datasets are read in to Hatchet, and we compute the difference in their metrics. With optimizations, we reduced the time for executing this workflow from 14 hours to 10 minutes and 30 seconds.

VII. Related Work

There is a wide variety of profilers that can collect call graphs or call paths for post-hoc analysis [2]–[5], [22], [23]. Many of these profilers also provide visualization tools for viewing calling context trees (CCT), including Tau [4], HPCToolkit's hpcviewer [24] and hpctraceviewer [25], CallFlow [26], [27], Cube GUI [28], and flame graphs [29]. All of these profilers support their own data format, and most visualization tools provide a custom GUI interface for viewing the call path. While some tools are capable of importing data from other tools, there is a lack of tools with a programmable interface for automating interactions with the profile data. With Hatchet, we develop a canonical data format for profile data, so that data from several popular profiler tools can be analyzed with Hatchet. Additionally, Hatchet provides interfaces to automate the performance analysis of call path data, so users do not have to learn new data formats or visual interfaces.

Within the tools community, there is an effort to leverage a database for storing data and to provide their own language for interacting with the data. PerfExplorer [30], for example, provides its own database, a GUI interface, and a custom data format known as PerfDMF [31]. Similarly, Open|SpeedShop uses an SQL database and its own GUI interface. The work most closely related to Hatchet is differential profiling, which demonstrated the benefits of computing the difference between two call trees to pinpoint performance bottlenecks [32], [33]. To expand on this idea and to enable analysis of larger profiles, Tallent et al. extended HPCToolkit to include derived metrics [34], [35]. Since Hatchet is built upon the pandas data analysis library [8], [10], it provides a number of data manipulation APIs that are performant on large tabular data.

VIII. Conclusion and Future Work

Analyzing performance and pinpointing bottlenecks in parallel programs are important to guide developers in their optimization workflow. It is a significant challenge to effectively analyze the performance of complex programs that contain tens of thousands of lines of code, resulting in large dynamic calling context trees or call graphs. In this paper, we provided an overview of four different efforts to enhance Hatchet's usability and performance.

We introduced Hatchet's new query language to enable users to specify call path patterns for filtering the graph. We demonstrated Hatchet's new interactive visualization capabilities in Jupyter, enabling users to drag and select a subtree to filter the graph. We have also improved the functionality and information displayed using Hatchet's tree-to-text renderer. We provided an overview of new APIs that have been added to Hatchet's analysis toolbox. Lastly, we discussed different optimizations to Hatchet's existing APIs and showed significant speedups at large scale.

In the future, we plan to add functionality to save Hatchet's GraphFrame to disk, enabling users to save intermediate GraphFrames periodically throughout the analysis process. For large datasets that may take a significant amount of time to read in, this capability will significantly improve the analysis workflow with Hatchet, since analysis can start from

an intermediate step. We also plan to explore options that will enable use of parallel frameworks for analyzing Hatchet data.

ACKNOWLEDGMENT

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-814318). This work was supported in part by funding provided by the University of Maryland College Park Foundation.

REFERENCES

[1] J. Fenlason and R. Stallman, *GNU* `gprof`*: the GNU Profiler, http://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html*, Free Software Foundation, November 7 1988.

[2] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance Introspection for HPC Software Stacks," in *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. IEEE Computer Society, Nov. 2016, pp. 47:1–47:11, LLNL-CONF-699263.

[3] A. Knüpfer, C. Rössel, D. a. Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*, H. Brunst, M. S. Müller, W. E. Nagel, and M. M. Resch, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 79–91.

[4] S. Shende and A. Malony, "The TAU Parallel Performance System," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–331, 2006.

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

[6] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the Overgrowth in Parallel Profiles," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, 2019.

[7] The pandas development team, "pandas-dev/pandas: Pandas," Feb. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[8] Wes McKinney, " Data Structures for Statistical Computing in Python ," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61.

[9] D. Böehme, D. Beckingsale, and M. Schulz, "Flexible Data Aggregation for Performance Profiling," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2017, pp. 419–428.

