Interactive Data Analysis of Multi-Run Performance Data

Vanessa Lama
University of Tennessee, Knoxville, vlama@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Vanessa Lama entitled "Interactive Data Analysis of Multi-Run Performance Data." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Michela Taufer, Major Professor

We have read this thesis and recommend its acceptance:

Jakob Luettgau, Silvina Caino-Lores, Michael W. Berry, Olga Pearce, Stephanie Brink

Accepted for the Council:
Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Interactive Data Analysis of Multi-Run Performance Data

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Vanessa Lama

May 2023

To my family, friends, mentors, peers, and supervisors

Acknowledgments

I would like to express my sincere gratitude to the following people and institutions for their support, guidance, and encouragement throughout my Master's thesis:

First and foremost, I would like to express my deepest appreciation to my thesis advisor, Dr. Michela Taufer, for her invaluable guidance, encouragement, and patience throughout the research process. Without her support, this thesis would not have been possible.

I am also deeply grateful to the members of my thesis committee, Dr. Michael Berry, Dr. Stephanie Brink, Dr. Silvina Caino-Lores, Dr. Jakob Luettgau, and Dr. Olga Pearce for investing their time and providing insightful feedback.

I would like to thank Treece Burgees, Ian Lumsden, Michael McKinsey, and Connor Scully-Allison for their help and support during the research and development process.

I would like to acknowledge the valuable resources provided by each member of the Global Computing Lab at the University of Tennessee.

I am also grateful to my family and friends for their unwavering support, encouragement, and understanding throughout my Master's program.

Thank you all for making this journey a truly fulfilling and rewarding experience.

Abstract

Multi-dimensional performance data analysis presents challenges for programmers, and users. Developers have to choose library and compiler options for each platform, analyze raw performance data, and keep up with new technologies. Users run codes on different platforms, validate results with collaborators, and analyze performance data as applications scale up. Site operators use multiple profiling tools to optimize performance, requiring the analysis of multiple sources and data types. There is currently no comprehensive tool to support the structured analysis of unstructured data, when, holistic performance data analysis can offer actionable insights and improve performance. In this work, we present thicket, a tool designed based on the experiences and insights of programmers, and users to address these needs. Thicket is a Python-based data analysis toolkit that aims to make performance data exploration more accessible and user-friendly for application code developers, users, and site operators. It achieves this by providing a comprehensive interface that allows for the easy manipulation, modeling, and visualization of data collected from multiple tools and executions. The central element of Thicket is the "thicket object," which unifies data from multiple sources and allows for various data manipulation and modeling operations, including filtering, grouping, and querying, and statistical operations. Thicket also supports the use of external libraries such as scikit-learn and Extra-P for data modeling and visualization in an intuitive call tree context. Overall, Thicket aims to help users make better decisions about their application's performance by providing actionable insights from complex and multi-dimensional performance data. Here, we present some capabilities extended by the components of thicket and important use cases that have implications beyond the data structure that provide these capabilities.

Table of Contents

1 Thesis Overview **1**

1.1 Thesis Statement 2

1.2 Organization 2

2 Background:
Overview on Performance Tools **4**

2.1 Overview of Performance Tools 4

2.1.1 HPCToolkit 4

2.1.2 Score-P 5

2.1.3 TAU 6

2.1.4 Allinea MAP 6

2.1.5 Caliper 7

2.2 Thicket 7

3 My Contributions:
The Metadata and Stats Capabilities **11**

3.1 Metadata 11

3.2 Aggregate Statistics 13

3.3 Filtering with Metadata 13

3.4 Grouping with Metadata 15

3.5 Filtering with Aggregate Statistics 18

4 Use Cases:
Impact of Metadata on Analysis 20
4.1 Use Cases: Settings 20
4.2 Use Case 1: Propagation of metadata filter onto performance data 21
4.3 Use Case 2: Refinement of aggregate statistics heat map using stats filter . . 21

5 Conclusion and Future Work 31
5.1 Summary 31
5.2 Future Work 32

Bibliography 33

Vita 35

List of Figures

2.1 A diagram depicting the relationship between different components of thicket and the structure of these components. 9

3.1 An example of a metadata table with several application runs. 12
3.2 An empty aggregate statistics table with multiple nodes stored in the table. . 14
3.3 Resulting metadata table after filtering Fig. 3.1 to select rows corresponding to the clang compiler. 16
3.4 Resulting metadata table after grouping metadata table from Fig. 3.1 with respect to the unique values present in the compiler column. 17
3.5 A filtered empty statistics table with significantly less no. of nodes compared to the original statistics table from Fig. 3.2. 19

4.1 An example of filtering metadata to only contain profiles derived from `clang` compiler. 22
4.2 Propagation of metadata filter onto performance data. 23
4.3 An aggregate statistics table with the inclusive time metrics mean, median, and percentile values. 25
4.4 The aggregate statistics table from Fig. 4.3 sorted on the basis of the average (median) exclusive run-time, or the third column from the referenced table. . 26
4.5 The heat map generated with respect to the time(exc) median metric from Fig. 4.4 27
4.6 The stats filter applied to Fig. 4.4, reducing the number of nodes to six. . . . 28
4.7 The heat map generated with respect to the time(exc) median metric from Fig. 4.6 30

Chapter 1

Thesis Overview

Optimizing performance is becoming increasingly difficult as the complexity of High Performance Computing (HPC) simulations, software stacks, and heterogeneous architectures grows. The intricacies involved often lead to missed opportunities for significant performance improvements. These missed opportunities can have a considerable impact, particularly in large-scale applications where even minor performance improvements can lead to substantial gains in efficiency and cost-effectiveness.

To address this, new tools are required to uncover these hidden opportunities and provide actionable insights from the massive amounts of multi-dimensional, multi-scale, multi-architecture, and multi-tool performance data generated by modern applications. The development of such tools is essential for maximizing performance.

Simply having access to performance data is not enough. It is also essential to have the technologies to identify performance opportunities and provide meaningful insights from the data. This requires the development of critical technologies that can enable rapid exploration of performance data, as well as the ability to identify trends an d patterns across multiple data sources.

Seeing the need for such a tool, we present a python based performance analysis tool that can store multi-run performance data and allow Exploratory Data Analysis (EDA) on that data. Our solution tool comes with a multitude of visualizationand analysis of peformance data. We present a few examples of such capabilities extended by the tool in this thesis. We present two helpful use cases to determine optimal performance setting for scientists to run their programs. Some of this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory (LLNL) under Contract DE-AC52-07NA27344 (LLNL-TH-848357), specifically LLNL's Performance Analysis and Visualization at Exascale (PAVE) team.

1.1 Thesis Statement

In this thesis, we claim that the increasing complexity of HPC simulations, software environments, and diverse computing infrastructures presents challenges in optimizing performance as applications with multiple parameters, complex software environments, and running on various hardware configurations require multiple tools to collect and analyze performance data. Exploring this data meaningfully is a significant bottleneck in identifying actionable insights related to application performance issues.

Our contributions to the problem are as follows:

- We provide an overview of some popular performance analysis tools and introduce a new tool, thicket, that support ensemble analysis of performance data.
- We design and integrate three new capabilities in the thicket tool: filtering metadata, grouping metadata, and filtering stats.
- We demonstrate the use of these capabilities, their impact on the performance data, and their flexibility in being integrated with other capabilities of thicket.

1.2 Organization

The upcoming layout of the thesis is as follows. Chapter 2 provides a brief overview of four performance analysis tools. It includes a brief introduction and background of thicket and all it's components. In chapter 3, we present the structures of thicket components in detail and provide a thorough example of the filtering and grouping capabilities of thicket. Chapter 4 consists of two use cases that expand on the capabilities from chapter 3 beyond the data structures to the filtering and grouping are applied. Finally, chapter 5 provides an overall conclusion on the thesis topic and possible future work for thicket.

Chapter 2

Background: Overview on Performance Tools

This chapter introduces a short overview of analysis tools for performance data. As part of this thesis, we present the tool we have expanded with new capabilities.

2.1 Overview of Performance Tools

A wide range of performance analysis tools are available, developed by independent and for-profit entities, that utilize various measurement methodologies and are suitable for different use cases. Comprehensive performance analysis tools such as HPCToolkit [1, 8], Score-P [7], TAU [9], Allinea MAP [6], and Caliper [3] collect detailed performance data of program executions for thorough scrutiny.

2.1.1 HPCToolkit

HPCToolkit [1, 8] is a set of multi-platform tools designed for performance analysis of optimized parallel programs. These tools use profile-based performance analysis techniques to collect performance data during program execution and present it in a meaningful and actionable way for developers. The tools are designed to work with various high-performance computing architectures, including clusters, supercomputers, and GPUs.

