Hatchet: Pruning the Overgrowth in Parallel Profiles

Abhinav Bhatele[†,⋆], Stephanie Brink[⋆], and Todd Gamblin[⋆]

[†]Department of Computer Science, University of Maryland, College Park
[⋆]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
bhatele@cs.umd.edu,brink2@llnl.gov,tgamblin@llnl.gov

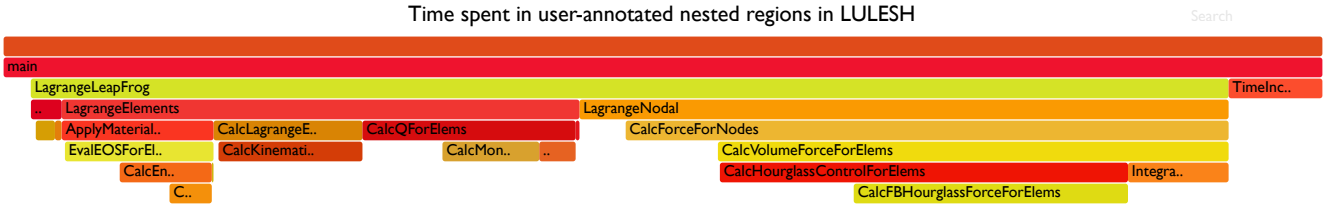


Figure 1: A flame graph representation of a user-annotated nested region profile of the LULESH proxy application.

ABSTRACT

Performance analysis is critical for eliminating scalability bottlenecks in parallel codes. There are many profiling tools that can instrument codes and gather performance data. However, analytics and visualization tools that are general, easy to use, and programmable are limited. In this paper, we focus on the analytics of structured profiling data, such as that obtained from calling context trees or nested region timers in code. We present a set of techniques and operations that build on the pandas data analysis library to enable analysis of parallel profiles. We have implemented these techniques in a Python-based library called Hatchet that allows structured data to be filtered, aggregated, and pruned. Using performance datasets obtained from profiling parallel codes, we demonstrate performing common performance analysis tasks reproducibly with a few lines of Hatchet code. Hatchet brings the power of modern data science tools to bear on performance analysis.

CCS CONCEPTS

• **General and reference** → *Performance*; • **Software and its engineering** → Software maintenance tools.

KEYWORDS

performance analysis, tool, parallel profile, calling context tree, call graph, graph analytics

ACM Reference Format:
Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. Hatchet: Pruning the Overgrowth in Parallel Profiles. In *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '19), November 17–22, 2019, Denver, CO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3295500.3356219

SC '19, November 17–22, 2019, Denver, CO, USA

ACM ISBN 978-1-4503-6229-0/19/11…$15.00
https://doi.org/10.1145/3295500.3356219

1 MOTIVATION

Understanding performance bottlenecks is critical to optimizing the performance of high performance computing (HPC) codes. Profiling tools [3, 5, 12, 21, 25] allow developers to focus their optimization efforts by pinpointing the parts within a code's execution that consume the most time. Without them, it would be extremely difficult to find performance problems, especially in modern applications, which can comprise millions of lines of code.

Attributing time to code can be tricky, and it requires a reasonable understanding of the structure of the program. Most basic profilers attribute time to functions or statements in the code. More sophisticated profilers can record time spent in different *call paths* or *calling contexts*. For example, the profiler would differentiate time spent in `MPI_Send` when it is called in a hydrodynamics routine from time spent in `MPI_Send` when it is called in a solver library. Such profilers maintain a prefix tree of unique calling contexts. Other profilers may attribute time to nodes in a static or dynamic *call graph*, in which case time would be attributed to nodes in the graph. So, code regions can be represented by a range of structures, from simple, "flat" strings, to nodes in trees or graphs. Figure 1 shows an example of a simple tree, where each node (each box in the flame graph) represents a code region annotated by the user.

Most profiling tools use their own unique format to store recorded data, and they may display this data as text or with a tool-specific viewer (typically a GUI). These tools are limited in the kinds of analysis they support, and they do not enable the end user to *programmatically* analyze performance data. Most profile data viewers provide a point-and-click-based workflow, with limited support for saving or automating analysis. As such, analyzing performance data can be very tedious, and using a new *measurement* tool often requires also using a new *analysis* tool to understand the data. Moreover, most tools only support viewing one or two call graphs

at a time. They are often insufficient for tasks like load balance analysis that require detailed averaging and clustering across ranks, threads, and time. They also lack general capabilities for effectively sub-selecting and focusing on specific *parts* of larger datasets.

Growing interest in data science has led to the availability of many fully-featured data analysis environments. Python and R in particular support *DataFrames* that blend features of traditional numerical computing environments such as MATLAB with database-derived features such as indexed queries, joins, and aggregations. Numerous plotting libraries and visualization tools are available to view data in these environments, all with programmable APIs. In addition, data analysis tools have more features, and are better maintained by open-source software communities than tool GUIs. While these environments can handle numerical and temporal indices with ease, they cannot handle structured datasets, such as profiles that are indexed by nodes in a tree or a graph.

This paper presents a set of techniques that allow modern data analysis libraries to be leveraged for parallel profile analysis. We have developed a canonical data model that can read, represent, *and index* the data generated by most profiling tools. We call this data model a *structured index*. We have developed techniques for selecting, filtering, and aggregating datasets with structured indices, and we have generalized these techniques for datasets that have hybrid indices over code, processes, threads, and time. Finally, we have implemented these techniques in a library that we call *Hatchet*, which builds on the popular pandas data analysis library [15, 16].

This paper makes the following contributions:

- A canonical data model that enables different types of profile data (e.g., HPCToolkit, Caliper, callgrind, gprof, perf) to be represented and analyzed in a common way;
- an indexing technique that allows structured graph or tree nodes to be used and queried as a DataFrame index;
- operators that allow structured data to be filtered, aggregated, and pruned to produce API-centric sub-profiles; and
- an implementation of these techniques in the Hatchet library that enables users to leverage modern data analysis approaches for analyzing large-scale call path profiling data.

Using performance datasets derived from running HPC codes, we also present several case studies that demonstrate performing common performance analysis tasks reproducibly with only a few lines of Hatchet code. Examples include: 1) identifying regions or callsites with the most load imbalance across MPI processes or threads; 2) filtering datasets by a metric or library/function names to focus on subsets of data; and 3) easily handling and analyzing multi-rank, profile data from multiple executions. We expect that Hatchet will make analysis of HPC performance data quicker, easier, and more reproducible.

2 BACKGROUND

We define the different kinds of structured profiles output by various profiling tools (in particular HPCToolkit and Caliper), and provide a brief background on the pandas data analysis library.

2.1 Structured Performance Data

Profiles can be collected either through *sampling* or through direct instrumentation. In a directly instrumented program, measurement code is typically inserted with a compilation tool, and data is collected at each instrumentation point. Sampled profilers instead periodically force an interrupt while a program runs, and data is collected at each interrupt. In either scenario, the collected data contains two types of information: *contextual information*, i.e., the current line of code, file name, the call path, the process ID, etc.; and *performance metrics*, such as the number of floating point operations or branch misses that occurred since the last sample. Depending on how these samples are aggregated, different types of profiles can be generated.

