LLNL-CONF-814318

Usability and Performance Improvements in Hatchet

S. Brink, I. Lumsden, C. Scully-Allison, K. Williams, O. Pearce, T. Gamblin, M. Taufer, K. Isaacs, A. Bhatele

September 4, 2020

International Conference for High Performance Computing, Networking, Storage, and Analysis
Atlanta, GA, United States
November 9, 2020 through November 19, 2020

Usability and Performance Improvements in Hatchet

Stephanie Brink[1], Ian Lumsden[2], Connor Scully-Allison[3], Katy Williams[3],
Olga Pearce[1], Todd Gamblin[1], Michela Taufer[2], Katherine E. Isaacs[3], Abhinav Bhatele[4]

[1]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA, USA
[2]Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, TN, USA
[3]Department of Computer Science, University of Arizona, Tucson, AZ, USA
[4]Department of Computer Science, University of Maryland, College Park, MD, USA

***Abstract*—Performance analysis is critical for pinpointing bottlenecks in applications. Many different profilers exist to instrument parallel programs on HPC systems, however, there is a lack of tools for analyzing such data programmatically. Hatchet, an open-source Python library, can read profiling data from several tools, and enables the user to perform a variety of analyses on hierarchical performance data. In this paper, we augment Hatchet to support new features: a syntax query language for representing call path-related queries, visualizations for displaying and interacting with the structured data, and new operations for performing analysis on multiple datasets. Additionally, we present performance optimizations in Hatchet's HPCToolkit reader and the unify operation to enable scalable analysis of large profiles.**

***Index Terms*—performance analysis, tool, parallel profile, calling context tree, call graph, graph analytics**

I. Introduction

Profilers measure code performance on HPC systems [1]–[5], allowing users to identify performance and scalability bottlenecks. Unfortunately, most profilers use their own unique format for storing profiling data. As a result, users are bound to the analysis tools provided by the profiling software. These tools are typically GUI-based, and they do not allow the user to analyze performance 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 time spent in different call paths or 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 performance across all occurrences. In all cases, profile data can represent code in a variety of ways, and analyzing performance data can be a tedious process.

Hatchet [6] is an open-source Python library that overcomes these analysis constraints by enabling users to read the hierarchical call graph data generated by different HPC profilers into a canonical data model. Hatchet builds upon a combination of the pandas Python library [7], [8] and graph-based hierarchical data representations. After reading the hierarchical call graph data into Hatchet, users can perform a variety of Hatchet's operations or they can perform their own further analysis in Python. We continue to extend and improve upon Hatchet's techniques and operations, such as the graph-based hierarchical data representations, to enable users to perform deeper analyses of their parallel performance data.

The contributions of this paper are as follows:

- a description of Hatchet's syntax query language and an example case study analyzing performance variations across MPI implementations;
- enhancements to Hatchet's graph visualizations including an interactive tree visualization;
- an overview of Hatchet's new APIs providing deeper analysis of structured data; and
- a performance study of Hatchet's optimized APIs.

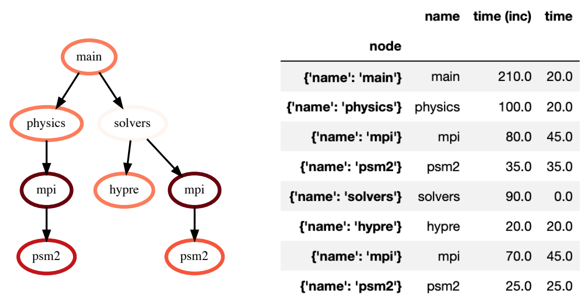
II. Background

In this section, we describe the structured, hierarchical performance data collected by different profiling tools, an overview of two popular profiler tools in the HPC community, as well as Hatchet's data model.

A. Structured Performance Data

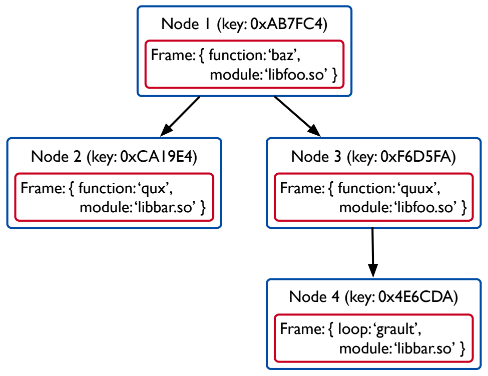
There are two methods for collecting execution profiles for an application: sampling or source code instrumentation. With sampling, a profiling tool (*e.g.,* HPCToolkit) will collect data at regular intervals as the program is executing. With source code instrumentation, annotation code (*e.g.,* Caliper) is inserted directly into the application and data is collected at each instrumentation point. The profile data may contain both *contextual information*, such as the current line number, file name, or call path, and *performance metrics*, such as the number of instructions retired or the number of cache misses since the previous sample. The execution profiles may vary depending on how the data is aggregated (*i.e.,* in graphs or trees).

