Hatchet: Pruning the Overgrowth in Parallel Profiles

Abhinav Bhatele*, Stephanie Brink†, and Todd Gamblin*

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

Time spent in user-annotated nested regions in LULESH

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

ABSTRACT
Performance analysis of parallel codes is critical to ensure high efficiency and scalability to large process counts. A large number of profiling tools exist to instrument codes and gather performance data. However, analytics and visualization tools to analyze such data that are effective, 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 the code. We present a set of techniques and operations leveraging modern data analysis libraries for parallel profile analysis. 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 how common performance analyses can be performed in a reproducible manner with only a few lines of Hatchet code.

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 some understanding of the structure of the software. The 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, e.g., the profiler would differentiate time spent in MPI_Send when it is called by a hydrodynamics routine from time spent in MPI_Send when it is called from 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 represents a code region annotated by the user (and is illustrated as a box in the flame graph).

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 via 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 include a way to view only one or two call graphs at a time. They are often insufficient for tasks like load balance analysis that require detailed analysis and clustering across ranks, threads, and time. They also lack general capabilities for effectively sub-selecting and focusing on specific parts of a larger dataset.

Growing interest in data science has led to the existence of many widely available and fully-featured data analysis environments. Python and R in particular support dataframes that blend features of traditional numerical computing environments like MATLAB with database-derived features like indexed queries, joins, and aggregations. Numerous plotting libraries and visualization tools are
available to view data in these environments, and all of it is available through 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].

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;
- a set of operators that allow structured data to be filtered, aggregated, and pruned to produce meaningful, API-centric sub-profiles; and
- an implementation of these techniques in the Hatchet library that enables users to leverage modern data analysis approaches for performance analysis of large-scale call path profiling data.

The remainder of this paper outlines Hatchet’s architecture, as well as how it can be used on real datasets. Using performance datasets derived from running HPC codes, we present several case studies to demonstrate how common performance analyses can be performed in a reproducible manner 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 HPC performance analysis quicker, easier, and reproducible.

2 BACKGROUND

We define the different kinds of structured profiles output by various profiling tools and specifically describe the HPCToolkit and Caliper outputs in detail.

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. A calling context tree (CCT) is a prefix tree of calling contexts. Each unique invocation of a function (by call path) becomes a node in the CCT, and the path from a given node to the root of the tree represents a distinct calling context. 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), but averages 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, HPCToolkit and Caliper, and their output formats.

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 application 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
2.3 Pandas and Dataframes

Pandas: Pandas is an open-source Python library providing data structures and tools for data analysis [15]. 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. 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, multiply 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 the 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 code region, 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 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 has called 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 FrameIDs.](image-url)

3.1.1 Generic FrameIDs. Each node in Hatchet’s graph contains a generic identifier for the code construct it represents. We call this identifier a FrameID (after a call frame). FrameIDs 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. FrameIDs support basic comparison operations, such as ==, >, <, etc., which are evaluated based on the names and values of component fields. If two FrameIDs 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 FrameIDs; 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. FrameIDs 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 FrameIDs, 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 that column an index. By default, we use the Python id() function for the node key. This is equivalent, roughly, to C & 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 constructed 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.

Figure 3: In Hatchet, the graphframe consists of a graph and a dataframe object.

Because of the way we have architected the Hatchet structured index Graph, we can insert Node objects directly into the pandas data frame. The nodes are indexed and 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 FrameID. For example, in the figure, we have added the name and nid columns from the FrameID 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, 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 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 frame 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 FrameID 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, that GraphFrame models the edge relationships between nodes in the structured data, and a dataframe storing the numerical (performance metrics such as time, PAPI counters 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 split JSON files generated by Caliper. In addition, one can provide structured data in Graphviz’ DOT format or a simple string literal.

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 files. 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 important operators provided by the Hatchet API allowing 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 return a new graphframe.

