Automated Programmatic Performance Analysis of Parallel Programs

Onur Cankur  
ocankur@umd.edu  
Department of Computer Science,  
University of Maryland  
College Park, Maryland, USA

Aditya Tomar  
adityatomar@berkeley.edu  
Department of Electrical Engineering and Computer Sciences, University of California, Berkeley  
California, USA

Daniel Nichols  
dnicho@umd.edu  
Department of Computer Science,  
University of Maryland  
College Park, Maryland, USA

Connor Scully-Allison  
cscullyallison@email.arizona.edu  
Department of Computer Science,  
The University of Arizona  
Arizone, USA

Katherine E. Isaacs  
kisaacs@sci.utah.edu  
Department of Computer Science,  
The University of Utah  
Utah, USA

Abhinav Bhatele  
bhatele@cs.umd.edu  
Department of Computer Science,  
University of Maryland  
College Park, Maryland, USA

ABSTRACT

Developing efficient parallel applications is critical to advancing scientific development but requires significant performance analysis and optimization. Performance analysis tools help developers manage the increasing complexity and scale of performance data, but often rely on the user to manually explore low-level data and are rigid in how the data can be manipulated. We propose a Python-based API, Chopper, which provides high-level and flexible performance analysis for both single and multiple executions of parallel applications. Chopper facilitates performance analysis and reduces developer effort by providing configurable high-level methods for common performance analysis tasks such as calculating load imbalance, hot paths, scalability bottlenecks, correlation between metrics and CCT nodes, and causes of performance variability within a robust and mature Python environment that provides fluid access to lower-level data manipulations. We demonstrate how Chopper allows developers to quickly and succinctly explore performance and identify issues across applications such as AMG, Laghos, LULESH, Quicksilver and Tortuga.

KEYWORDS

simplified, performance, analysis, parallel

1 INTRODUCTION

Ensuring that parallel applications run efficiently on modern supercomputers is essential to achieve scientific discoveries rapidly. Identifying performance problems is the first step in the process of optimizing performance of a parallel program. However, performance analysis is a complex and time-consuming task due to the inherent complexity of large-scale parallel applications and architectures, and the large quantity of performance data that can be collected when running in parallel. In addition, parallel applications may suffer from a variety of performance issues. Therefore, in order to minimize the developer’s burden, we require highly effective performance analysis techniques that can quickly identify performance problems and their root causes.

A variety of performance measurement tools exist, including profilers and tracing tools that can generate performance data [1, 7, 17, 28]. However, the data generated can be extremely large, making it challenging to sift through this data to identify performance issues. Several performance measurement tools also provide visual analytics counterparts to facilitate performance analysis. Typically, the analysis support is in the form of a graphical user interface (GUI) to visualize and manipulate performance data [3, 14, 16, 22] although some tools also provide a scripting interface [9, 26]. The GUIs help in visualizing performance data and in many cases, the user can connect such data to source code (file and line numbers).

Although GUIs provide some effective functionalities, having to analyze performance data only via a GUI can make it inefficient to identify performance issues. GUIs depend on the user to manually explore the visualizations, and to identify different patterns that might suggest performance problems. As the data being analyzed grows, this becomes more and more challenging. In addition, to analyze multiple executions, GUIs typically require opening multiple separate windows with the datasets. Some of them can visualize multiple datasets on the same window, however, they still require significant manual effort to compare different executions. Finally, adding new kinds of analyses on top of a GUI may not be possible for an end user.

The main aim of this work is to simplify common performance analysis tasks for the end user by reducing the time and effort required. We present a Python-based API that simplifies several performance analysis tasks, and offers flexibility and customization to enable users to perform analyses with speed and effectiveness. To achieve this goal, we explored the common functionalities in other performance analysis tools and also collected feedback from developers and users of performance tools to identify the most needed functionalities. We develop this new API on top of Hatchet that provides an interface for programmatic analysis of performance data via Python [5].

By virtue of being developed on top of Hatchet, Chopper supports data formats of various performance tools, including but not limited
we give background information on profiling, call graphs, and common performance analysis techniques. We also mention Hatchet and other related work.

2.1 Profiling and Call Graphs

Generally, there are two methods for performance measurement: profiling and tracing. Profiling provides a statistical approximation instead of exact timestamps for each event in the program, unlike tracing. In this paper we focus on profiling.