The HPCToolkit [1, 8] suite includes several tools for different stages of performance analysis, including instrumentation, measurement, and analysis. These tools use static and dynamic analysis techniques to collect performance data, including call-path profiling, source-code annotation, and hardware performance counters. The collected data is presented in various visualizations and reports, allowing developers to identify performance bottlenecks and optimize code for improved performance.

The tool provides a comprehensive and flexible performance analysis of optimized parallel programs, enabling developers to improve the performance of their applications and achieve better scalability on HPC architectures.

2.1.2 Score-P

Score-P [7] is a joint performance measurement runtime infrastructure designed to support multiple performance analysis tools. Score-P uses a modular and extensible architecture to provide flexible and scalable performance measurement capabilities for various HPC architectures.

Score-P [7] provides a unified interface for instrumenting and measuring performance data during program execution, including call-path profiling, trace-based analysis, and hardware performance counter measurements. The collected data is presented in various visualizations and reports, allowing developers to identify performance bottlenecks and optimize code for improved performance. This tool also includes online and offline analysis support, enabling developers to perform performance analysis in real time or after program execution. The tool supports various input and output formats, allowing users to integrate easily with other performance analysis tools and workflows.

Score-P [7] provides a powerful and flexible platform for performance analysis of HPC applications, enabling developers to optimize their code for improved performance and scalability.

2.1.3 TAU

The TAU [9] Parallel Performance System is a comprehensive tool for profiling and tracing parallel applications. It provides a variety of performance analysis capabilities, including function-level profiling, call-graph visualization, and trace-based analysis.

TAU [9] supports various programming languages, including C++, Fortran, and Java, and can be used to analyze applications running on different parallel architectures, including clusters, grids, and supercomputers. The tool provides several runtime options for customizing data collection and analysis, allowing users to focus on specific regions of interest and optimize their code for improved performance.

It is a powerful and flexible tool for analyzing the performance of parallel applications, allowing developers to identify bottlenecks and optimize their code for improved performance and scalability. This tool also includes a variety of visualization and reporting tools for analyzing performance data, including graphical call-graph views, statistical summaries, and source-code annotations. Additionally, TAU [9] supports several output formats for integrating with other performance analysis tools and workflows.

2.1.4 Allinea MAP

Allinea MAP [6] is a software tool for analyzing the performance of parallel applications running on HPC systems. It provides comprehensive profiling and analysis features, including call-graph visualization, OpenMP profiling, and energy consumption analysis.

One of the tool's key features is its low overhead, which allows developers to collect detailed performance data without significantly impacting the runtime performance of their applications. This is achieved through various techniques, such as sampling-based profiling and lightweight instrumentation.

Allinea MAP [6] also provides visualization and reporting tools for analyzing performance data, including graphical call-graph views, statistical summaries, and source-code annotations. The tool supports various output formats for integrating with other performance analysis workflows and tools. It is a powerful and user-friendly tool for analyzing parallel application performance, helping developers identify performance bottlenecks and optimize their code for improved efficiency and scalability. Its ability to analyze performance and energy consumption makes it a valuable tool for optimizing the performance of HPC systems.

2.1.5 Caliper

Caliper [3] is a performance measurement and analysis framework that provides a consistent way to collect, store, and analyze performance data for high-performance computing (HPC) applications. It enables users to measure and analyze the performance of their HPC applications, identify bottlenecks and performance issues, and optimize the performance of their applications. Caliper supports a variety of HPC platforms, including CPUs, GPUs, and accelerators, and is designed to be easily integrated into existing applications and workflows. Its primary goal is to provide insights into the performance characteristics of HPC applications, which can then be used to improve the efficiency and scalability of these applications.

Caliper's analysis tools include call-path profiling, which allows users to see where their application is spending its time, and identify bottlenecks and performance issues. It also provides performance regression analysis, which can be used to compare the performance of different runs of the same application, and identify performance trends and anomalies. In this thesis work, we use profiles generated using caliper for examples and use cases.

2.2 Thicket

All the tools introduced above require the input to be a single profile and do not offer specific capabilities for analyzing an ensemble of runs. A team of HPC experts at LLNL have been targeting this limit by developing a new tool that generates a collection of multi-run data for EDA. The tool is called thicket [5], and it differs from the above-described tools by allowing users to store multi-run performance data in a single thicket object for further analysis. In this thesis, we extend thicket and augment it with new capabilities.

Thicket [5] is a performance tool designed to help users study multi-dimensional performance data after completing traditional performance analysis operations. It instantiates a collection of performance profiles into a thicket object composed of performance data, metadata, and aggregated statistics. The thicket tool allows built-in functionality to aggregate, visualize, and export performance data from thicket objects.

A thicket object's architecture is designed to link the different performance dimensions through primary and foreign keys, providing a flexible data model. This architecture enables the structured analysis of unstructured performance data. Fig. 2.1, borrowed from thicket [5], presents the key factors that belong to a thicket object: a call tree, performance metrics, multi-performance metrics, metadata, and aggregate statistics. Specifically, Fig. 2.1 shows how the nodes of the call tree defines the structure of the multi-index performance data table as one of the table index. Thicket uses a unique profile hash to identify each unique profile read by the thicket tool. This profile is the other index in the multi-index performance data table, and there can be multiple profile hashes for each node. For example, if the node is a function that gets called several times during program execution, each row in the table associated with that function will represent the performance metrics for one of those executions. Therefore, each row in the performance data table corresponds to a unique call tree node and profile index combination.

Profiles include metadata such as the program's launch date, user, machine name, etc. corresponding to each profile. The metadata table, a thicket object component, has the same profile hashes as its performance data table. In the metadata table, the profile hash is the only index.

Additional statistical computations can be conducted on the performance metrics of the performance data table. A thicket object reserves a separate table to store each node's computational or statistical results. This component is called the aggregate statistics table and by default, only contains the nodes from performance data as an index and an additional column with the names of the nodes. In this thesis work, we use aggregate statistics table and statistics table interchangeably.

This organization of the three main components of a thicket object replicates the relational database model with the performance data table as the main table. By organizing the performance data in this way, we can easily compare the performance of different program executions and analyze the results.

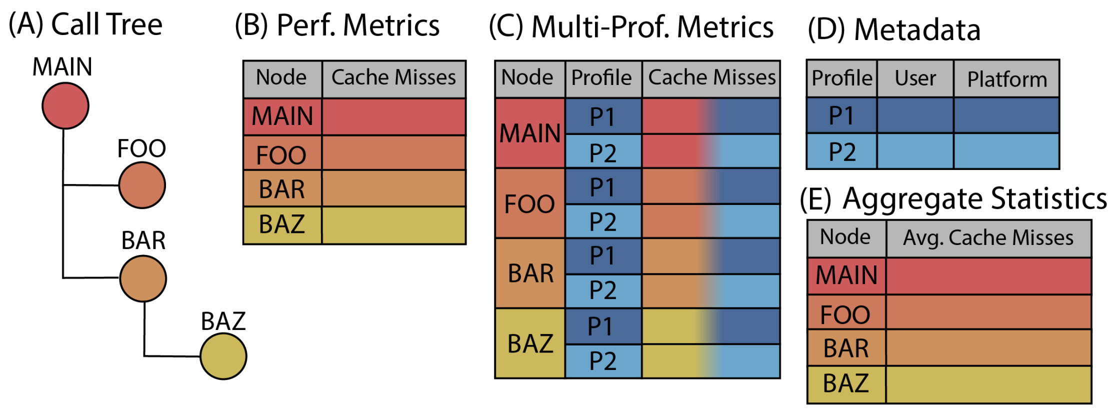


Figure 2.1: A diagram depicting the relationship between different components of thicket and the structure of these components.

All the table components of thicket are implemented by leveraging pandas DataFrame API and the call tree component is an extension of Hatchet's call tree structure [2, 4].

In the next chapter, we provide details of our contributions for the development of thicket and how these contributions support different capabilities in thicket.

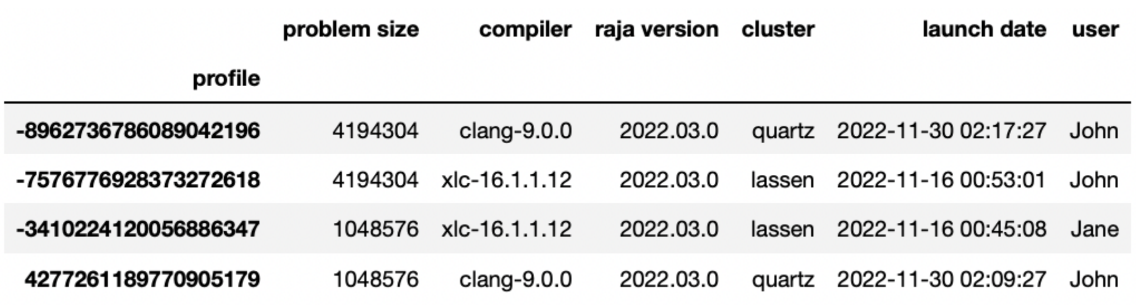
Chapter 3

My Contributions: The Metadata and Stats Capabilities

In this chapter, we discuss our two contributions. First, we describe our design of data structures for performance data (i.e., metadata and statistics). Second, we discuss the implementation of our three new ticket capabilities for grouping metadata, filtering metadata, and filtering statistics.