Call Graphs: Call graphs attribute data to the name of the function rather than performing an analysis of the call stack and determining the call path or calling context. In a call graph, an edge connecting two nodes implies that one function called the other, and the samples are averaged across all occurrences regardless of where they originated. Call graphs are considered a very concise representation of call paths, however, they lack the context in which particular functions are called.



node	name	time (inc)	time
{'name': 'main'}	main	210.0	20.0
{'name': 'physics'}	physics	100.0	20.0
{'name': 'mpi'}	mpi	80.0	45.0
{'name': 'psm2'}	psm2	35.0	35.0
{'name': 'solvers'}	solvers	90.0	0.0
{'name': 'hypre'}	hypre	20.0	20.0
{'name': 'mpi'}	mpi	70.0	45.0
{'name': 'psm2'}	psm2	25.0	25.0

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


Node 1 (key: 0xAB7FC4)
Frame: { function: 'baz', module: 'libfoo.so' }
Node 2 (key: 0xCA19E4)
Frame: { function: 'qux', module: 'libbar.so' }
Node 3 (key: 0xF6D5FA)
Frame: { function: 'quux', module: 'libfoo.so' }
Node 4 (key: 0x4E6CDA)
Frame: { loop: 'grault', module: 'libbar.so' }

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

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

Calling Context Trees (CCTs): A calling context tree is a hierarchical structure storing the calling context of an application. Each unique call of a function becomes a node in the CCT, and a path from the root to any node in the graph represents the calling context that led to that particular child function. Because CCTs provide a more verbose representation of call paths, they are useful for analyzing performance of different hardware performance counters within the context of how they were called.

B. Call Path Profiler Tools

There are many different profiler tools for identifying performance bottlenecks. We provide an overview of two common call path profiling tools below.

HPCToolkit: HPCToolkit [9] is a suite of tools for performance measurement, analysis, and visualization. HPCToolkit uses thread- and process-level sampling to measure different hardware performance counters and attributes the value to the full calling context in which they occur (recorded as a CCT). The call path for a given node is determined by tracing the path from that node to the root. The performance database generated by HPCToolkit's `hpcprof` or `hpcprof-mpi` tool correlates the call path profiles with the application's source code. The resulting database includes a single XML-formatted CCT (for all processes) and individual files containing the process-level metrics for all nodes in the CCT.

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 [10] for online or offline analysis. During execution, Caliper builds a generalized context tree consisting of user-defined key-value pairs known as attributes. The context information for any given node is derived by collecting all attributes on the path from that node to the root node. Caliper generates a hierarchical profile or CCT by enabling the call path service or by using the source code annotations. To generate Caliper's JSON-split output format, a user can run `cali-query` on the raw Caliper samples or enable the `hatchet-region-profile` service.

C. An Overview of the Hatchet Library

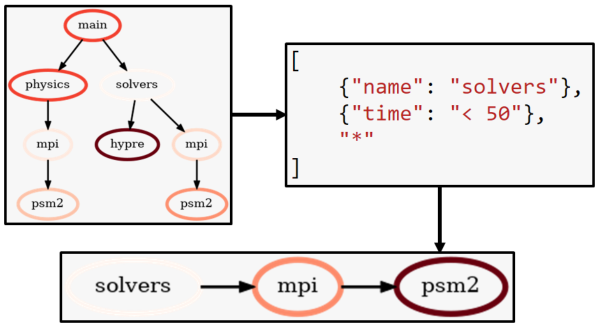
The primary data structure in Hatchet is called a *GraphFrame*, which consists of two components: a *Graph* defining the caller-callee relationships and a pandas *DataFrame* storing the categorical and numerical data associated with each node. Fig. 1a shows these two objects of a GraphFrame. Pandas [8], [11] is an open-source library in Python providing data structures and manipulation tools for data analysis. Pandas is well-suited for tabular data, making it a great option for storing Hatchet's data.

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 for that node. The frame for each node is determined by the file format readers (*e.g.,* HPCToolkit reader, Caliper reader), and is a dictionary of key-value pairs. The frame can classify the node type as a function/procedure node, a loop node, or a statement node.

Hatchet provides readers for several input formats from popular profiling tools, such as HPCToolkit's database directory and Caliper's json-split format. Additionally, Hatchet can read data from the GraphViz DOT format or data stored in a list of dictionaries. Once the data has been read into Hatchet, a user can perform operations to select, filter, or aggregate the GraphFrame. There are also operations that are applied to a single GraphFrame (*e.g.,* filter), while others are meant to compare across GraphFrames (*e.g.,* divide). In the following sections, we present new features that have been added to Hatchet since it was initially introduced in [6].

III. Query Language

We design a query language that enables data reduction using call path pattern matching. We demonstrate the capabilities of our augmented Hatchet by examining the performance of



```
[
    {"name": "solvers"},
    {"time": "< 50"},
    "*"
]
```

Fig. 2: Example using Hatchet's new 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.