Calling Context Trees (CCT): *Callpath profilers* analyze the stack at runtime to determine the full *calling context* of each sample. Calling contexts, or call paths, refer to the sequence of function invocations that led to the sampled one. Each stack frame becomes a node in the CCT, and the path from the root of the tree to a given node represents the call path, or calling context, that led to the leaf invocation. A *calling context tree (CCT)* is a prefix tree of call paths. CCTs allow analysts to understand differences in function behavior that depend on *how* the function was called.

Call Graphs: *Call graph profiles* [7] do not perform the stack analysis required to generate CCTs; they attribute data only to the *name* of each function called. Edges in the call graph represent *static* calling relationships (i.e., that one function is called by another), averaged across all invocations regardless of origin. Samples can also be aggregated at the granularity of user-annotated regions, which can produce an even coarser tree or graph than call graphs. Call graphs are a more concise, (and sometimes clearer) representation than a CCT but they discard all context information.

The most insightful representation of a given performance profile depends on the problem being analyzed. Hatchet's data model is designed to handle structured profiling data generated at various granularities from different file formats. To motivate this, we provide a brief overview of two profiling tools in the next section.

2.2 Call Path Profilers

HPCToolkit: HPCToolkit [3] provides a suite of performance tools enabling measurement, analysis, correlation, and visualization. When asynchronous or synchronous events occur in the application, HPCToolkit records the full calling context as a CCT. With this data structure, a unique call path for a given node corresponds to the path from that node to the root. In HPCToolkit, a CCT node is not limited to function invocations only, but can also record loops, statements, and other code structures. Moreover, for parallel programs, HPCToolkit records the metrics per process and thread for every node in a unified CCT. The database generated by HPCToolkit's `hpcprof-mpi` utility is derived by modifying the CCT to include the node ID and timestamp. Since there can be multiple processes in a parallel run, `hpcprof-mpi` outputs the unified tree in XML format as well as separate binary files for each process that contain the metrics for all the nodes in the CCT.

Caliper: Caliper [5] provides a general abstraction layer for application performance introspection. Application developers can use Caliper's annotation APIs to collect performance information. It has a flexible data aggregation model [4] designed to handle a

wide variety of information that can be analyzed offline or in situ. At runtime, Caliper builds a generalized context tree consisting of attributes representing data elements. The context information for any given node can be derived by collecting all attributes on the path between the node and the root node. User annotations in the code or enabling the call path service in Caliper can generate a structured profile or CCT. Caliper supports JSON output formats that can be generated by either running `cali-query` on the raw Caliper samples or by enabling the `mpireport` service.

2.3 Pandas and DataFrames

Pandas: *pandas* is an open-source Python library providing data structures and tools for data analysis [15, 16]. It is built on top of NumPy and is well-suited for various kinds of data including tabular and statistical datasets. Pandas provides two main data structures: *Series* and *DataFrame*.

Series: A Series is a one-dimensional, homogeneously-typed array that has an associated *index*. Unlike a traditional array, the index need not be numerical; a Series can be indexed by any ordered, comparable data type. In this sense a Series in pandas is somewhat like a sorted dictionary or hashtable, as it allows fast lookup by non-numerical values.

DataFrame: A DataFrame is a two-dimensional tabular structure with potentially heterogeneously-typed columns. Each column in a DataFrame can be thought of as its own Series, and certain columns can be made the *index* of the DataFrame. Columns have titles, or *labels* that can be used to retrieve their corresponding Series, and the values in index columns can be thought of as keys for fast lookup of rows in the DataFrame. DataFrames with like indices can thus be aligned and compared. In addition to indexing, DataFrames support a wide range of functionality borrowed from the world of spreadsheets and SQL databases: operations to handle missing data, slicing, fancy indexing, subsetting, inserting and deleting columns, merging datasets, aggregating or grouping data, etc. Most importantly, multi-indexing a DataFrame provides an intuitive way of working with high-dimensional data in a two-dimensional data structure.

MultiIndex: pandas allows designating multiple columns in a DataFrame to be a composite *MultiIndex*. This enables users to easily store and manipulate data with an arbitrary number of dimensions. For example, in parallel performance analysis, we may want to index data not only by a calling context or other structured code identifier, but also by MPI rank, node hostname, or thread id. Pandas makes this natural; we can easily add data from additional levels of parallelism by adding additional index columns to a MultiIndex.

3 THE HATCHET LIBRARY

We have created Hatchet, a Python library that builds on top of pandas so that it can deal with structured profiling data such as calling context trees (CCTs) and call graphs. As mentioned in the previous section, pandas provides easy ways to manipulate data in Series and DataFrames, and it has support for arbitrary-dimensional indexes through MultiIndexes. However, pandas cannot handle structured datasets such as profiles that refer to source code, and are indexed by nodes in a tree or a graph. Pandas by default handles indexes of numbers, text, or dates, but these are all essentially linear data spaces. Call trees and call graphs have nonlinear node and edge structures, and we want to be able to preserve the ability to reason about graph- and tree-based relationships like parent, ancestor, and child. To overcome this, Hatchet provides data structures that enable indexing rows of a DataFrame by nodes in the graph.

3.1 Structured Indexes

We have developed a canonical data model that can represent and *index* the data generated by most profiling tools. We call this data model a *structured index*. This index enables nodes in a structured graph or tree to be used as a DataFrame index. There can be multiple types of nodes, such as procedure/function nodes, loop nodes (representing loop structures), and statement nodes (leaf-level nodes).

Hatchet's structured index is, at the most basic level, an in-memory graph. The structure of the graph is shown in Figure 2. This particular graph happens to be a tree, but Hatchet can support both call graphs and call trees. In the example graph, a function `A` calls a function `B` and a function `C`, and `B` contains a loop nest, `D`. These code structures span two libraries, or "modules", `libfoo.so` and `libbar.so`.

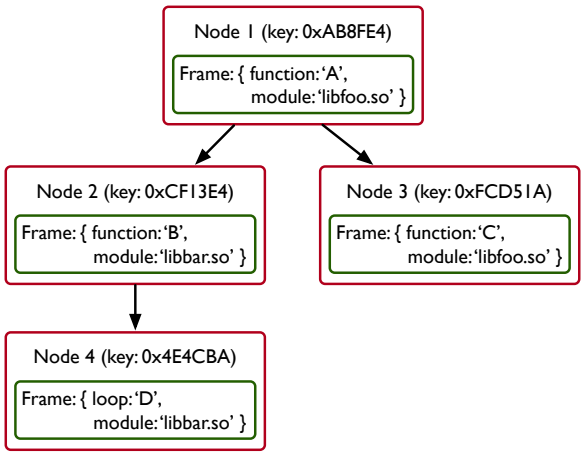


Figure 2: Hatchet's graph object, showing nodes and Frames.