3.1 Metadata

We designed the metadata table to store information about how the application was made and run. Each row in this table represents one application run, and each row is given a unique number called a "profile hash" that helps keep track of it. This table helps us understand important details about how the application was built and used. We structured the metadata table to store a single record (or profile) that can be associated with several rows in the performance data table. This is the case as the metadata is saved for each profile run, meaning that only one set of metadata is connected to a single profile run. Therefore, any node in the performance data table corresponding to the same profile run will have the same metadata. This helps to keep track of the various executions of the application and their associated metadata. Fig. 3.1 represents information corresponding to four distinct profiles.



profile	problem size	compiler	raja version	cluster	launch date	user
-8962736786089042196	4194304	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:17:27	John
-7576776928373272618	4194304	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:53:01	John
-3410224120056886347	1048576	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:45:08	Jane
4277261189770905179	1048576	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:09:27	John

Figure 3.1: An example of a metadata table with several application runs.

The information in this example includes the compiler linked to each of the application runs, the date and time that the application was initiated, and the user who requested the execution of the application. We also designed the aggregate statistics component of thicket, which stores the nodes present in the performance data, as we briefly explained in Fig. 2.1.

3.2 Aggregate Statistics

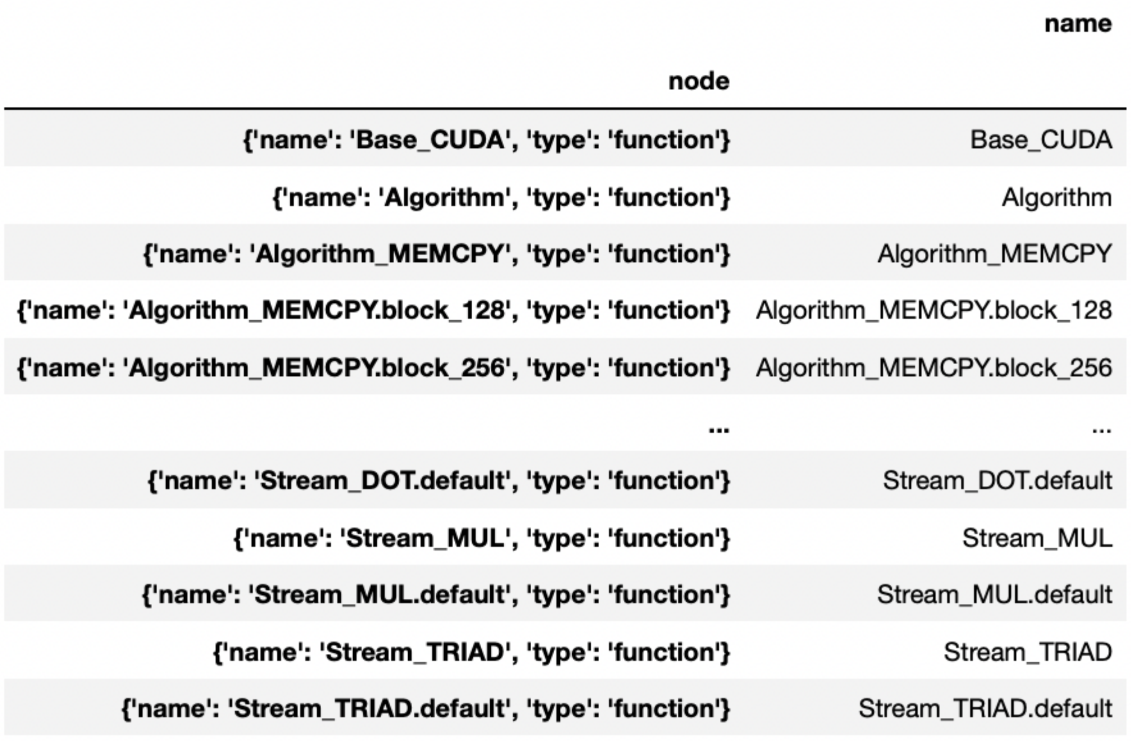
We store the aggregate statistics data as a table and a corresponding call tree. After performing statistical calculations, we store the computational and statistical results in a table for each row of a node index carried over from the performace data. Fig. 3.2 shows an empty statistics table that contains multiple rows of node index along with a column containing the name of each node. We provide this as the default form of the statistics table when a new thicket object is instantiated.

We present our two new capabilities extended by thicket's metadata table, that provides filtering and grouping benefits. We focus on this chapter's metadata and statistics table structures to showcase these capabilities.

3.3 Filtering with Metadata

The first capability is the filtering of profiles based on metadata. We enable filtering specific profiles (one or multiple rows in the metadata table) by applying a callable function. This is implemented by leveraging pandas built-in .apply() attribute to use a function along a column of the pandas DataFrame. The user specifies the callable function to apply on the table. This ensures that the profiles in the metadata table should match profiles in the performance data table and that no inconsistencies are present.

We first make a copy of the provided thicket. Then, the filter function is applied to the thicket's metadata table. The filtering of profiles is reflected on the thicket copy's performance data table (where the performance data is collected). This process resets the aggregate statistics table to its default state, just an empty stats table containing the index column with the nodes and a name column with the names of corresponding nodes.



node	name
{'name': 'Base_CUDA', 'type': 'function'}	Base_CUDA
{'name': 'Algorithm', 'type': 'function'}	Algorithm
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	Algorithm_MEMCPY
{'name': 'Algorithm_MEMCPY.block_128', 'type': 'function'}	Algorithm_MEMCPY.block_128
{'name': 'Algorithm_MEMCPY.block_256', 'type': 'function'}	Algorithm_MEMCPY.block_256
...	...
{'name': 'Stream_DOT.default', 'type': 'function'}	Stream_DOT.default
{'name': 'Stream_MUL', 'type': 'function'}	Stream_MUL
{'name': 'Stream_MUL.default', 'type': 'function'}	Stream_MUL.default
{'name': 'Stream_TRIAD', 'type': 'function'}	Stream_TRIAD
{'name': 'Stream_TRIAD.default', 'type': 'function'}	Stream_TRIAD.default

Figure 3.2: An empty aggregate statistics table with multiple nodes stored in the table.

An example of this is shown in the next chapter. Then, we return the thicket copy with all these changes. An example of metadata filtering is seen in Fig. 3.3. The original metadata table is from Fig. 3.1, and in this example, we filter the metadata table concerning the compiler column, wherein we select all the profiles in Fig. 3.1 that correspond to the clang compiler.

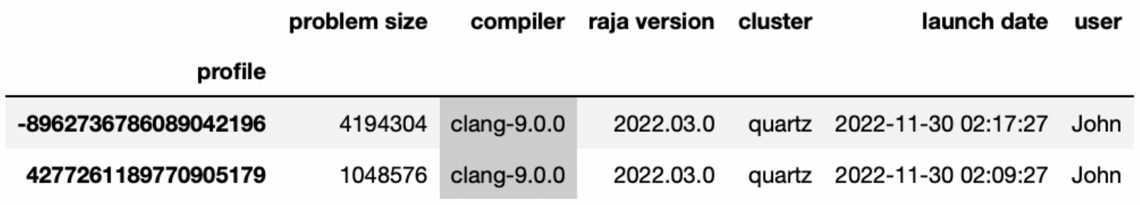
This capability can be applied to any one of the columns of the metadata table for a specified value in the column. Next, we present the grouping capability of thicket's metadata table.

3.4 Grouping with Metadata

The second capability is the grouping of profiles based on unique column values. We enable the grouping of profiles based on the unique values present in a column or a unique combination of values in multiple columns of the metadata table. To this end, we bring the intuition of Pandas *groupby()* semantics to a thicket. The user specifies the column(s) by which the grouping is to occur. Several thickets are formed, specific to the user's grouping instructions. Similar to metadata filtering, the changes made to the metadata table are propagated to the performance data and aggregate statistics table. As mentioned above, all the thickets formed with this grouping function have the aggregate statistics reset to their default state.