MPI calls in three High Performance Computing (HPC) benchmarks (*i.e.,* AMG2013 [12], Kripke [13], and Lammps [14]) when using MVAPICH and Spectrum-MPI. In doing so, we identify hidden performance losses in specific MPI functions.

A. *Design and Implementation*

Previously, Hatchet did not provide a way to utilize the relational caller-callee data collected by HPC profilers in analysis, thus limiting the types of analysis a user can perform. To leverage this data in analysis, we design a graph query language that filters performance data through call path pattern matching. Our query language is based on Cypher [15] and GQL [16]. 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 data graphs in three steps. First, we match all real nodes in the graph to the *abstract nodes* in the user-provided *query path*. Next, we collect an exhaustive list of all paths in the graph that match the entire *query path*. Finally, we use the exhaustive list to create a new graph containing only the real nodes found in the list of matched paths. An example of this is shown in Fig. 2.

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 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.

B. *Case Study: Identifying Sources of Performance Losses*

We evaluate the effectiveness of Hatchet once augmented with our query language in identifying sources of performance losses associated with MPI calls in three HPC benchmarks (*i.e.,* AMG2013 [12], Kripke [13], and Lammps [14]). We use two different MPI libraries (i.e., MVAPICH and Spectrum-MPI) with 64, 128, 256, and 512 ranks on LLNL's Lassen supercomputer. We profile all the benchmark runs using HPC-Toolkit [5]. Because of our query language, we can 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 calls and children calls that are most important to the performance of the benchmark running with a particular MPI library.

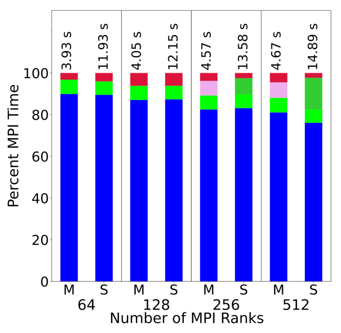
In this paper, we only show results related to the `MPI_Allgather` function in AMG2013 due to space constraints. Our reasons for this are two-fold. First, as shown in Fig. 3a, `MPI_Allgather` clearly comprises the majority of the MPI time spent in the AMG2013 benchmark. Second, the AMG2013 benchmark has the largest performance difference between MVAPICH and Spectrum-MPI. This suggests that, if we can determine a likely primary 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 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 AMG2013 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. 3b as most important to the performance of the program. The results of this reduction are shown in Fig. 3c.

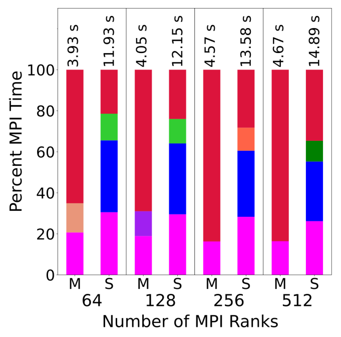
In our tests with MVAPICH and Spectrum-MPI, we determine that the `pthread_spin_lock` function is consistently a major contributor to MPI runtime (i.e., 10% or more of the MPI time, usually 20% or more). Additionally, when considering `MPI_Allgather`, we conclude that the worse performance of Spectrum-MPI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

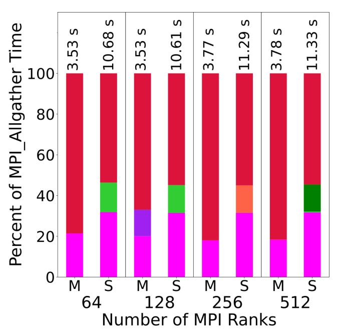
We improve both the Hatchet Jupyter-based and terminal-based tree visualizations. Our Jupyter visualization now has



(a)



(b)



(c)

MPI Function Calls

🟦	`MPI_Allgather`	🟪	`MPI_Waitall`
🟩	`MPI_Allreduce`	🟩	`MPI_Finalize`
🟥	Remaining MPI Time		

Child Function Calls

🟧	`<unknown file> [libmlx5.so.1.0.0]:1133`	🟪	`memset.S:1133`
🟦	`<unknown file> [libmlx5.so.1.0.0]:0`	🟧	`Geometry.h:0`
🟪	`pthread_spin_lock.c:26`	🟩	`malloc.c:0`
🟩	`stl_vector.h:0`	🟥	Remaining MPI Time

Fig. 3: The percent of (a) total MPI time in AMG2013 spent in MPI functions, (b) total MPI time in AMG2013 spent in the children calls of MPI functions, and (c) total `MPI_Allgather` time in AMG2013 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*. We denote the MVAPICH and Spectrum-MPI libraries by *M* and *S*, respectively.

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. Both visualizations have refined designs for readability.

A. Interactive Tree Visualization in Jupyter

A central design goal of Hatchet is easing analysis done on calling context trees and other similar performance data structures. 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 D3js [17] and Roundtrip [18].

The interactive tree permits selection of a single node on-click or multiple nodes by brush (drawing a gray box). Selection of a group of 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 data values. Selected nodes are outlined with a thick black line.