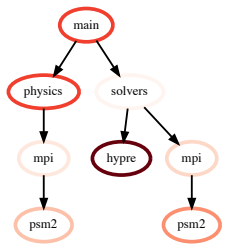
3.1.1 Generic Frames. Each node in Hatchet's graph contains a generic identifier for the code construct it represents. We call this identifier a *Frame* (after a *call frame*). Frames are generated by data sources such as file readers; they contain a set of key/value pairs that describe the source code the node represents. Depending on the type and amount of data the reader provides, this may be as simple as a function name or region name. It may also be more complex, including file and line number, code module, or other data from the particular input tool. Frames support basic comparison operations, such as ==, >, <, etc., which are evaluated based on the names and values of component fields. If two Frames do not have completely identical keys *and* values, they are not considered equal. The key design point here is that there is not a rigid schema for the Frames; they are *generic* and can be generated by each data

source according to its granularity. Hatchet attempts to regularize the *names* of fields as much as possible over different sources to enable comparison of data across measurement tools, but this is not a requirement.

3.1.2 Nodes. Frames are associated with nodes in the Hatchet graph, and node objects define connectivity and structure of the Hatchet model. Each node knows its children and its ancestors in the graph, and each node has a unique *key*. The key is not meant to be accessed by Hatchet users. Rather, like Frames, Hatchet nodes expose their *own* comparison operations (==, >, <, etc.), which opaquely operate on this key. This means that we can insert Node objects directly into a pandas DataFrame column and make it an *index*. By default, we use the Python `id()` function for the node key. This is equivalent, roughly, to C's & operator, in that it returns an integer representing the address of the Python object in memory. We require only that the node key be *unique* for each node. We can optionally use keys that provide certain useful orderings (like pre-order, post-order, etc.), *if* we want to pay the cost of a graph traversal (or sort) to generate more structured keys. We default to only guaranteeing uniqueness and not order in our keys.

3.2 GraphFrame

The central data structure in the Hatchet library is a *GraphFrame*, which combines the structured index *Graph* with a pandas *DataFrame*. Figure 3 shows the two objects in a GraphFrame – a graph object (the index), and a DataFrame object storing the metrics associated with each node.



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

Figure 3: In Hatchet, the GraphFrame consists of a graph and a DataFrame object.

Because of the way we have architected the structured index Graph, we can insert Node objects directly into the pandas DataFrame. The nodes are sorted using their basic comparison operators, which operate on their `key` attribute. Thus, the first column in the DataFrame (the node) is the index column. As a convenience, we may also add columns (like name) based on attributes from each node's Frame. For example, in the figure, we have added the `name` and `nid` columns from the `Frame` subclass for HPCToolkit. This allows us to use regular pandas operations (selection, filtering) on these values directly. As we will see later, the `node` column itself also allows various graph-semantic functions to be used, as well. Finally, in addition to the identifying information for each node, we *also* add columns for each associated performance metric (inclusive and exclusive time in the figure).

Graphs vs. Trees: Hatchet stores the structure (typically a prefix tree generated from call paths) in the input data as a directed graph (instead of a tree) for two reasons. First, subsequent operations on a tree can create edges or merge nodes, turning the tree into a graph. Additionally, output from tools such as callgrind is already in the form of a DAG. Hatchet's directed graph could be connected or have multiple disconnected components. Each entity in the graph, such as a callsite, procedure frame, or function, is stored as a node and the caller-callee relationships are stored as directed edges. Each node in the graph can have one or multiple parents and children.

Benefits of DataFrames: We use a pandas DataFrame to store all the numerical and categorical data associated with each node. Profile data can be inherently high-dimensional when metrics are gathered per-MPI process and/or per-thread. In such cases, each node in the call tree or graph has metrics per-MPI process and/or thread and this data needs to be stored and indexed hierarchically. To index the rows of the data frame in such cases, a `MultiIndex` consisting of the structured index for the node and MPI rank or thread ID is used. In the most general case, a row in the data frame is indexed by a process and/or thread ID (and any other needed identifiers in even higher dimensional cases).

3.3 Immutable Graph Semantics

Astute readers may have noted that we are adding direct references to graph nodes into the DataFrame. The risk this poses in our API is that client code can extract a subset of a DataFrame and hand it off to *other* client code, which then modifies the graph index nodes directly and corrupts all DataFrames that use the same nodes. One key aspect of Hatchet is that its graph nodes use *immutable* semantics. The GraphFrame API is responsible for ensuring that operations *between* any two GraphFrames use immutable graph node references, and that any operations that would modify a graph node in place instead create an *entirely new* graph index for the new GraphFrame to work with. So, we implement immutable semantics using copy-on-write to simplify the management of the graph and DataFrame together.

One further consequence of our index model is that to use two DataFrames *together*, we require that their graphs be *unified*. That is, that they share the same index. This should be obvious when considering that the nodes are compared by their `key` values, and two nodes can only be considered identical within an index if they have identical keys, which means that they *must* be in the same graph for comparison to make sense. We accomplish this by traversing the graphs and computing their *union* according to their connectivity and Frame values (described further in the API section). Incidentally, this type of restriction is not unusual in pandas, where comparing two data frames frequently requires reconciling their indexes, as well. We abstract the details of these graph operations in Hatchet through the GraphFrame API, which determines when and how GraphFrames should be unified.

3.4 Reading a CCT Dataset

With all of these components, the structured index Graph models the edge relationships between nodes in the structured data, *and* a DataFrame stores the numerical (performance metrics such as time, performance counter data, etc.) and categorical data (e.g., load

module, file, line number) associated with each node. The generality of what can be stored in a pandas DataFrame enables Hatchet to store almost any kind of contextual information recorded during sampling by diverse profiling tools.

Hatchet provides readers for several input formats to support data collected by popular profiling tools in the HPC community. Hatchet can read in the database directory generated by HPCToolkit (`hpcprof-mpi`), and also JSON files generated by Caliper. In addition, one can provide structured data in the Graphviz DOT format, or simple dictionary and list literals in Python.

Most profiling tools that generate CCTs have two kinds of information in their output, often separated into different parts of a file or different files. The first information is the structure of the CCT – present in `experiment.xml` in HPCToolkit databases, and the `nodes` section of a Caliper JSON file. The second piece of information is the performance metrics attached to each node – available in `metric-db` files in HPCToolkit data and in the *data* section of a Caliper JSON file. The readers in Hatchet read in both pieces of information. The CCT structure is used to construct the graph object of the GraphFrame and the performance metrics are used to construct the DataFrame object. As the readers construct these two objects, they also make connects between the graph and DataFrame objects using the structured index.

4 THE HATCHET API

We now describe some of the operators provided by the Hatchet API that allow structured data to be manipulated in different ways: filtered, aggregated, pruned, etc. Even though all of the operations below are performed on the GraphFrame, some only modify the DataFrame, some only modify the graph, and others modify both. They are categorized accordingly in the following sections. Note that we consider a graph to be immutable, so any operations that lead to changes in the graph structure will create a new graph and return a new GraphFrame indexed by the new graph's nodes.

4.1 DataFrame Operations

filter: Filter takes a user-supplied function and applies that to all rows in the DataFrame. The resulting Series or DataFrame is used to filter the DataFrame to only return rows that are true. The returned GraphFrame preserves the original graph provided as input to the filter operation. Figure 4 shows a DataFrame before and after a filter operation. In this case, the applied function returns all rows where time is greater than 10.0.