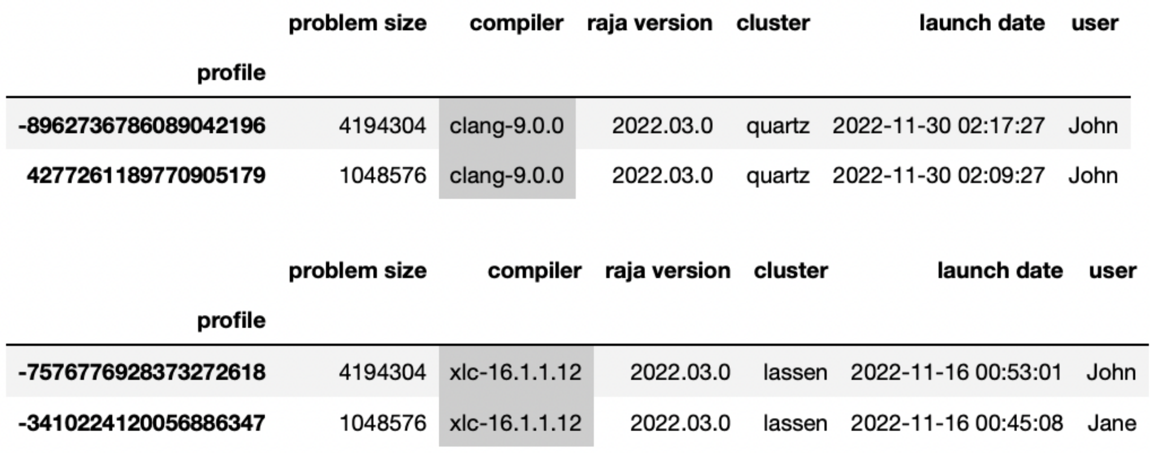
Similar to filtering, a copy of the provided thicket is created. And the grouping function is applied to the thicket's metadata table for specified column(s). For each subset of the metadata table created, the grouping of profiles is reflected on the resulting thickets' performance data table. Profiles in the metadata table of each resulting thicket should match profiles in the performance data of the same thicket. As a result of this process, a list containing the thickets is returned to the user.

Fig.3.4 shows an example of grouping with the metadata table with a similar approach in Fig. 3.3. Here, we present a case where the group is conducted, again, concerning the compiler column of the metadata table. The function looks for the compiler column, finds all the unique values in that column, and creates a new thicket for each value.



profile	problem size	compiler	raja version	cluster	launch date	user
-8962736786089042196	4194304	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:17:27	John
4277261189770905179	1048576	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:09:27	John

Figure 3.3: Resulting metadata table after filtering Fig. 3.1 to select rows corresponding to the clang compiler.



	problem size	compiler	raja version	cluster	launch date	user
profile						
-8962736786089042196	4194304	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:17:27	John
4277261189770905179	1048576	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:09:27	John

	problem size	compiler	raja version	cluster	launch date	user
profile						
-7576776928373272618	4194304	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:53:01	John
-3410224120056886347	1048576	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:45:08	Jane

Figure 3.4: Resulting metadata table after grouping metadata table from Fig. 3.1 with respect to the unique values present in the compiler column.

These new thickets only contain profiles that carry the unique value in their compiler column. In Fig. 3.4, we show a grouping where two new thickets are created as the compiler column has two unique values.

We then move on from the metadata component of a thicket and look at a capability extended by the aggregate statistics table of a thicket.

3.5 Filtering with Aggregate Statistics

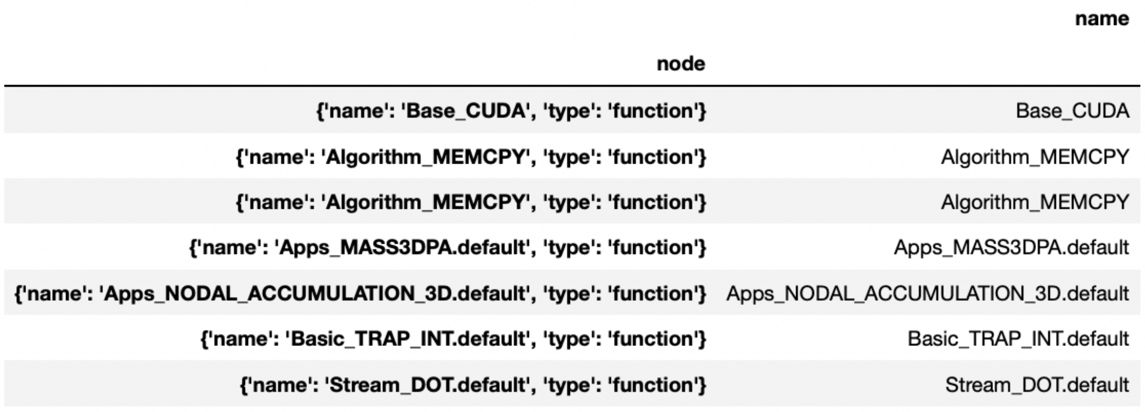
The last capability we implement is a transformation of the filtering on metadata to a different data structure, the aggregate statistics table. We enable the filtering of nodes in the statistics table based on columns and column values provided by the user. We implement the filtering of specific nodes by applying a callable function similar to the metadata filter. This is also implemented by leveraging pandas' built-in *.apply()* attribute to use a function along a table column.

We begin by creating a copy of the provided thicket object. Then, the filter function is applied to the thicket copy's statistics table. We ensure that the changes in nodes are reflected on the thicket copy's performance data table, and the new thicket is returned as output.

In Fig. 3.5, we show an example of the statistical filter applied on the table from Fig. 3.2. The filter is specified to select the Base_CUDA, Algorithm_MEMCPY, Apps_MASS3DPA.default, Apps_NODAL_ACCUMULATION_3D.default, Basic_TRAP_INT.default, Stream_DOT.default values from the name column of Fig. 3.2.

It is important to note that selecting rows from the statistics table based on the name of the nodes isn't the only possible selection. The filter allows the selection of rows based on other columns and column values once they are appended to the table.

In the next chapter, we look at such an example, where we showcase how these filtering and grouping capabilities affect the performance data. We also provide a use case where these capabilities can be used creatively for EDA.



	name
node	
{'name': 'Base_CUDA', 'type': 'function'}	Base_CUDA
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	Algorithm_MEMCPY
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	Algorithm_MEMCPY
{'name': 'Apps_MASS3DPA.default', 'type': 'function'}	Apps_MASS3DPA.default
{'name': 'Apps_NODAL_ACCUMULATION_3D.default', 'type': 'function'}	Apps_NODAL_ACCUMULATION_3D.default
{'name': 'Basic_TRAP_INT.default', 'type': 'function'}	Basic_TRAP_INT.default
{'name': 'Stream_DOT.default', 'type': 'function'}	Stream_DOT.default

Figure 3.5: A filtered empty statistics table with significantly less no. of nodes compared to the original statistics table from Fig. 3.2.

Chapter 4

Use Cases: Impact of Metadata on Analysis

This chapter presents two use cases of our newly introduced ticket's capabilities. We visualize the implications of our capabilities on the performance data and how they can be used in co-ordination with other capabilities of thicket.

4.1 Use Cases: Settings

To successfully present the use cases in this chapter, we refer to the examples of the metadata, and aggregate statistics table seen in chapter 3. We demostrate how the filtering and grouping functions on the metadata table propagate changes onto the performance data. Then, we show how filtering the nodes of the aggregate statistics table can be a helpful step for the EDA of performance data. Beyond filtering or grouping metadata and aggregate statistics, the capabilities propagate their relevant changes onto the performance data. For example, if the metadata table is filtered to select a subset of the original profiles, this change is propagated to the performance data table. Therefore, only those selected profiles will be present in the performance data of the filtered thicket object. This feature allows further EDA on the thicket object without the need to carry out these analysis on irrelevant dataset.

We use Caliper [3] for our research purposes, and it is an instrumentation-based tool for HPC performance profiling. It records performance data for every run as a call tree profile, where the collected performance metrics are assigned to nodes in the call tree. Each node corresponds to a function or nested source code region executed during the program run.