[10] W. McKinney, *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media, 2017.

[11] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An Evolving Query Language for Property Graphs," in *2018 International Conference on Management of Data (SIGMOD18)*, 2018.

[12] A. Green, P. Furniss, P. Lindaaker, P. Selmer, H. Voigt, and S. Plantikow, "GQL Scope and Features," ISO, Tech. Rep., 2019.

[13] "AMG2013 – parallel algebraic multigrid solver," 2015. [Online]. Available: https://computing.llnl.gov/projects/co-design/amg2013

[14] V. E. Henson and U. M. Yang, "BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner," *Appl. Numer. Math.*, vol. 41, no. 1, p. 155–177, Apr. 2002.

[15] R. D. Falgout and U. M. Yang, "Hypre: A Library of High Performance Preconditioners," in *Proceedings of the International Conference on Computational Science-Part III*, ser. ICCS '02. Berlin, Heidelberg: Springer-Verlag, 2002, p. 632–641.

[16] A. Kunen, T. Bailey, and P. Brown, "KRIPKE-A massively parallel transport mini-app," Lawrence Livermore National Laboratory (LLNL), Tech. Rep., 2015.

[17] S. Plimpton, "Fast Parallel Algorithms for Short-Range Molecular Dynamics," *J. Comput. Phys.*, vol. 117, no. 1, p. 1–19, Mar. 1995.

[18] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-Driven Documents," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011.

[19] J. Bartels, "Roundtrip," https://github.com/hdc-arizona/roundtrip, Last Accessed September 2020.

[20] Python, "The Python Profilers," https://docs.python.org/3/library/profile.html, Last Accessed September 2020.

[21] S. Behnel, R. W. Bradshaw, and D. S. Seljebotn, "Cython tutorial," in *Proceedings of the 8th Python in Science Conference*, G. Varoquaux, S. van der Walt, and J. Millman, Eds., Pasadena, CA USA, 2009, pp. 4 – 14.

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

[23] The Open|SpeedShop Team, "Open|SpeedShop for Linux." [Online]. Available: http://www.openspeedshop.org

[24] J. Mellor-Crummey, R. Fowler, and G. Marin, "HPCView: A tool for top-down analysis of node performance," *The Journal of Supercomputing*, vol. 23, pp. 81–101, 2002.

[25] N. R. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto, "Scalable Fine-Grained Call Path Tracing," ser. ICS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 63–74.

[26] H. T. Nguyen, L. Wei, A. Bhatele, T. Gamblin, D. Boehme, M. Schulz, K. Ma, and P. Bremer, "VIPACT: A Visualization Interface for Analyzing Calling Context Trees," in *2016 Third Workshop on Visual Performance Analysis (VPA)*, Nov 2016, pp. 25–28.

[27] 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.

[28] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr, "Cube V4," *Procedia Comput. Sci.*, vol. 51, no. C, p. 1343–1352, Sep. 2015.

[29] B. Gregg, "Flame graphs," Online, 2015, http://www.brendangregg.com/Slides/FreeBSD2014_FlameGraphs.pdf.

[30] K. A. Huck and A. D. Malony, "PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing," in *Supercomputing 2005 (SC'05)*, Seattle, WA, November 12-18 2005, p. 41.

[31] K. Huck, A. D. Malony, R. Bell, L. Li, and A. Morris, "PerfDMF: Design and implementation of a parallel performance data management framework," in *International Conference on Parallel Processing (ICPP'05)*, 2005.

[32] P. E. McKenney, "Differential profiling," in *MASCOTS '95. Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Jan 1995, pp. 237–241.

[33] M. Schulz and B. R. de Supinski, "Practical Differential Profiling," in *Euro-Par 2007 Parallel Processing*, A.-M. Kermarrec, L. Bougé, and T. Priol, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 97–106.

[34] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing Performance Bottlenecks in Emerging Petascale Applications," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: Association for Computing Machinery, 2009.

[35] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '10. USA: IEEE Computer Society, 2010, p. 1–11.