Filter is one of the operations that leads to the graph object and DataFrame object becoming inconsistent. After a filter operation, there are nodes in the graph that do not return any rows when used to index into the DataFrame. Typically, the user will perform a squash on the GraphFrame after a filter operation to make the graph and DataFrame objects consistent again.

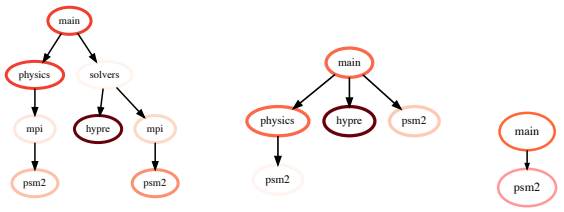
drop_index_levels: When there is per-MPI process or per-thread data in the DataFrame, a user might be interested in aggregating the data in some fashion to analyze the graph at a coarser granularity. This function allows the user to drop the additional index columns in the hierarchical index by specifying an aggregation

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

node	name	nid	node	time	time (inc)
main	main	0	main	40.0	200.0
physics	physics	1	physics	40.0	60.0
psm2	psm2	3	psm2	15.0	15.0
hypre	hypre	5	hypre	65.0	65.0
psm2	psm2	7	psm2	25.0	25.0

```
1 gf = GraphFrame( ... )
2 filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
```

Figure 4: The DataFrame before (left) and after (right) a filter operation on the time column.




```
1 filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
2 squashed_gf = filtered_gf.squash()
3
4 filtered_gf = gf.filter(
5     lambda x: x['name'] in ("main", "psm2"))
6 squashed_gf = filtered_gf.squash()
```

Figure 5: A graph before (left) and after two squashes (middle, right) on the GraphFrame.

function. Essentially, this performs a `groupby` and `aggregate` operation on the DataFrame. The user-supplied function is used to perform the aggregation over all MPI processes or threads at the per-node granularity.

update_inclusive_columns: When a graph is rewired (i.e., the parent-child connections are modified), all the columns in the DataFrame that store inclusive values of a metric become inaccurate. This function performs a post-order traversal of the graph to update all columns that store inclusive metrics in the DataFrame for each node.

4.2 Graph Operations

squash: The squash operation is typically performed by the user after a filter operation on the DataFrame. As shown in Figure 5, the squash on line 2 removes nodes from the graph that were previously removed from the DataFrame due to a filter operation. When one or more nodes on a path are removed from the graph, the nearest remaining ancestor is connected by an edge to the nearest remaining child on the path. All call paths in the graph are re-wired in this manner.

In some cases, a squash may need to merge nodes. The filter and squash calls on lines 4-6 remove the `physics` and `hypre` nodes from the graph, but `main` must now connect to *both* `psm2` nodes. In a calling context tree, a node cannot have two children with identical frames, so we merge the `psm2` nodes together. The graph now represents *only* the time spent in `psm2` when called directly *or* transitively from `main`. As mentioned earlier, node merging can convert a tree into a graph. Squash and other Hatchet API calls are general and handle both trees and graphs.

A squash operation creates a new DataFrame in addition to the new graph. The new DataFrame contains all rows from the original DataFrame, but its index points to nodes in the new graph. Additionally, a squash operation will make the values in all columns containing inclusive metrics inaccurate, since the parent-child relationships have changed. Hence, the squash operation also calls `update_inclusive_columns` to make all inclusive columns in the DataFrame accurate again.

equal: This checks whether two graphs have the same nodes and edge connectivity when traversing from their roots. If they are equivalent, it returns true, otherwise it returns false.

union: The union function takes two graphs and creates a unified graph, preserving all edges structure of the original graphs, and merging nodes with identical context. When Hatchet performs binary operations on two GraphFrames with unequal graphs, a union is performed beforehand to ensure that the graphs are structurally equivalent. This ensures that operands to element-wise operations like add and subtract, can be aligned by their respective nodes.

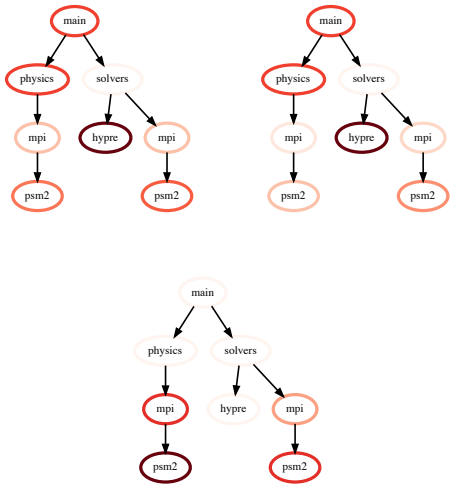
4.3 GraphFrame Operations

copy: The copy operation returns a shallow copy of a GraphFrame. It creates a new GraphFrame with a copy of the original GraphFrame's DataFrame, but the *same* graph. As mentioned earlier, graphs in Hatchet use immutable semantics, and they are copied only when they need to be restructured. This property allows us to reuse graphs from GraphFrame to GraphFrame if the operations performed on the GraphFrame do not mutate the graph.

unify: Similar to `union` on graphs, unify operates on GraphFrames. It calls union on the two graphs, and then reindexes the DataFrames in both GraphFrames to be indexed by the nodes in the unified graph. Binary operations on GraphFrames call unify which in turn calls union on the respective graphs.

add: Assuming the graphs in two GraphFrames are equal, the add (+) operation computes the element-wise sum of two DataFrames. In the case where the two graphs are not identical, `unify` (described above) is applied first to create a unified graph before performing the sum. The DataFrames are copied and reindexed by the combined graph, and the add operation returns new GraphFrame with the result of adding these DataFrames. Hatchet also provides an in-place version of the add operator: + =.

subtract: The subtract operation is similar to the add operation in that it requires the two graphs to be identical. It applies union and reindexes DataFrames if necessary. Once the graphs are unified, the subtract operation computes the element-wise difference between the two DataFrames. The subtract operation returns a new GraphFrame, or it modifies one of the GraphFrames in place in the case of the in-place subtraction (− =). Figure 6 shows the subtraction of one GraphFrame from another and the graph for the resulting GraphFrame.




```
1 gf1 = GraphFrame( ... )
2 gf2 = GraphFrame( ... )
3
4 gf2 -= gf1
```

Figure 6: Subtraction operation on two GraphFrames (resulting graph at the bottom).

4.4 Visualizing Output

Hatchet provides its own visualization as well as support for two other visualizations of the structured data stored in the graph object. The native visualization in Hatchet is a string that can be printed to the terminal to display the graph. Hatchet can also output the graph in the DOT format or a folded stack used by flame graph [8].

The dot utility in Graphviz produces a hierarchical drawing of directed graphs, particularly useful for showing the direction of the edges. Flame graphs are useful for quickly identifying the performance bottleneck, that is the box with the largest width. The y-axis of the flame graph represents the call stack depth. Figure 7 shows the same Hatchet graph presented in the three supported visualizations: terminal, DOT, and flame graph. For particularly large graphs, these visual representations can be useful for quickly identifying caller-callee relationships. However, identifying performance bottlenecks or load imbalance might be easier in the DataFrame.