4.2 Use Case 1: Propagation of metadata filter onto performance data

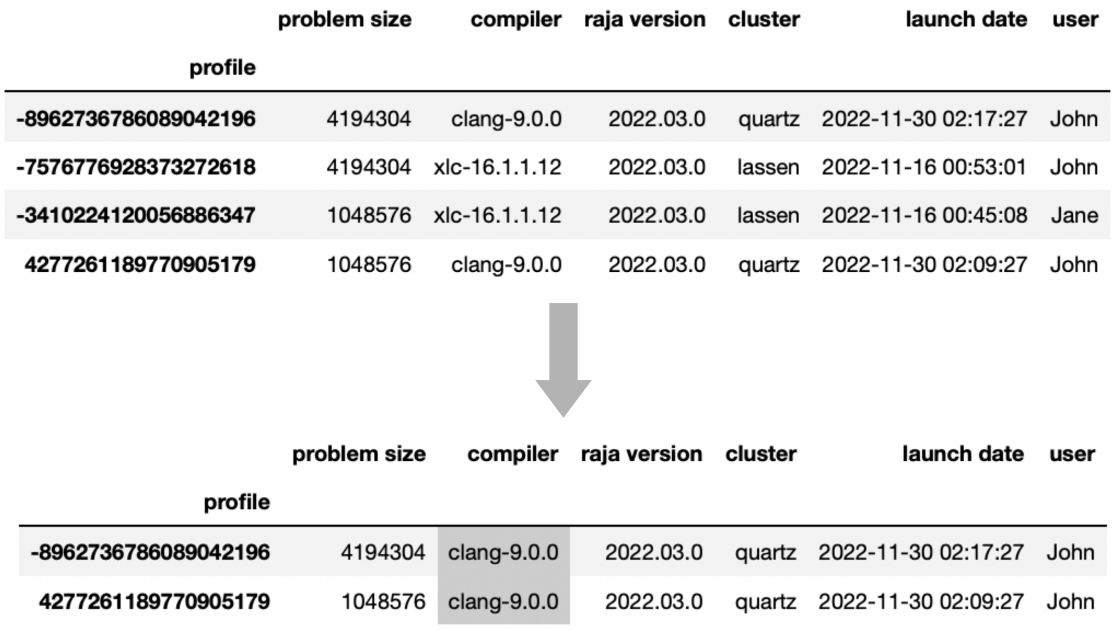
We provide an example of filtering the metadata table to preserve profile rows corresponding to the clang compiler. Fig. 4.1 shows the filter in effect, and it is important to note all the profile hashes remaining in the filtered metadata table.

We observe the profile hashes in Fig. 4.1 to determine that the original performance data table in Fig. 4.2 corresponding to the same thicket, has all the profiles as part of the multi-dimensional index. Then, on the resulting performance data table, we see that the changes in profiles made in Fig. 4.1 have propagated to the performance data table.

A small but significant observation is that as we filter the metadata table to only contain profiles corresponding to the clang compiler, the corresponding performance data has one less node index. We deduce that the profiles corresponding to the clang compiler do not contain the Base_CUDA function call, or no measurements are conducted for that specific functional call for these profiles. Similarly to the metadata filter, the statistics filter also propagates changes onto the performance data table. In the following example, we look at an example that demonstrates the stats filter.

4.3 Use Case 2: Refinement of aggregate statistics heat map using stats filter

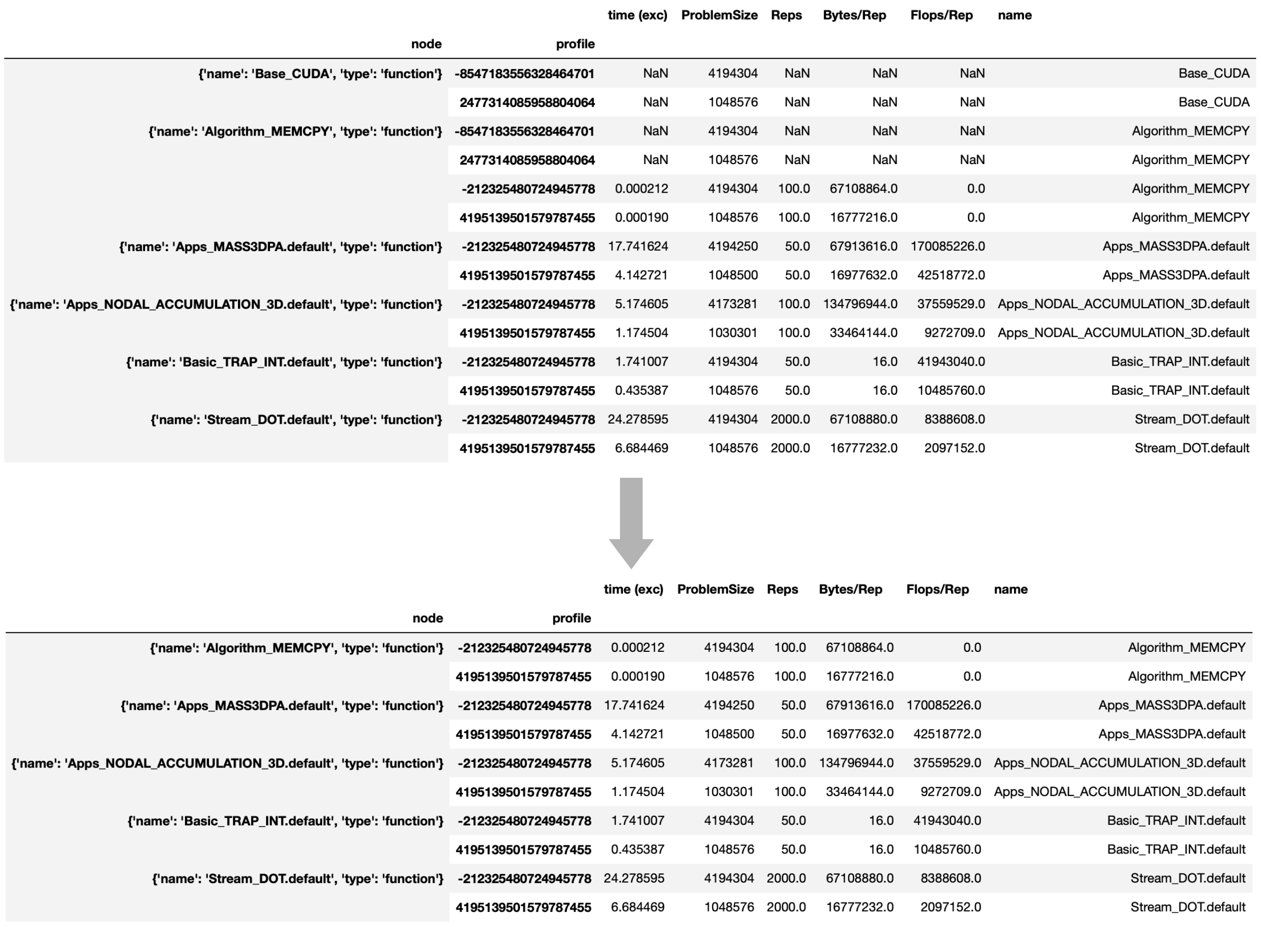
The filtering capability for the aggregate statistics table is demonstrated in Fig. 3.5. In addition to applying changes to the aggregate statistics table, the changes are also propagated to the performance data table, similar to the first example in this chapter. These capabilities can complement other thicket capabilities beyond filtering and grouping. Other statistical functions of thicket can be applied to an instantiated thicket object.



profile	problem size	compiler	raja version	cluster	launch date	user
-8962736786089042196	4194304	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:17:27	John
-7576776928373272618	4194304	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:53:01	John
-3410224120056886347	1048576	xlc-16.1.1.12	2022.03.0	lassen	2022-11-16 00:45:08	Jane
4277261189770905179	1048576	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:09:27	John

profile	problem size	compiler	raja version	cluster	launch date	user
-8962736786089042196	4194304	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:17:27	John
4277261189770905179	1048576	clang-9.0.0	2022.03.0	quartz	2022-11-30 02:09:27	John

Figure 4.1: An example of filtering metadata to only contain profiles derived from `clang` compiler.



node	profile	time (exc)	ProblemSize	Reps	Bytes/Rep	Flops/Rep	name
{'name': 'Base_CUDA', 'type': 'function'}	-8547183556328464701	NaN	4194304	NaN	NaN	NaN	Base_CUDA
	2477314085958804064	NaN	1048576	NaN	NaN	NaN	Base_CUDA
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	-8547183556328464701	NaN	4194304	NaN	NaN	NaN	Algorithm_MEMCPY
	2477314085958804064	NaN	1048576	NaN	NaN	NaN	Algorithm_MEMCPY
	-212325480724945778	0.000212	4194304	100.0	67108864.0	0.0	Algorithm_MEMCPY
	4195139501579787455	0.000190	1048576	100.0	16777216.0	0.0	Algorithm_MEMCPY
{'name': 'Apps_MASS3DPA.default', 'type': 'function'}	-212325480724945778	17.741624	4194250	50.0	67913616.0	170085226.0	Apps_MASS3DPA.default
	4195139501579787455	4.142721	1048500	50.0	16977632.0	42518772.0	Apps_MASS3DPA.default
{'name': 'Apps_NODAL_ACCUMULATION_3D.default', 'type': 'function'}	-212325480724945778	5.174605	4173281	100.0	134796944.0	37559529.0	Apps_NODAL_ACCUMULATION_3D.default
	4195139501579787455	1.174504	1030301	100.0	33464144.0	9272709.0	Apps_NODAL_ACCUMULATION_3D.default
{'name': 'Basic_TRAP_INT.default', 'type': 'function'}	-212325480724945778	1.741007	4194304	50.0	16.0	41943040.0	Basic_TRAP_INT.default
	4195139501579787455	0.435387	1048576	50.0	16.0	10485760.0	Basic_TRAP_INT.default
{'name': 'Stream_DOT.default', 'type': 'function'}	-212325480724945778	24.278595	4194304	2000.0	67108880.0	8388608.0	Stream_DOT.default
	4195139501579787455	6.684469	1048576	2000.0	16777232.0	2097152.0	Stream_DOT.default