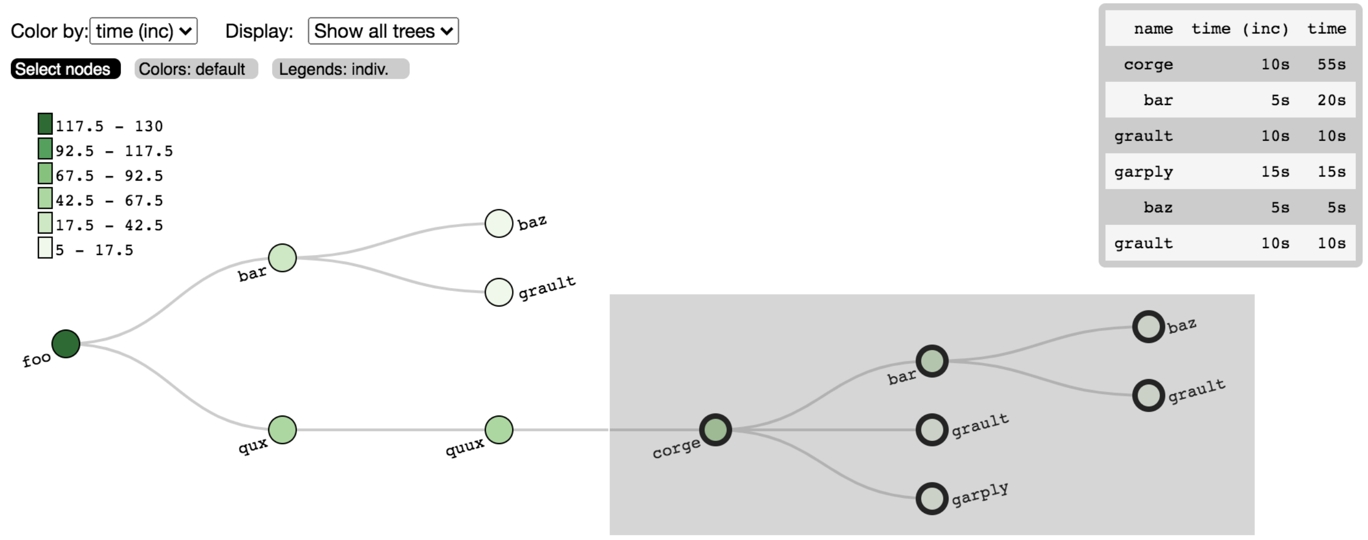
Once a user has drawn a selection over multiple nodes, the corresponding query can be accessed in any other Jupyter cell using the Roundtrip `fetchData` operation. As shown in Fig. 5, calling `fetchData(mySelection)` will return the corresponding syntax query based on the selection, and stores this into the `mySelection` Python variable. The variable is then used to filter the data by the specified query, producing the subtree that was selected in the interactive visualization. With the interactive tree, users can visually select nodes that can then be manipulated programmatically, allowing users to combine interactive visual selection with scripting.

Additionally, the Jupyter tree visualization has several interactive controls for adjusting the display. Available data values collected by node are available in the `Color by` drop down. In Fig. 4, inclusive time is selected to be displayed via color. When multiple trees are present, users can choose to display them all or just one, using a unified color scale, or separate ones. Color scales can also be inverted.

B. Tree Visualization in the Terminal

Hatchet provides its own visualization, whereby a string can be printed to the terminal to display the graph. We have redesigned the output and extended its functionality to provide users with more customization over their visualizations. Examples of the current terminal output can be seen in Fig. 6. 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 terminal visualization to control how many levels of the tree to output and how many digits of precision to output in the metric values.

We also provide users with increased control over the colormap. By default, the colormap annotates nodes with the highest metric value 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 easily invert the colormap by specifying `invert_colormap=True`. Additionally, Hatchet annotates nodes that may only exist in one of two GraphFrames as a result of a performing an arithmetic operation, such as subtraction. As shown in Fig. 6, the graphs of the two GraphFrames are structurally different, so we first unify the graphs before computing the difference. Unifying the graphs means that some nodes appeared in one GraphFrame, but not the other, and vice versa. We annotate those nodes with a red left arrow to indicate the node exists only in the left graph or a green right arrow to indicate the node only exists in the right.

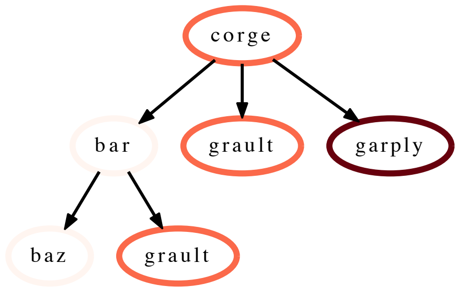

Color by: time (inc) Display: Show all trees

Select nodes Colors: default Legends: indiv.