5 PERFORMANCE

It is vital that performance analysis tools have low overheads and that they enable quick analysis of performance datasets without the user having to wait for a long time for each operation to complete. In Figure 8, we provide execution times for some operations in

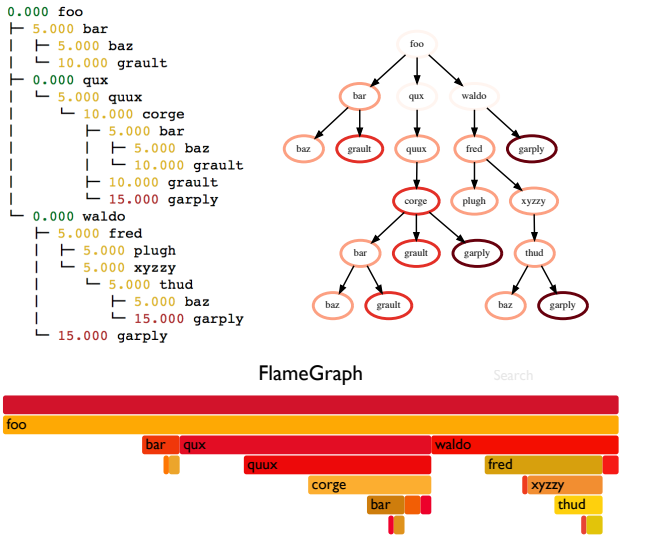


Figure 7: Visualization outputs supported in Hatchet include terminal output (left), DOT (right), and flame graph (bottom).

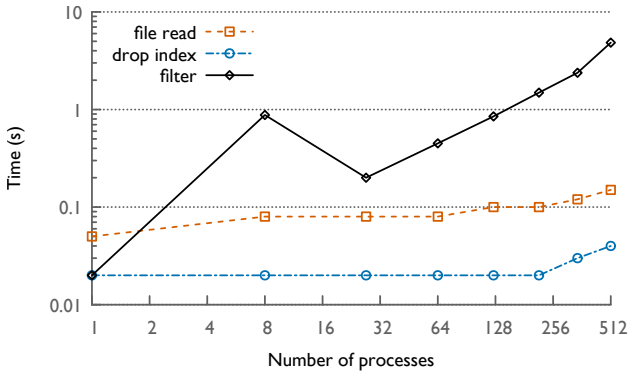


Figure 8: Performance overheads for different operations in Hatchet shown on a logarithmic scale. *file read* is the time to convert the data into the Hatchet representation, *drop index* and *filter* are the time to complete the `drop_index_levels` and `filter` operations, respectively.

Hatchet when using increasingly large datasets. We ran LULESH to generate Caliper profiles on 1 to 512 cores. LULESH requires a cubed number of processes. Hatchet was run on a relatively slow macOS laptop (1.8 GHz Intel Core i5). In the plot, *file read* is the time to read the input dataset into memory and convert it into the Hatchet data representation (graph and DataFrame). *drop index* represents the `drop_index_levels` operation, which we use to aggregate the per process information. If we apply a filter after dropping the second index (MPI rank), the filter operation takes a constant amount of time (~ 0.2 seconds). Hence, in the plot, the time shown for filter is measured for the case when filter is done without aggregating the per-process information. We see that the time increases linearly with the increase in the size of the dataset (both axes have a logarithmic scale).

Hatchet only adds a modest amount of code on top of the pandas library. Currently, the Hatchet code is nearly 2,400 lines of Python (obtained using `sloccount` [26]). We expect it to grow modestly as we add more readers and operations to it.

6 CASE STUDIES

In this section, we present several case studies demonstrating how common performance analyses can be executed in an automated manner using the Hatchet API and a few lines of Python code. The first set of case studies analyze single execution profiles for two scientific proxy applications, while the second set of case studies compare profiles from multiple executions.

6.1 Experimental Setup

We performed our single- and multi-node experiments on the Quartz supercomputer at Lawrence Livermore National Laboratory (LLNL). Each node of Quartz contains two Intel Broadwell processors with 36 cores per node. Our case studies used two scientific proxy applications. LULESH [1] is a Lagrangian shock hydrodynamics mini-application that solves a Sedov blast problem. For these case studies, we instrumented the LULESH code with Caliper annotations to collect performance metrics in Caliper's split JSON format. The second proxy application we used was Kripke [2, 13], which simulates neutron transport. We used HPCToolkit to generate the execution profiles of Kripke.

6.2 Analyzing a Single Execution Profile

Analyzing the profiling output from a single application execution is a fairly common performance analysis task. Typically, end users or performance researchers profile their code on a platform using a number of processes where they expect or have witnessed a performance degradation, and then analyze the output of such profiling. One of the most common tasks is to pin-point the regions of code or functions where the code spends most of its time. This is traditionally called a flat profile because the calling context is lost and we just get a flat view of functions or statements or code regions.

Flat profiles: Flat profiles can be easily generated in Hatchet using the `groupby` functionality in pandas. The flat profile can be based on any categorical column (e.g., function name, load module, file name). Similar to the sort feature in perf, the flat profile groups the nodes by the specified categorical column. Figure 9 shows the code to generate a flat profile by applying a `groupby` operation on the DataFrame object. The data read into Hatchet was generated by profiling 20 time steps of Kripke using HPCToolkit. We can transform the CCT generated by HPCToolkit into a flat profile by specifying the column on which to apply the `groupby` operation and the function to use for aggregation. In this case, we use `sum` to get the total time spent in each function.

Load imbalance: When program developers run their code on a large number of MPI processes, load imbalance across processes is often a scaling bottleneck. Hatchet makes it extremely easy to

name	nid	time	time (inc)
<unknown file> [kripke]:0	17234	1.825282e+08	1.825282e+08
Kernel_3d_DGZ::scattering	60	7.669936e+07	7.896253e+07
Kernel_3d_DGZ::LTimes	30	5.010439e+07	5.240528e+07
Kernel_3d_DGZ::LPlusTimes	115	4.947707e+07	5.104498e+07
Kernel_3d_DGZ::sweep	981	5.018862e+06	5.018862e+06
memset.S:99	3773	3.168982e+06	3.168982e+06
memset.S:101	3970	2.120895e+06	2.120895e+06
Grid_Data::particleEdit	1201	1.131266e+06	1.249157e+06
<unknown file> [libpsm2.so.2.1]:0	324763	9.733415e+05	9.733415e+05
memset.S:98	3767	6.197776e+05	6.197776e+05