node	profile	time (exc)	ProblemSize	Reps	Bytes/Rep	Flops/Rep	name
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	-212325480724945778	0.000212	4194304	100.0	67108864.0	0.0	Algorithm_MEMCPY
	4195139501579787455	0.000190	1048576	100.0	16777216.0	0.0	Algorithm_MEMCPY
{'name': 'Apps_MASS3DPA.default', 'type': 'function'}	-212325480724945778	17.741624	4194250	50.0	67913616.0	170085226.0	Apps_MASS3DPA.default
	4195139501579787455	4.142721	1048500	50.0	16977632.0	42518772.0	Apps_MASS3DPA.default
{'name': 'Apps_NODAL_ACCUMULATION_3D.default', 'type': 'function'}	-212325480724945778	5.174605	4173281	100.0	134796944.0	37559529.0	Apps_NODAL_ACCUMULATION_3D.default
	4195139501579787455	1.174504	1030301	100.0	33464144.0	9272709.0	Apps_NODAL_ACCUMULATION_3D.default
{'name': 'Basic_TRAP_INT.default', 'type': 'function'}	-212325480724945778	1.741007	4194304	50.0	16.0	41943040.0	Basic_TRAP_INT.default
	4195139501579787455	0.435387	1048576	50.0	16.0	10485760.0	Basic_TRAP_INT.default
{'name': 'Stream_DOT.default', 'type': 'function'}	-212325480724945778	24.278595	4194304	2000.0	67108880.0	8388608.0	Stream_DOT.default
	4195139501579787455	6.684469	1048576	2000.0	16777232.0	2097152.0	Stream_DOT.default

Figure 4.2: Propagation of metadata filter onto performance data.

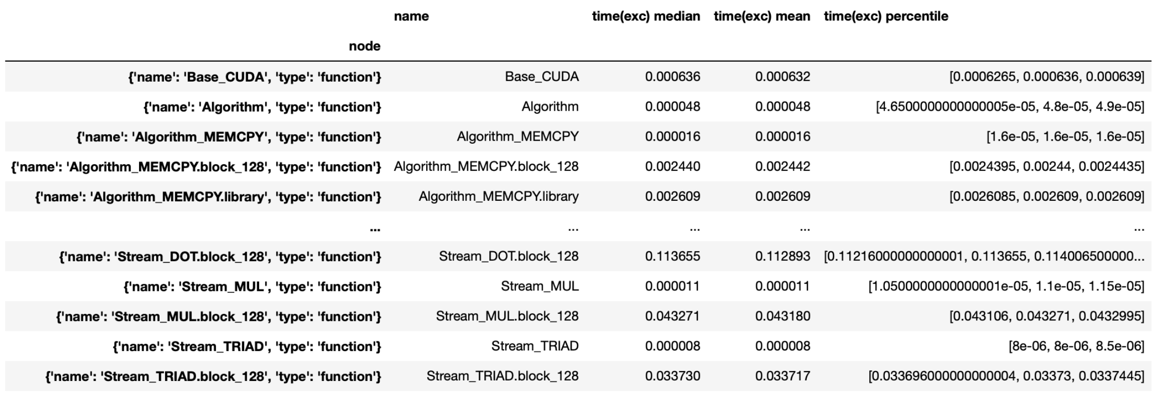
Then, after observing the results, we can use the statistical filter to manipulate the performance data table. This helps provide a more focused visualization or application of statistical functions on a selected number of nodes of the customized performance data. Fig. 4.3 shows an example of a thicket aggregate statistics table. Compared to the empty stats table from Fig. 3.2, this table has appended results. The appended results are the mean, median, and percentile of the exclusive run-time for each node from the performance data table.

We provide a range of statistical and visualization functionalities as part of the thicket API. One of these, is the ability to generate heat maps using a column from the appended aggregate statistics table. The table in Fig.4.3 has multiple rows of information, and we can generate the heat map of all these rows for the median exclusive time column. Before we perform any visualization on this table, we sort this table on the basis of the average (median) exclusive run-time for each node. We then have a resulting table as can be seen in Fig. 4.4.

We will now use this sorted table to generate a heat-map of all the nodes present, based on the average exclusive run-time to get a proper understanding of how the run-times are distributed for each of these nodes on average. The sorting of table based on the median column help with this visualization technique by allowing us to easily separate the longer and shorter run-times.

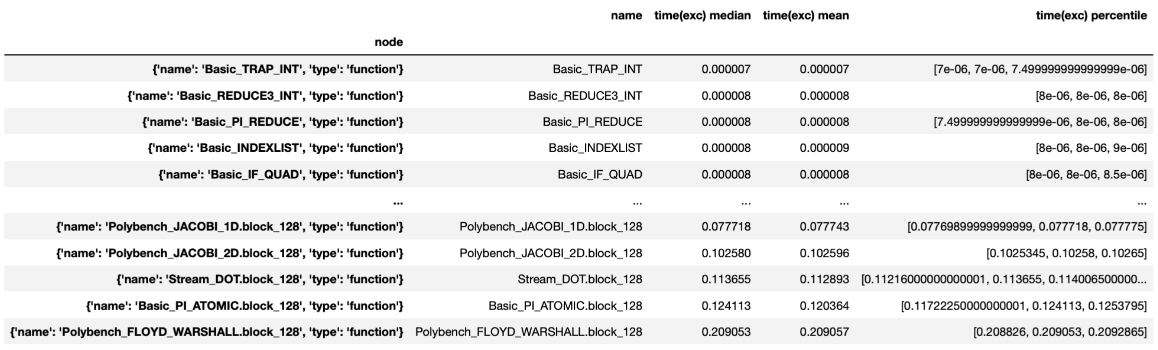
Fig. 4.5 represents the heat map with this setting. The heat map is color coded, where the lighter the color, the higher the average run-time, and the darker the color, the lower the average run-time. The capability of generating heat maps help visualize the overall distribution of the nodes and their corresponding average run-time. However, the data is too large at this scale for proper analysis. This figure is much more helpful in getting an overview of the data and then determining what nodes to focus on.

This is a case where filtering the statistics table is complimentary in combination with the visualization. In Fig. 4.6, we present a much shorter table than the table in Fig. 4.4 with a chosen set of nodes. We achieve this using the statistics filter to filter down to just the Basic_TRAP_INT, Basic_REDUCE3_INT, Basic_PI_REDUCE, Stream_TRIAD, and the Stream_TRIAD.block_128 nodes.



	name	time(exc) median	time(exc) mean	time(exc) percentile
node				
{'name': 'Base_CUDA', 'type': 'function'}	Base_CUDA	0.000636	0.000632	[0.0006265, 0.000636, 0.000639]
{'name': 'Algorithm', 'type': 'function'}	Algorithm	0.000048	0.000048	[4.6500000000000005e-05, 4.8e-05, 4.9e-05]
{'name': 'Algorithm_MEMCPY', 'type': 'function'}	Algorithm_MEMCPY	0.000016	0.000016	[1.6e-05, 1.6e-05, 1.6e-05]
{'name': 'Algorithm_MEMCPY.block_128', 'type': 'function'}	Algorithm_MEMCPY.block_128	0.002440	0.002442	[0.0024395, 0.00244, 0.0024435]
{'name': 'Algorithm_MEMCPY.library', 'type': 'function'}	Algorithm_MEMCPY.library	0.002609	0.002609	[0.0026085, 0.002609, 0.002609]
...	...	...	...	...
{'name': 'Stream_DOT.block_128', 'type': 'function'}	Stream_DOT.block_128	0.113655	0.112893	[0.11216000000000001, 0.113655, 0.114006500000...
{'name': 'Stream_MUL', 'type': 'function'}	Stream_MUL	0.000011	0.000011	[1.0500000000000001e-05, 1.1e-05, 1.15e-05]
{'name': 'Stream_MUL.block_128', 'type': 'function'}	Stream_MUL.block_128	0.043271	0.043180	[0.043106, 0.043271, 0.0432995]
{'name': 'Stream_TRIAD', 'type': 'function'}	Stream_TRIAD	0.000008	0.000008	[8e-06, 8e-06, 8.5e-06]
{'name': 'Stream_TRIAD.block_128', 'type': 'function'}	Stream_TRIAD.block_128	0.033730	0.033717	[0.033696000000000004, 0.03373, 0.0337445]

Figure 4.3: An aggregate statistics table with the inclusive time metrics mean, median, and percentile values.