🟩	117.5 – 130
🟩	92.5 – 117.5
🟩	67.5 – 92.5
🟩	42.5 – 67.5
🟩	17.5 – 42.5
⬜	5 – 17.5

name	time (inc)	time
corge	10s	55s
bar	5s	20s
grault	10s	10s
garply	15s	15s
baz	5s	5s
grault	10s	10s

Fig. 4: An example tree rendered using our interactive tree visualization in Jupyter. The user has drawn a gray box to select the `corge` subtree. Details of the selected nodes are shown in the upper right. This selection can be accessed in other Jupyter cells using the `fetchData` operation. Additional controls allow adjusting the color scale, changing the trees displayed, and changing the data displayed.



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

Fig. 5: Example demonstrating how the query (translated from a user's selection on the interactive tree) can be applied programmatically to filter the tree.

```
0.000 foo
├─ 5.000 bar
|  └─ 10.000 grault
└─ 0.000 qux
   └─ 5.000 quux
      └─ 10.000 corge
         ├─ 5.000 bar
         |  └─ 10.000 grault
         ├─ 15.000 garply
         └─ 10.000 grault
```

```
0.000 foo
├─ 5.000 bar
|  ├─ 5.000 baz
|  └─ 10.000 grault
└─ 0.000 qux
   └─ 5.000 quux
      └─ 10.000 corge
         ├─ 5.000 bar
         |  ├─ 5.000 baz
         |  └─ 10.000 grault
         └─ 10.000 grault
```

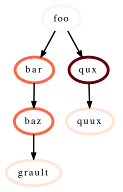
```
0.000 foo
├─ 0.000 bar
|  ├─ -5.000 baz ▶
|  └─ 0.000 grault
└─ 0.000 qux
   └─ 0.000 quux
      └─ 0.000 corge
         ├─ 0.000 bar
         |  ├─ -5.000 baz ▶
         |  └─ 0.000 grault
         ├─ 15.000 garply ◀
         └─ 0.000 grault
```

Fig. 6: Terminal-output tree visualizations for a subtraction of two GraphFrames in Hatchet (result graph on the bottom). For any node that only exists in one of the two graphs, its node will be annotated with a green or red arrow in the result graph. Here, the two `baz` nodes exist only in the right graph, and are annotated with a green arrow. The `garply` node exists only in the left graph, and is annotated with a red arrow.

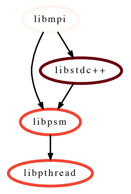
V. API Improvements

We describe four GraphFrame APIs (*i.e.*, *filter*, *groupby_aggregate*, *mul*, and *div*) that have been added to the Hatchet package since its initial release. The first two APIs operate on a single GraphFrame, while the latter two APIs operate on two GraphFrames. All of the operators enable deeper analysis of structured data and allow structured data to be manipulated in different ways.

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 be to keep all rows that have a time value greater than some threshold. The filter function has been extended to support taking a user-defined query path to filter the graph using the caller-callee relational data. Hatchet's new syntax query language is described in more detail in Section III. The resulting Series or DataFrame is used to filter the 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. The graph is rewired such that the nearest remaining ancestor is connected to the nearest remaining child on the path for all call paths.



node	module	name	time	time (inc)
{'name': 'foo'}	libmpi	foo	0.0	130.0
{'name': 'bar'}	libpsm	bar	5.0	20.0
{'name': 'baz'}	libpthread	baz	5.0	5.0
{'name': 'grault'}	libpthread	grault	1.0	8.0
{'name': 'qux'}	libstdc++	qux	10.0	55.0
{'name': 'quux'}	libpsm	quux	1.0	1.0



node	time (inc)	time	name	nid
{'name': 'libmpi', 'type': 'module'}	130.0	0.0	libmpi	0
{'name': 'libpsm', 'type': 'module'}	21.0	6.0	libpsm	1
{'name': 'libpthread', 'type': 'module'}	13.0	6.0	libpthread	2
{'name': 'libstdc++', 'type': 'module'}	55.0	10.0	libstdc++	3

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

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

groupby_aggregate: The groupby and aggregate operation takes a user-defined group and an aggregation function, and produces a re-organized DataFrame. By default, Hatchet's DataFrame is grouped by nodes. This operation is useful for aggregating the data into alternative groups for different analyses. As an example, users may want to look at the performance attributed to modules or function names (instead of nodes), and aggregate the data values accordingly.

As part of the groupby and aggregate operation, the graph is also reindexed to match the new groups in the DataFrame. For each group, the reindex phase merges all nodes belonging to the group into a single supernode. When a node is merged into a supernode, any of its edges to a parent or child are created as edges in the new supernode. For each supernode, we create a new node in Hatchet with the corresponding name and group, and use this as the index in the DataFrame. The `groupby_aggregate` returns a new GraphFrame with a reindexed graph and a grouped-aggregated DataFrame. Fig. 7 shows the graph before and after a groupby-aggregate is performed, specifying *module* as the new group.

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

div: The division (/) operation requires that the two graphs have the same structure. If this is true, then the division operator computes the element-wise division of two DataFrames. Otherwise, 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 massively parallel programs. These efforts target two critical functions in Hatchet: HPCToolkit reader and `unify`. We detail the optimization process and the results of these efforts in the following subsections.