module	nid	time	time (inc)
'Kripke/build-mvapich2.3/kripke	14366	1.825802e+08	5.847993e+08
i4/gcc-4.9.3/hpctoolkit-develop-olkit/ext-libs/libmonitor.so.0.0.0	2512	0.000000e+00	1.918548e+08
/usr/lib64/ld-2.17.so	9676	0.000000e+00	9.340625e+02
/usr/lib64/libc-2.17.so	37970	0.000000e+00	7.150550e+06
/usr/lib64/libdl-2.17.so	4427	0.000000e+00	2.804062e+02
/usr/lib64/libpsm2.so.2.1	433252	0.000000e+00	2.496037e+06
/usr/lib64/libpthread-2.17.so	2679	0.000000e+00	4.674375e+02
c-4.9.3/lib64/libstdc++.so.6.0.20	14945	0.000000e+00	3.898480e+05
npiler/lib/intel64_lin/libintlc.so.5	1215	0.000000e+00	9.357812e+01
3-intel-18.0.1/lib/libmpi.so.12.1.1	126726	0.000000e+00	7.962225e+06

file	nid	time	time (inc)
own file> [kripke]	50314	3.651083e+08	1.802709e+09
t86_64/memset.S	26693	6.148785e+06	1.229496e+07
[libpsm2.so.2.1]	783495	1.041419e+06	3.537456e+06
i/malloc/malloc.c	180270	9.252864e+05	1.859379e+06
em/handlemem.c	10844	4.230814e+05	6.440107e+05
src/mpidi_calls.c	17239	2.530799e+05	2.607438e+05
src/psm_queue.c	73291	2.066301e+05	1.599298e+06
i/pt2pt/testany.c	5704	1.746973e+05	1.605984e+06
/syscall-template.S	13787	6.691503e+04	1.338301e+05
[libmpi.so.12.1.1]	24482	2.587261e+04	5.043725e+04

```
gf = GraphFrame.from_hpctoolkit('kripke')

grouped = gf.dataframe.groupby('name').sum() # replace 'name' with 'module' or 'file'
```

Figure 9: Generating a flat profile in Hatchet using the `groupby` functionality of pandas. Traditional tools create a flat profile based on function names or callsite labels. In Hatchet, you can choose any categorical column to group by: name of the function (left figure), load module (middle figure), or file (right figure).

study load imbalance across processes or threads at the per-node granularity (call site or function level). A typical metric to measure imbalance is to look at the ratio of the maximum and average time spent in a code region across all processes. If the maximum-to-average ratio is high, it represents heavy imbalance. On the other hand, if the ratio is close to one, that signifies a well-balanced code.

Figure 10 shows the code for calculating an imbalance metric in an execution profile. We perform a `drop_index_levels` operation on the GraphFrame in two different ways: by providing mean as a function in one case and max as the function to another copy of the DataFrame. This generates two DataFrames, one containing the average time spent in each node, and the other containing the maximum time spent in each node by any process. If we divide the corresponding columns of the two DataFrames and look at the nodes with the highest value of the max-to-average ratio, we have located the nodes with highest imbalance. The figure shows all the nodes in LULESH with an imbalance greater than 2.0.

node	name	nid	time	time (inc)	imbalance
LagrangeNodal	LagrangeNodal	3.0	2.242594e+06	2.593621e+07	2.494720
main	main	0.0	1.106013e+05	5.357208e+07	2.161845
CalcForceForNodes	CalcForceForNodes	4.0	1.033639e+06	2.369361e+07	2.142526
CalcQForElems	CalcQForElems	16.0	3.351894e+06	6.649351e+06	2.037651
CalcEnergyForElems	CalcEnergyForElems	22.0	1.571996e+06	2.807323e+06	2.013174
CalcPressureForElems	CalcPressureForElems	23.0	1.235327e+06	1.235327e+06	2.005437

```
gf1 = GraphFrame.from_caliper('lulesh-512cores')
gf2 = gf1.copy()

gf1.drop_index_levels(function=np.mean)
gf2.drop_index_levels(function=np.max)

gf1.dataframe['imbalance']
    = gf2.dataframe['time'].div(gf1.dataframe['time'])
```

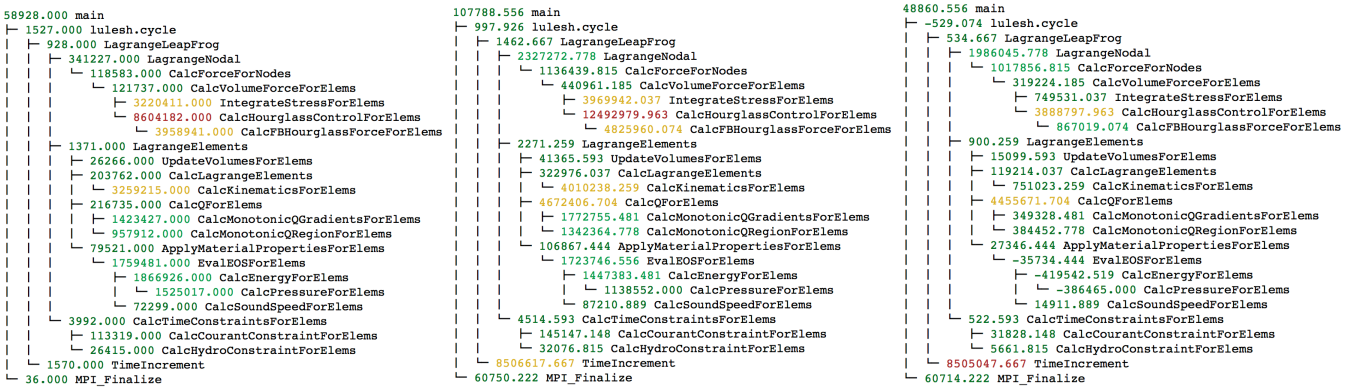
Figure 10: Load imbalance within a single execution is derived by calculating the mean and maximum values of a metric at each node across all MPI processes or threads and then dividing the two values for each node.

6.3 Comparing Execution Profiles

Another important task in parallel performance analysis is comparing the performance of an application on two different thread counts or process counts. This typically entails generating two sets of profiles on the different process counts in question and then comparing them in a GUI. Most tool GUIs do not provide automated ways to compare multiple datasets. As a result, in most cases the user manually goes over the two datasets in two instances of the tool to look for areas of the tree or graph where the performance looks different. This can be extremely cumbersome, inefficient and in many cases, ineffective. The filter, squash and subtract operations provided by the Hatchet API can be extremely powerful in comparing profiling datasets from two executions.

On-node scaling: In the first example, we ran LULESH in two modes: on a single core of a node and using 27 cores on a node. We generated profiles for the two executions and we wanted to identify the most important code regions in LULESH with respect to increase in time as one scales on node. Figure 11 presents the code to do such analysis. We read in the two profiles into two different GraphFrames and do a subtract (after dropping the additional index level using a mean function). The GraphFrame returned by the subtraction shows the nodes that have the largest increase in execution time. Although the nodes with the largest absolute time in the two cases is `CalcHourglassControlforElems`, the largest increase in time between the two executions happens on the `TimeIncrement` node (time shown in red in the right graph visualization).