	name	time(exc) median	time(exc) mean	time(exc) percentile
node				
{'name': 'Basic_TRAP_INT', 'type': 'function'}	Basic_TRAP_INT	0.000007	0.000007	[7e-06, 7e-06, 7.499999999999999e-06]
{'name': 'Basic_REDUCE3_INT', 'type': 'function'}	Basic_REDUCE3_INT	0.000008	0.000008	[8e-06, 8e-06, 8e-06]
{'name': 'Basic_PI_REDUCE', 'type': 'function'}	Basic_PI_REDUCE	0.000008	0.000008	[7.499999999999999e-06, 8e-06, 8e-06]
{'name': 'Basic_INDEXLIST', 'type': 'function'}	Basic_INDEXLIST	0.000008	0.000009	[8e-06, 8e-06, 9e-06]
{'name': 'Basic_IF_QUAD', 'type': 'function'}	Basic_IF_QUAD	0.000008	0.000008	[8e-06, 8e-06, 8.5e-06]
...	...	...	...	...
{'name': 'Polybench_JACOBI_1D.block_128', 'type': 'function'}	Polybench_JACOBI_1D.block_128	0.077718	0.077743	[0.07769899999999999, 0.077718, 0.077775]
{'name': 'Polybench_JACOBI_2D.block_128', 'type': 'function'}	Polybench_JACOBI_2D.block_128	0.102580	0.102596	[0.1025345, 0.10258, 0.10265]
{'name': 'Stream_DOT.block_128', 'type': 'function'}	Stream_DOT.block_128	0.113655	0.112893	[0.11216000000000001, 0.113655, 0.114006500000...
{'name': 'Basic_PI_ATOMIC.block_128', 'type': 'function'}	Basic_PI_ATOMIC.block_128	0.124113	0.120364	[0.11722250000000001, 0.124113, 0.1253795]
{'name': 'Polybench_FLOYD_WARSHALL.block_128', 'type': 'function'}	Polybench_FLOYD_WARSHALL.block_128	0.209053	0.209057	[0.208826, 0.209053, 0.2092865]

Figure 4.4: The aggregate statistics table from Fig. 4.3 sorted on the basis of the average (median) exclusive run-time, or the third column from the referenced table.

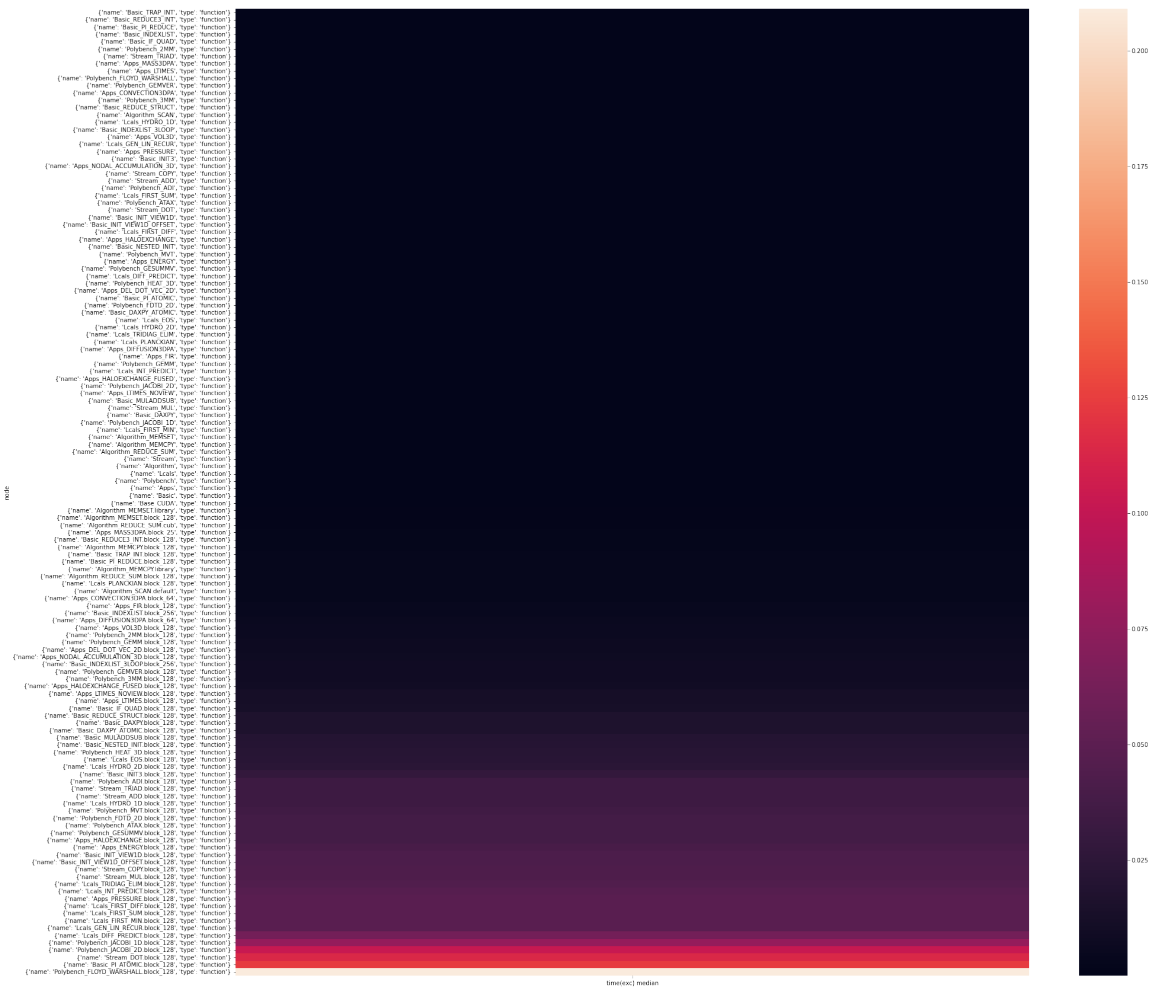
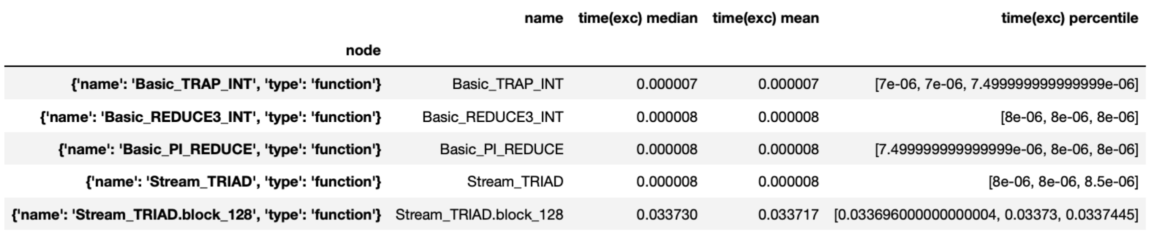


Figure 4.5: The heat map generated with respect to the time(exc) median metric from Fig. 4.4



node	name	time(exc) median	time(exc) mean	time(exc) percentile
{'name': 'Basic_TRAP_INT', 'type': 'function'}	Basic_TRAP_INT	0.000007	0.000007	[7e-06, 7e-06, 7.499999999999999e-06]
{'name': 'Basic_REDUCE3_INT', 'type': 'function'}	Basic_REDUCE3_INT	0.000008	0.000008	[8e-06, 8e-06, 8e-06]
{'name': 'Basic_PI_REDUCE', 'type': 'function'}	Basic_PI_REDUCE	0.000008	0.000008	[7.499999999999999e-06, 8e-06, 8e-06]
{'name': 'Stream_TRIAD', 'type': 'function'}	Stream_TRIAD	0.000008	0.000008	[8e-06, 8e-06, 8.5e-06]
{'name': 'Stream_TRIAD.block_128', 'type': 'function'}	Stream_TRIAD.block_128	0.033730	0.033717	[0.033696000000000004, 0.03373, 0.0337445]

Figure 4.6: The stats filter applied to Fig. 4.4, reducing the number of nodes to six.

The filtering of nodes for this specific example is done to take a more concentrated look at the nodes with the highest three average run-times and the lowest two run-times. As a result of the filtering, we end up with a much more readable and precise heat map in Fig. 4.7. The figure truly enables the EDA of performance data as a combination of manipulation and visualization of the performance data.

We provide users with the flexibility to manipulate data with either of the three components of the thicket. In practice, a user can append several other statistical calculations to the statistics table. Then, generate heat maps based on those calculations, using them as the metric for this visualization. The thicket's several statistical and visualization capabilities make exploring thicket performance data non-linear, where a user can either manipulate performance data, perform statistical calculations, or visualize data in any order and any number of times.

And as demonstrated in the last example, it can be used in combination to make conclusions about performance. The capabilities of the thicket are extendable and can include a wide variety of implications in the future. In the next chapter, chapter 5, we provide an overview of current improvements being implemented by the PAVE team and possible future avenues for thicket.