A. Analysis Infrastructure

To enable performance analysis of Hatchet, we developed a custom-made cProfile [19] wrapper class. This class provides simple annotations for starting, stopping, and resetting the profiler in an application. Furthermore, we created several interfaces for aggregating and exporting the measured performance data for post-mortem analysis. This minimal profiling infrastructure provides developers with a workflow for quickly identifying bottlenecks within the Hatchet APIs.

B. HPCToolkit Reader

Of the many different profiler tools supported by Hatchet, HPCToolkit is more fine-grained in its metric collection when compared to other tools, resulting in larger and more complex datasets. As an example, an HPCToolkit profile of Lammps collected on 512 nodes produces a calling context tree (CCT) of over 34,000 call sites and approximately 50,000,000 performance data records. The large size of HPCToolkit's profiles made Hatchet's HPCToolkit reader an obvious first step towards extending Hatchet's support for big data. The following subsection discusses the primary bottleneck identified in the HPCToolkit reader and explains the optimization methodology.

1) Bottlenecks and Optimization: After initial analysis, we pinpoint the critical bottleneck 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, this function also derives exclusive timing metrics from inclusive metrics collected natively by HPCToolkit. Hatchet calculates the exclusive metrics for a node by subtracting the current node's inclusive metrics from its parent's. The few lines of Python code dedicated to this procedure dominates 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 the Hatchet 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 for each call site. This means that a given node ID appears in the DataFrame n times, where n is the total number of

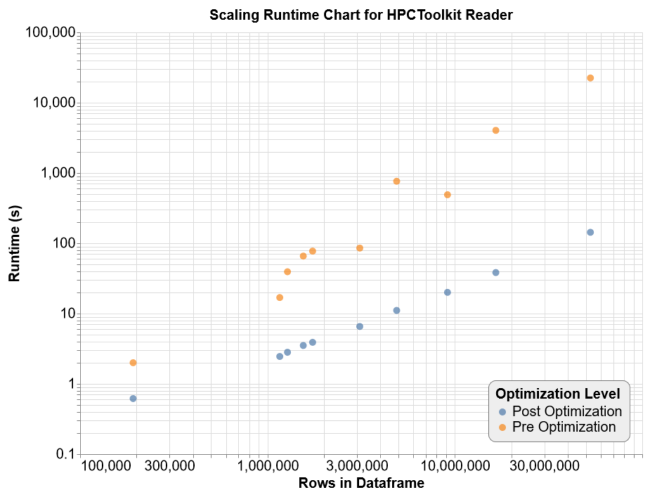


Fig. 8: Log-log plot showing performance before and after optimization of the HPCToolkit reader as the size of the Hatchet DataFrame increases. The optimized HPCToolkit reader scales significantly better compared to its unoptimized predecessor.

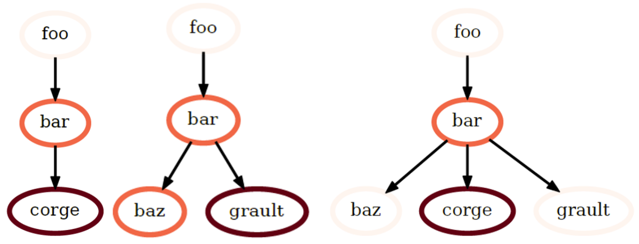
execution threads the profiled application was running on. For highly parallel programs, this can mean searching for tens of thousands of rows containing the same parent and child ID.

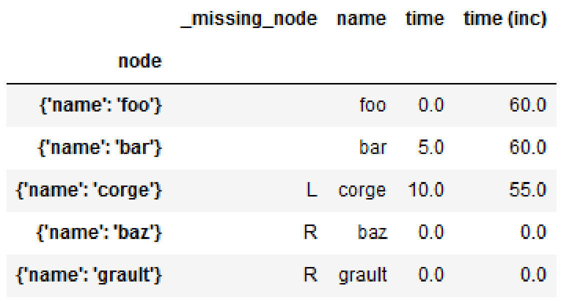
This significant duplication of rows, with slight variations in the metric data across threads, causes the Hatchet DataFrame to explode in size compared to its corresponding CCT. As the pandas DataFrame is not optimized to handle operations on very large datasets, operations such as conditional indexing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

Fortunately, the structure of this data as well as opportunities to speedup array-based operations provided by the C/Python hybrid language, Cython [20], made optimizing the conditional index operation simple. We first extract the relevant columns (*i.e.,* inclusive 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 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 parents metrics at each iteration. For the largest dataset tested, this reduces the number of iterations over our DataFrame (per function call) from greater than 100,000,000 to a little over 30,000.

2) Results: To examine the impact of our optimizations, we measured the runtime of Hatchet's HPCToolkit reader on a series of profiles which increase in size from 999 call graph nodes and 191,808 rows in our dataframe to 34,855 nodes and 53,537,280 rows. The scaling run was executed five times for each read first on the unoptimized code and again on the optimized code. These scaling profiles came from parallel runs of AMG2013 [12], Kripke [13], and Lammps [14] used to create benchmarks for the case study in Section III. The results of these trials are presented in Fig. 8. In this figure each datapoint represents the average runtime over five trials to read in a HPCToolkit profile of a particular size.