Note that we do not need to visualize the graph to find the nodes with bottlenecks, especially when the graphs are very large. We can analyze the DataFrame of the GraphFrame returned by the subtract operation. Each column in the new DataFrame contains




```
gf1 = GraphFrame.from_caliper('lulesh-1core.json')
gf2 = GraphFrame.from_caliper('lulesh-27cores.json')

gf2.drop_index_levels()

gf3 = gf2 - gf1
```

node	name	nid	time	time (inc)
TimeIncrement	TimeIncrement	25.0	8.505048e+06	8.505048e+06
CalcQForElems	CalcQForElems	16.0	4.455672e+06	5.189453e+06
CalcHourglassControlForElems	CalcHourglassControlForElems	7.0	3.888798e+06	4.755817e+06
LagrangeNodal	LagrangeNodal	3.0	1.986046e+06	8.828475e+06
CalcForceForNodes	CalcForceForNodes	4.0	1.017857e+06	6.842429e+06

Figure 11: The subtract operation in Hatchet enables comparing execution profiles. In this figure, the left graph is subtracted from the middle graph to obtain the right graph. When we sort the nodes in the right graph by time, we can easily identify the biggest offenders.

```
gf1 = GraphFrame.from_caliper('lulesh-27cores')
gf2 = GraphFrame.from_caliper('lulesh-512cores')

filtered_gf1
    = gf1.filter(lambda x: x['name'].startswith('MPI'))
filtered_gf2
    = gf2.filter(lambda x: x['name'].startswith('MPI'))

squashed_gf1 = filtered_gf1.squash()
squashed_gf2 = filtered_gf2.squash()

diff_gf = squashed_gf2 - squashed_gf1
```

node	node	time (inc)	name	nid	time
MPI_Allreduce	MPI_Allreduce	2.072371e+06	MPI_Allreduce	3	2.072371e+06
MPI_Finalize	MPI_Finalize	4.042198e+04	MPI_Finalize	0	4.042198e+04
MPI_Isend	MPI_Isend	1.753768e+04	MPI_Isend	15	1.753768e+04
MPI_Isend	MPI_Isend	7.718737e+03	MPI_Isend	13	7.718737e+03
MPI_Isend	MPI_Isend	7.542969e+03	MPI_Isend	7	7.542969e+03
MPI_Waitall	MPI_Waitall	4.573508e+03	MPI_Waitall	5	4.573508e+03
MPI_Barrier	MPI_Barrier	4.240952e+03	MPI_Barrier	12	4.240952e+03

Figure 12: Hatchet makes it easy to extract the calls in a particular library, MPI for example, using the filter operation, and then to compare the extracted sub-graphs using the subtract operation. In the example above, we can easily identify which specific `MPI_Send` calls take more time when we scale from 27 to 512 cores.

the results of row-wise subtraction of the two input DataFrames. Sorting the columns in the new DataFrame by decreasing time can quickly identify the most problematic nodes.

Multi-node scaling: In a similar scenario, a user might be interested in comparing two executions that use a different number of MPI processes. Let's say that the user is interested in finding the difference in times spent in different MPI routines by call site. We can do this also using the Hatchet API and a few lines of code.

Figure 12 shows the code for this analysis. We read in the two datasets of LULESH, and filter them both on the name column by matching the names against `^MPI`. After the filtering operation, we squash the DataFrames to generate GraphFrames that just contain the MPI calls from the original datasets. We can now subtract the squashed datasets to identify the biggest offenders. In the figure, we observe that as we scale from 27 to 512 cores, the largest time increase is in `MPI_Allreduce`. As we can see, the graph and DataFrame objects in Hatchet and the powerful pandas API can help in simplifying complex performance analysis tasks, which would have possibly taken many man-hours in another tool.

Finally, we demonstrate the use of Hatchet for comparing several datasets to study the weak scaling behavior of an application. We ran LULESH from 1 to 512 cores on third powers of some numbers (a requirement of the application). We read in all the datasets into Hatchet, and for each dataset, we use a few lines of code to filter the regions where the code spends most of the time. We then use the pandas' pivot and plot operations to generate a stacked bar chart that shows how the time spent in different regions of LULESH

```
datasets = glob.glob('lulesh*.json')
datasets.sort()

dataframes = []
for dataset in datasets:
    gf = GraphFrame.from_caliper(dataset)
    gf.drop_index_levels()

    num_pes = re.match('(.*)-(\d+)(.*)', dataset).group(2)
    gf.dataframe['pes'] = num_pes
    filtered_gf = gf.filter(lambda x: x['time'] > 1e6)
    dataframes.append(filtered_gf.dataframe)

result = pd.concat(dataframes)
pivot_df = result.pivot(index='pes', columns='name', values
    ='time')
pivot_df.loc[:,:].plot.bar(stacked=True, figsize=(10,7))
```

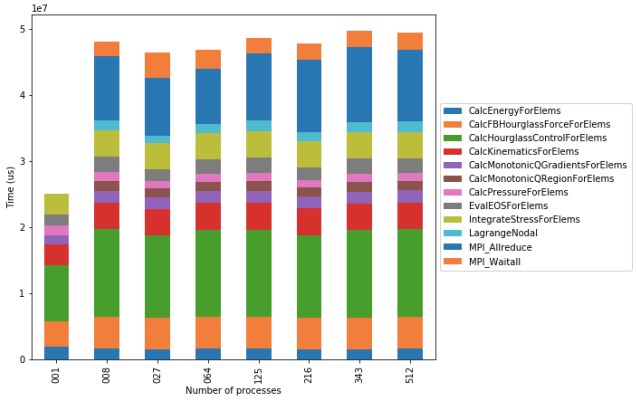


Figure 13: We read in eight LULESH caliper datasets and create a GraphFrame for each. We then filter the datasets to focus on the most time-consuming regions. For plotting, we concatenate all the DataFrames into one while storing a key that identifies the number of processes, and then use pivot to rearrange the data in a format more suitable for pandas' plot function. The resulting stacked bar chart is shown on the right.

changes as the code scales (Figure 13). This case study demonstrates the combined potential of Hatchet and pandas in making data analytics quick and convenient for the HPC user. We believe no other performance tool provides the functionality to generate such information without significant time and effort.

7 RELATED WORK

There are many profilers that can display call path or call graph profiles. Tools like Caliper [5], Open|SpeedShop [25], TAU [21], Score-P [12], and HPCToolkit [3] can all gather fine-grained execution profiles for post-mortem performance analysis. CallFlow [18], hpcviewer [17] in HPCToolkit, TAU, and flame graphs [9] are visualization tools specifically for CCTs. All of these tools have their own format for storing the collected data, and all but Caliper and flame graphs provide a custom GUI for viewing call path profiles. Some, like TAU, can import data from other tools, but none of them offer a *programmable* interface for dealing with raw, structured profile data from parallel runs. Users must point and click to analyze the data, which can be time consuming and inflexible for large datasets or custom analyses.

Many performance tools provide facilities to store performance data in a database and to apply machine learning and other data analysis tools to it. PerfExplorer [11] provides a database, a GUI analysis environment, and the PerfDMF [10] data format. Open|SpeedShop has an internal SQL database used by the GUI to load parts of performance datasets. However, all of these tools predate the popularization of data analysis frameworks like R [19] and pandas [15, 16], and they do not provide rich APIs for manipulating data. TauDB, part of PerfDMF, provides language bindings for exploring datasets, but it does not provide the in-memory query or aggregation capabilities that modern frameworks have. All "queries" in these tools must be written in SQL, with a fixed schema, and handed off directly to the backend database. There is no in-memory DataFrame or abstraction layer as we have leveraged in Hatchet. The closest related work to Hatchet is likely *differential profiling*. Early work [14, 20] showed the utility of subtracting similar or scaled call trees to pinpoint performance issues. This work was improved upon by techniques for scaling analysis implemented in HPCToolkit [23, 24]. HPCToolkit provides facilities for calculating derived expressions from performance metrics on call trees *within* the GUI, and this can be used to scale and subtract columns in the hpcviewer GUI. However, the usage model is cell-based like a spreadsheet; it is not fully programmable or easily integrated with other frameworks.