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.

```python
gf = GraphFrame( ... )
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.

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 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 some metric become inaccurate. This function performs a post-order traversal of the graph to update all inclusive metrics stored 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, squash 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 alive ancestor is connected by an edge to the nearest alive child on the path. All call paths in the graph are re-wired in this manner. After a squash operation, the graph and dataframe become consistent again. 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.

```python
filtered_gf = gf.filter(lambda x: x['time'] > 10.0)
squashed_gf = filtered_gf.squash()
```

Figure 5: The graph before (left) and after (right) a squash operation on the graphframe.
check_equivalence: This checks whether two graphs are exactly equivalent or not in their structures by comparing the call paths of the respective children. If they are equivalent, it returns true, otherwise it returns false.

union: The union function takes two graphs and creates a unified graph, preserving as much structure of the original graphs as possible. Typically, a union is called if the structures of two graphs are different. This operation is useful when we wish to perform an add or subtract operation on two graphframes and the graphs are not structurally equivalent. In this case, a union is performed first before the add or subtract operation.

4.3 Graphframe Operations

copy: The copy operation returns a copy of a graphframe by creating a copy of the dataframe object and the graph object, which in turn involves cloning all the nodes in the graph. Creating a copy enables the user to modify a copy of the graphframe object, while keeping the original object unchanged. This is useful for example, in the case of add and subtract, when there are two operands and the result needs to return a new graphframe.

add: Assuming the graphs in two graphframes are identical (i.e., check_equivalence returns true), the add operation computes the sum of two dataframes column-wise. In the case where the two graphs are not identical, union (described above) is applied first to create a unified graph before performing the sum. The add operation returns a new resulting graphframe or modifies one of the graphframes in place in the case of the following addition assignment: \( a + b \).

subtract: The subtract operation is similar to the add operation in that it requires the two graphs to be identical. Once the graphs are structurally equivalent, the subtract operation computes the difference between the two dataframes column-wise. The subtract operation returns a new resulting graphframe or modifies one of the graphframes in place in the case of the subtraction assignment \( a - b \). Figure 6 shows the subtraction of one graphframe from another and the graph for the resulting graphframe.

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 9, we provide execution times for some operations in Hatchet when using increasingly large datasets. We ran LULESH
gf = GraphFrame()
gf.from_hpctoolkit('kripke')
grouped = gf.dataframe.groupby('name').sum()

Figure 8: 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).

Hatchet only adds a modest amount of code on top of the pandas library. Currently, the Hatchet code is nearly 900 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 8 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 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 11 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 dataframe in the figure shows all the nodes in LULESH with an imbalance greater than 2.0.

### 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 10 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 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 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], OpenSpeedShop [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
gf1 = GraphFrame()
gf1.from_caliper('lulesh-1core.json')
gf2 = GraphFrame()
gf2.from_caliper('lulesh-27cores.json')
gf2.drop_index_levels()
gf3 = gf2 - gf1

Figure 10: 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()
gf1.from_caliper('lulesh-512cores')
gf2 = gf1.copy()
gf2.drop_index_levels(function=np.mean)
gf1.drop_index_levels(function=np.max)
gf1.dataframe['imbalance'] = gf2.dataframe['time'].div(gf1.dataframe['time'])

Figure 11: 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.

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 [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. After, 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.
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.

```python
gf1 = GraphFrame()
gf1.from_caliper('lulesh-27cores')
gf2 = GraphFrame()
gf2.from_caliper('lulesh-512cores')
filtered_gf1 = gf1.filter(lambda x: x['name'].startswith('MPI'))
squashed_gf1 = filtered_gf1.squash()
squashed_gf2 = filtered_gf2.squash()
diff_gf = squashed_gf2 - squashed_gf1
```

Figure 13: We read in eight LULESH caliper datasets in a for loop 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.

```python
datasets = glob.glob('lulesh*.json')
dataframes = []
for dataset in datasets:
    gf = GraphFrame()
    gf.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))
```

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 programatically 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 Hatchet will make HPC performance analysis quicker, easier, and more effective. Data analysis tools have more features, and are better maintained by open-source software communities than tool GUIs.

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