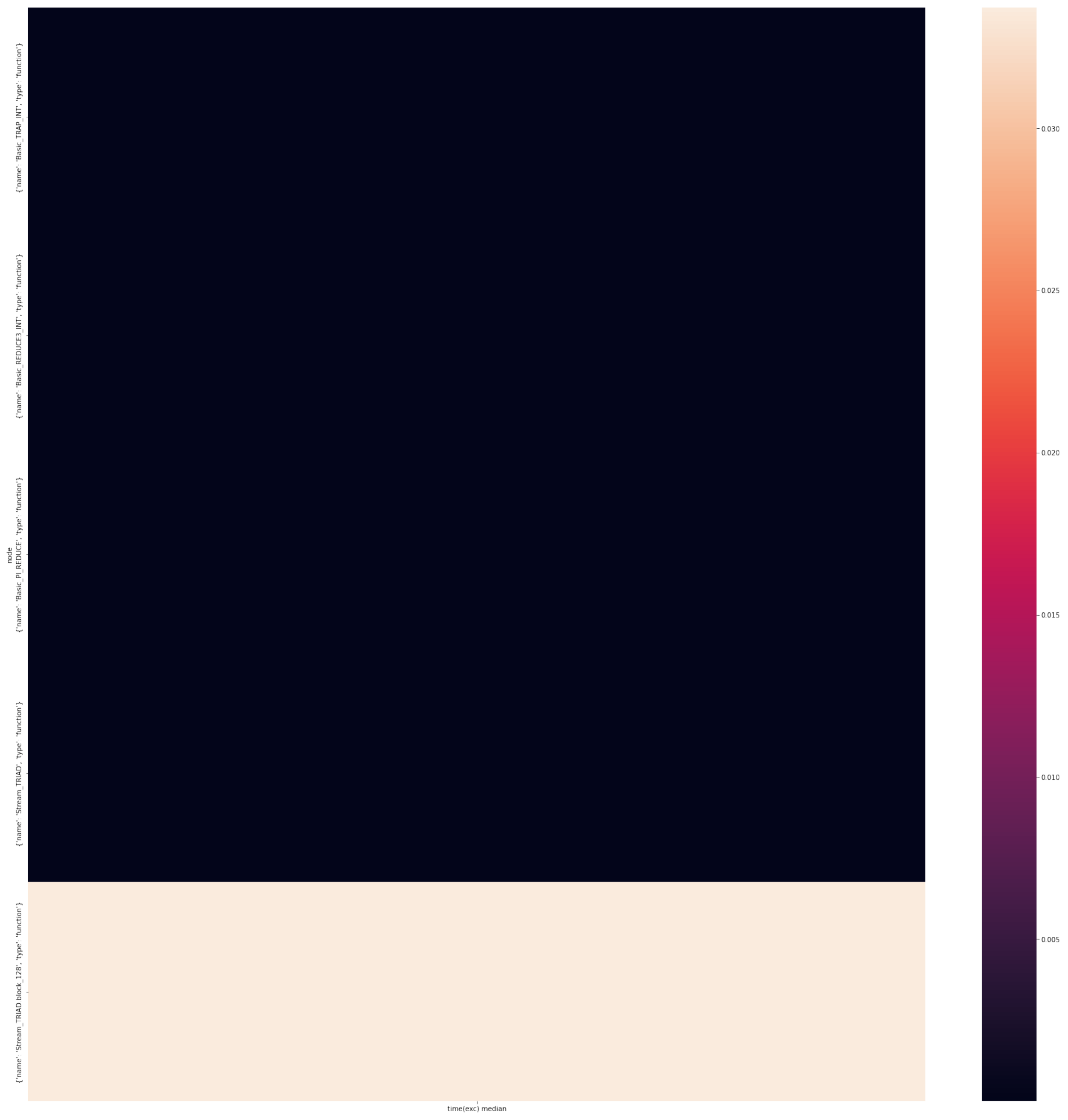


Figure 4.7: The heat map generated with respect to the time(exc) median metric from Fig. 4.6

Chapter 5

Conclusion and Future Work

In this chapter, we summarize our thesis and provide insight into future work that can extend the capabilities of thicket.

5.1 Summary

We provided an overview of four performance tools, HPCToolkit [1, 8], Score-P [7], TAU [9], and Allinea MAP [6], that generate profiles for different programs runs. These tools from chapter 2 also allow the analysis of performance data contained within the profiles.

We introduced thicket [5], a performance analysis tool that enables the analysis of an ensemble of the program runs together. We established that thicket differs from the rest of the performance tools in chapter 2 as it allows multiple profiles to be read into a single object for a thorough analysis instead of a single profile at a time. The three components of thicket were also introduced, providing the background necessary to understand our contributions to thicket.

We identified our contributions to thicket, which include extending the capabilities of two out of the three components of thicket: the metadata and aggregate statistics. The capabilities mentioned in chapter 3 is the filtering and grouping functions concerning the metadata table and the filtering function concerning the statistics table. We also demonstrated the effect of these functions with simple examples of the metadata and statistics tables.

We provided the use case settings for two cases demonstrated in chapter 4. These use cases helped us expand on the effects of the filtering and grouping functions from chapter 3 beyond the metadata and statistics tables. We displayed the visual representation of both use cases with the help of metadata and statistics table examples, the performance data table, and a heat map of the corresponding statistics table.

We found that the filtering and grouping capabilities are essential to the post-processing of performance data and can be combined with several other thicket capabilities for meaningful EDA.

5.2 Future Work

In the future, we are considering leveraging deep learning neural networks to take data from the tables of a thicket object and deduce optimal settings for running a program. An example of this would be using neural networks to provide users with a suggestion on what compiler will result in optimal performance while executing a program.

Another extension to thicket's capabilities is to export data stored in thicket objects directly onto pre-existing visualization tools like Tableau. This opens up the opportunity for scientists to visualize these data directly without and extensive knowledge of the code base.

We are expanding the accessibility of thicket by providing an open-access suite of Jupyter notebooks demonstrating additional capabilities of thicket beyond the work of this thesis. Easy-to-follow tutorials will also be provided as part of extensive documentation for thicket in the Read the Docs format. The suite of Jupyter notebooks will also be executable with the help of the Binder platform.

Bibliography

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. (2010). Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22:685–701. 4, 5, 31

[2] Bhatele, A., Brink, S., and Gamblin, T. (2019). Hatchet: Pruning the Overgrowth in Parallel Profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA. Association for Computing Machinery. 10

[3] Boehme, D., Gamblin, T., Beckingsale, D., Bremer, P.-T., Gimenez, A., LeGendre, M., Pearce, O., and Schulz, M. (2016). Caliper: Performance Introspection for HPC Software Stacks. SC '16. IEEE Press. 4, 7, 20

[4] Brink, S., Lumsden, I., Scully-Allison, C., Williams, K., Pearce, O., Gamblin, T., Taufer, M., Isaacs, K. E., and Bhatele, A. (2020). Usability and Performance Improvements in Hatchet. In *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 49–58. 10

[5] Brink, S., McKinsey, M., Boehme, D., Hawkins, D., Scully-Allison, C., Lumsden, I., Burgess, T., Lama, V., Isaacs, K. E., Luettgau, J., Taufer, M., and Pearce, O. (2023). Thicket: Seeing the performance experiment forest for the individual run trees. In *High-Performance Parallel and Distributed Computing (HPDC 2023)*. 7, 8, 31

[6] January, C., Byrd, J., Oró, X., and O'Connor, M. (2015). Allinea map: Adding energy and openmp profiling without increasing overhead. In Niethammer, C., Gracia, J., Knüpfer, A., Resch, M. M., and Nagel, W. E., editors, *Tools for High Performance Computing 2014*, pages 25–35, Cham. Springer International Publishing. 4, 6, 31

[7] Knüpfer, A., Rössel, C., Mey, D. a., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S., Tschüter, R., Wagner, M., Wesarg, B., and

Wolf, F. (2012). Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir. In Brunst, H., Müller, M. S., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2011*, pages 79–91, Berlin, Heidelberg. Springer Berlin Heidelberg. 4, 5, 31

[8] Mellor-Crummey, J. (2003). HPCToolkit: Multi-platform tools for profile-based performance analysis. In *5th International Workshop on Automatic Performance Analysis (APART)*. 4, 5, 31

[9] Shende, S. S. and Malony, A. D. (2006). The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2):287–311. 4, 6, 31

Vita

Vanessa Lama is an M.S. student in Computer Science under the advisory of Dr. Michela Taufer at the University of Tennessee, Knoxville. Her expected M.S. graduation date is in Spring 2023. She has a B.S. in Computer Science and Mathematics from McNeese State University, Lake Charles, LA. Vanessa's research interests in high-performance computing include the development of performance data analysis tool, as well as bench marking and evaluating machine learning and deep learning frameworks assisted by HPC. She has been involved in projects with HPC, machine learning applications, and overall research software development.