Likely the most scalable existing call path *visualizer* is HPCTraceViewer [22], which provides visualizations of call paths over time, MPI ranks, and threads in parallel codes. This tool and Libra [6] are the closest analogs to the per-MPI-rank analyses in this paper. Again, though, these are GUI tools and they do not provide the flexibility to easily script new analyses or to easily query, filter, aggregate, and squash profile data in an indexed DataFrame as Hatchet does. Typically, the available analyses are manually selected through drop down menus or some other user-interface, and there is limited flexibility for customization.

With Hatchet, we provide a common data model for representing structured profiles from today's HPC tools. We provide a means to index a DataFrame by structured attributes, such as nodes in a call tree or call graph, and Hatchet builds on the widely used pandas data analysis framework, and all of the plotting and analysis libraries that can be used with it. Hatchet is not a closed-universe tool; it provides a *canonical* representation of profile data and can read data from many existing tools. If Hatchet users need to analyze data from a new measurement tool, they can do so without modifying their analysis scripts, and without learning a new format, new API, or new GUI. We advocate the use of existing *measurement* tools with Hatchet for analysis, in order to achieve more automated, reproducible results.

8 CONCLUSION

Analyzing performance and connecting performance degradation to parts of the code is important to guide application developers in their performance optimization efforts. Large parallel applications with tens to thousands of lines of codes are difficult to analyze.

Additionally, performance profiles of such applications can have hundreds of thousands of call sites or nodes in a dynamic execution profile. Most existing tools fall short in allowing users to programmatically analyze performance data.

In this paper, we presented Hatchet, a Python-based library leveraging the powerful API of data analysis tools, such as pandas to analyze structured profiling data. Since pandas does not support structured data indexed by nodes in a graph, Hatchet provides a hierarchical index to support indexing DataFrame rows by nodes in the graph. Hatchet provides a canonical data model that enables representing and analyzing different types of performance data.

Leveraging many DataFrame operations and adding its own, Hatchet simplifies many common performance analysis tasks on structured profiling data. Using case studies, we demonstrated that Hatchet provides an easy way to perform many complex tasks on parallel profiles by writing a few lines of code. These tasks include, 1) identifying regions or call sites with the most load imbalance across MPI processes or threads, 2) filtering datasets by a metric or library/function names to focus on subsets of data; and 3) easily handling and analyzing multi-rank, profile data from multiple executions. We expect that Hatchet will make HPC performance analysis quicker, easier, and more effective.

In the future, we plan to add a query language for the Hatchet user to specify expressions for filtering a graph. The user should be able to specify for example, select all the nodes whose load module is X and a descendant is an MPI routine. We believe that with a language to specify different ways to dissect and prune graphs and trees, Hatchet could become even more powerful in the terms of the kinds of analyses it can support.

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-772402).

REFERENCES

[1] [n.d.]. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254.

[2] [n.d.]. Kripke. https://codesign.llnl.gov/kripke.php.

[3] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[4] D. Böehme, D. Beckingsale, and M. Schulz. 2017. Flexible Data Aggregation for Performance Profiling. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 419–428. https://doi.org/10.1109/CLUSTER.2017.34

[5] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: Performance Introspection for HPC Software Stacks. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '16)*. IEEE Computer Society, Article 47, 11 pages. http://dl.acm.org/citation.cfm?id=3014904.3014967 LLNL-CONF-699263.

[6] Todd Gamblin, Bronis R. de Supinski, Martin Schulz, Robert J. Fowler, and Daniel A. Reed. 2008. Scalable Load-Balance Measurement for SPMD Codes. In *Supercomputing 2008 (SC'08)*. Austin, Texas. http://www.cs.unc.edu/~tgamblin/pubs/wavelet-sc08.pdf LLNL-CONF-406045.

[7] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. 1982. Gprof: A call graph execution profiler. *SIGPLAN Not.* 17, 6 (1982), 120–126.

[8] Brendan Gregg. [n.d.]. Flame Graphs. https://github.com/brendangregg/FlameGraph.

[9] Brendan Gregg. 2015. Flame graphs. Online. http://www.brendangregg.com/Slides/FreeBSD2014_FlameGraphs.pdf.

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

[11] Kevin A. Huck and Allen D. Malony. 2005. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In *Supercomputing 2005 (SC'05)*. Seattle, WA, 41.

[12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. 2012. Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In *Tools for High Performance Computing 2011*, Holger Brunst, Matthias S. Müller, Wolfgang E. Nagel, and Michael M. Resch (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 79–91.

[13] AJ Kunen, TS Bailey, and PN Brown. 2015. KRIPKE-A massively parallel transport mini-app. *Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep* (2015).

[14] P. E. McKenney. 1995. Differential profiling. In *MASCOTS '95. Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. 237–241. https://doi.org/10.1109/MASCOT.1995.378681

[15] Wes McKinney. 2010. Data Structures for Statistical Computing in Python. In *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman (Eds.). 51 – 56.

[16] Wes McKinney. 2017. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly Media. https://www.amazon.com/Python-Data-Analysis-Wrangling-IPython/dp/1491957662?SubscriptionId=AKIAIOBINVZYXZQZ2U3A&tag=chimbori05-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1491957662

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

[18] H. T. Nguyen, L. Wei, A. Bhatele, T. Gamblin, D. Boehme, M. Schulz, K. Ma, and P. Bremer. 2016. VIPACT: A Visualization Interface for Analyzing Calling Context Trees. In *2016 Third Workshop on Visual Performance Analysis (VPA)*. 25–28. https://doi.org/10.1109/VPA.2016.009

[19] R Core Team. 2013. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. http://www.R-project.org/

[20] Martin Schulz and Bronis R. de Supinski. 2007. Practical Differential Profiling. In *Euro-Par 2007 Parallel Processing*, Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 97–106.

[21] S. Shende and A. D. Malony. 2006. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications* 20, 2 (2006), 287–311. https://doi.org/10.1177/1094342006064482

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

[23] Nathan R. Tallent, Laksono Adhianto, and John M. Mellor-Crummey. 2010. Scalable Identification of Load Imbalance in Parallel Executions Using Call Path Profiles.

[24] Nathan R. Tallent, John M. Mellor-Crummey, Laksono Adhianto, Michael W. Fagan, and Mark Krentel. 2011. Diagnosing performance bottlenecks in emerging petascale applications.

[25] The Open|SpeedShop Team. [n.d.]. Open|SpeedShop for Linux. http://www.openspeedshop.org

[26] David Wheeler. 2012. SLOCCount. http://www.dwheeler.com/sloccount