	_missing_node	name	time	time (inc)
node				
{'name': 'foo'}		foo	0.0	60.0
{'name': 'bar'}		bar	5.0	60.0
{'name': 'corge'}	L	corge	10.0	55.0
{'name': 'baz'}	R	baz	0.0	0.0
{'name': 'grault'}	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. The result is a single unified graph shown on the right. The DataFrame produced from the unify operation contains a new `_missing_node` column identifying which nodes were exclusive to each graph are shown with an L or R.

After examining this runtime data, we found that our optimizations improve the performance of Hatchet's HPCToolkit reader significantly. As can be seen in Fig. 8, the slowdown of the pre-optimized solution becomes more pronounced with larger DataFrames, while the optimized code scales consistently. The divergence demonstrates that the relative speedup of the optimized HPCToolkit reader will increase as datasets continue to grow in size. For a DataFrame containing 50,000,000 rows, the HPCToolkit reader went from read times of six hours and fifteen minutes to two minutes and twenty-four seconds, a reduction of two orders of magnitude.

C. Unify

The unify operation takes two distinct GraphFrames, unifies the graphs, and reindexes the DataFrames by the nodes in the unified graph. The updated GraphFrames contain the unified graph created from the union of the two graphs (as shown in Fig. 9) and a reindexed DataFrame containing all the nodes from both DataFrames. This updated DataFrame stores metadata about the origin of nodes with a column `_missing_node`, which denotes that a particular node existed only in the original left or right GraphFrame. If a row was shared by both DataFrames, then this column is left empty.

Although not the major bottleneck operation in Hatchet, we chose to optimize unify since it is a primary operation in most of Hatchet's binary APIs. Specifically, unify underlies all element-wise arithmetic operations, such as multiply or add. Section V discusses some of these APIs in more detail. Since these arithmetic 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 5,892 seconds or one hour and 38 minutes. Even when unifying smaller datatsets (100,000 rows and 1,000 nodes in the CCT), the unify operation is notably slow, consuming 30 seconds.

1) Bottlenecks and Optimizations: Mirroring the narrative of the HPCToolkit reader, we determine the runtime of unifying GraphFrames is dominated by the time spent updating the DataFrames. In contrast to HPCToolkit however, slowdowns are spread out among several pandas library operations in Hatchet's internal DataFrame management function, `_insert_missing_rows`.

Following a more mechanical approach to optimizing the pandas library, the initial performance enhancements involved tweaking existing code to follow pandas conventions. Assigning values in DataFrames in C-like loops are swapped out for assigning values using Python lists or NumPy arrays that are then assigned to a pandas DatatFrame column. Both native Python and NumPy handle single-element array access and their array/list creation are significantly better than pandas. This optimization provides some marginal speedup and reveals a need to start integrating Cython for more substantial performance gains.

The next bottleneck in unify exists within this same function, `_insert_missing_rows`, where a call is made to the pandas `isin` method. 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 — which makes up the vast majority of our test cases — pandas falls back to NumPy's `isin` functionality, which combines `np.unique` for sorting and a binary search for determining membership. This `isin` operation does not perform especially well with lists of complex objects, such as the nodes used by Hatchet.

After examining the source of 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 reduce 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 implement the `isin` function itself as a binary search inside of a loop over the searched-for array of elements. We further speed 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, ranks, 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 eliminate this layer of calls to "multi" and reduce the slowdown observed with many pandas methods.

2) Results: To measure the impact of our optimizations to the `unify` method, we collected the runtimes produced by joining two similar GraphFrames constructed from the same pool of HPCToolkit profiles we used to measure our HPCToolkit reader. The performance results, averaged over five

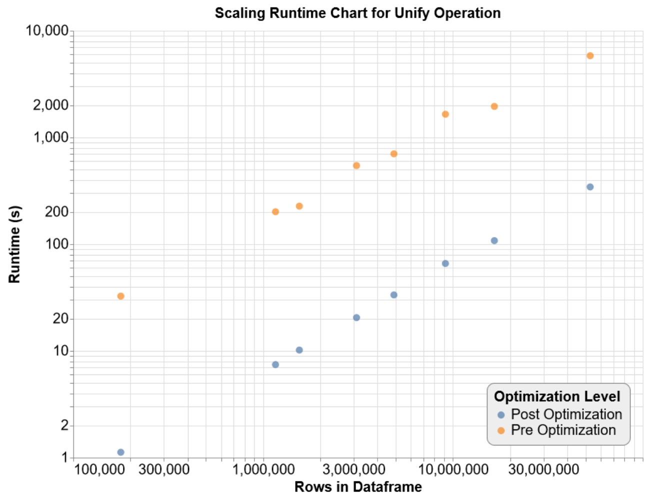


Fig. 10: Log-log plot showing the pre- and post-optimization performance of the `unify` operation as the size of the dataset increases.

```
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 this workflow from 14 hours to 10 minutes and 30 seconds.

trials per-unify are shown in Fig. 10. Because this experiment required pairs of datasets per run, there are two fewer data points in this figure compared to Fig. 8. The two missing profiles did not have a similar implementation to unify on so they were omitted from our scaling run.

For smaller datasets, we saw a reduction from 30 seconds to approximately 1 second. For mid-sized DataFrames containing millions of rows, `unify` went from minutes down to tens of seconds. For very large datasets, `unify`'s performance 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 trend 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, no algorithmic advantage exists like that produced by our HPCToolkit reader optimizations.

D. Workflow Improvements

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 arithmetic operators to make a quick comparison between the two runs, storing the result in a new GraphFrame. Before optimization, this program would take fourteen hours to compare two HPCToolkit profiles collected from a large program run on 512 nodes.

After integrating the optimizations detailed in this section, the overall runtimes for analyzing large datasets has been significantly reduced. The runtime for these operations has been reduced to ten minutes and thirty seconds for the same large profiles (80X improvement). The optimizations detailed in this section improved the performance for a key workflow in Hatchet.

VII. Related Work

There are many profilers available that can collect call graphs or call paths for post-hoc analysis [2]–[5], [21], [22]. There are also CCT-specific visualization tools, such as that provided by TAU, HPCToolkit's HPCViewer [23], CallFlow [24], [25], and flame graphs [26]. Currently, the most scalable tool for visualizing call paths is HPCTraceViewer [27], which shows the calls paths varied across time, MPI ranks, and threads. There are two limitations with existing tools. First, they are limited to their own data format, meaning they cannot import data from other tools. Secondly, they are limited to their own custom GUI interface for viewing the call graph, and do not provide flexibility to script analyses to manipulate the profile data. With Hatchet, we provide a canonical data format for profile data, so it can read data from several different profile tools. Additionally, Hatchet provides operators to automate the performance analysis of structured data without having to learn a new format or how to interface with a new GUI.

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 [28], for example, provides its own database, a GUI interface, and a custom data format known as PerfDMF [29]. 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 [30], [31]. To expand on this idea and to enable analysis of larger profiles, Tallent et. al extended HPCToolkit to include derived metrics [32], [33]. Since Hatchet is built upon the pandas data analysis library [8], [11], it provides a number of data manipulation APIs that are performant on large tabular data.

VIII. Conclusions and Future Work

Analyzing performance and pinpointing bottlenecks in the application are important to guiding application developers in the optimization workflow. It is a huge challenge to effectively analyze large codes that may contain several thousands lines of code and several unique nodes in the call graph or calling context tree. Many of the current tools are insufficient to provide programmatic capabilities for reproducible analysis.

In this paper, we provided an overview of four different efforts to extend Hatchet's profiling capabilities. We introduced Hatchet's new syntax query language, so users can specify expressions for filtering the graph. We also demonstrated Hatchet's new interactive visualization capabilities in Jupyter, enabling users to drag and drop a subtree to filter the graph. As a result of the drag and drop, the Jupyter integration will show the query that can be inserted into scripts for future analysis. We have also increased the flexibility of Hatchet's terminal tree output, such as inverting the colormap depending on which nodes a user wants to draw attention to and annotating nodes that may only exist in one graph or the other. We also provided an overview of some 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.

For future work, we plan to develop Hatchet's own data output format, enabling users to save out GraphFrames periodically throughout the analysis process. For large datasets that may take a significant amount of time to process, this capability will significantly improve the analysis workflow with Hatchet, since analysis can start from an intermediate step. We are looking into making this output format interoperable with databases, so Hatchet can store its data in a database in the future.

Acknowledgments

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. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014967

[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] 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.

[10] 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.

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

[12] U. Yang, R. Falgout, and J. Park, "Algebraic multigrid benchmark, version 00," 8 2017.

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

[14] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, p. 1–19, Mar. 1995.

[15] 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.

[16] A. Green, P. Furniss, P. Lindaaker, P. Selmer, H. Voigt, and S. Plantikow, "GQL scope and features," ISO, Tech. Rep., 2019.

[17] M. Bostock, V. Ogievetsky, and J. Heer, "D3: Data-driven documents," *IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis)*, 2011. [Online]. Available: http://vis.stanford.edu/papers/d3

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

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

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

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

[27] N. Tallent, J. Mellor-Crummey, M. Franco, R. Landrum, and L. Adhianto, "Scalable Fine-grained Call Path Tracing," Jun. 2011.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] N. R. Tallent, J. M. Mellor-Crummey, L. Adhianto, M. W. Fagan, and M. Krentel, "Diagnosing performance bottlenecks in emerging petascale applications," Nov. 2011.

[33] N. R. Tallent, L. Adhianto, and J. M. Mellor-Crummey, "Scalable identification of load imbalance in parallel executions using call path profiles," Nov. 2010.