The performance data generated by profiling tools provide a variety of information such as function call sequences, performance metrics (e.g., time, cache misses, floating-point operations per second), and MPI process topologies. In this work we study the analysis of calling context trees (CCT) and call graphs, in both of which the nodes typically represent procedures and edges represent the caller-callee relationships (i.e., function call sequences). A path from any given node to the root is called a call path or calling context. A collection of distinct calling contexts forms a CCT. Unlike CCTs, in call graphs a procedure that is called in different call paths is represented as a single node with aggregated metric values. Therefore, call graphs are less context-sensitive, but they provide a more concise representation. Additionally, performance data typically contains information about function names, file names, line numbers, and process or thread IDs.

2.2 Hatchet

Hatchet is a Python-based tool that enables analyzing hierarchical data, such as CCTs and call graphs, programmatically [5]. It reads performance data from several profiling tools (e.g., Caliper, HPCToolkit, Score-P, TAU, timer) and provides an interface for programmatic analysis of performance data. It also provides several visualization functionalities such as terminal, DOT, and interactive Jupyter notebook visualization. It supports low level operations to manipulate the data and requires significant programming.

Hatchet’s central data structure is called GraphFrame, which is a combination of a pandas DataFrame [20, 21] and a Graph. It stores the caller-callee relationships in the graph object and the associated performance metrics and contextual information in the DataFrame. Hatchet provides graph-indexed DataFrames, which means every index of the DataFrame points to a node in the graph. Therefore, these two data structures are connected and can be manipulated together. This data structure enables the practical implementation of different analysis tasks. We utilize Hatchet to implement our analysis API, Chopper.

2.3 Common Performance Analysis Problems

The performance of a parallel application can suffer from communication or computational inefficiencies. Performance problems can be revealed by investigating imbalances, scalability, variability, and hot paths in the program.

Load Imbalance: Parallel programs use multiple processing elements (e.g., processes and threads). Ideally, the work done by the program should be equally distributed over processing elements, so that they can finish their tasks at the same time. However, the ideal scenario is almost never achieved in complex workflows, which makes the load imbalance a common problem. It can be identified by investigating the cost incurred by different processes.

Hot Paths: One way to pinpoint the bottlenecks in the program is to examine the most time-consuming call paths. This task is called hot path analysis [2]. For a given metric (e.g., time), every node in a hot path accounts for more than 50% percent of its parents. Manually finding the hot path is a tedious task when there are millions of nodes in the call graph.

Poor Scalability: Scalability analysis shows how well a program utilizes the increasing number of processing elements. A program that has poor scalability may work slower than expected despite using more processing elements. Scalability problems can be identified by performing scaling experiments and observing the change in speedup and efficiency.

Performance Variation: The performance of a program may differ in different runs even though all of the parameters used in each run are the same (e.g., hardware architecture and input parameters). For example network congestion can lead to variability in performance [6]. Variability can be identified by analyzing multiple identical runs of a program.

2.4 Related Work

The idea of analyzing single and multiple call graph data to pinpoint bottlenecks is defined as differential profiling by early work [19].
Later works demonstrated the usefulness of manipulating and visualizing call graph data by performing differential analysis [27]. With that knowledge many studies utilized the call graph data to effectively identify and visualize performance problems. Several studies manipulated performance metrics to identify load imbalances [11, 13, 29]. Adhianto et al. [2] defined hot path analysis and demonstrated how to perform it using HPCViewer. Several studies demonstrated applying differential profiling by using call path profiles to analyze the scalability of the programs [10, 18, 30]. Benedict et al. [4] examined the scalability of the programs by instrumenting the region of interests in the programs and analyzing the performance of different processes on those regions. Variability in performance of HPC applications is another commonly studied topic [24] However, to the best of our knowledge, there is no study on analyzing performance variability using call graph data.

2.4.1 Performance Analysis Tools. Many performance analysis tools are developed to facilitate performance analysis. Table 1 provides a summary of different tools. Cube [26] is a performance analysis tool for Score-P. Extra-P [9] is an automated performance modeling tool that focuses mostly on scaling behavior of applications. HPCViewer [22] enables analyzing profile and trace data generated by HPCToolkit. ParaProf [3] is also a performance analysis tool and a part of TAU toolkit. It supports several different profile data formats. All of these tools present call graph along with performance metrics. Additionally, Cube and ParaProf can visualize MPI process topologies. PerfExplorer [14] framework is also a part of TAU toolkit and supports several data mining operations such as correlation analysis and clustering. Thicket [8], provides performance analysis capabilities for multi-run performance experiments. It utilizes Hatchet and Extra-P and develop new capabilities on top of them. Even though all of these tools provide useful analysis capabilities, most of them provide only a desktop GUI. However, GUIs typically are not as flexible and dynamic as a programmatic interface and they do not provide rich APIs to manipulate the profile data. This limitation becomes more obvious when the data is very large and complex. Additionally, making changes or adding new capabilities to a GUI is hardly possible for the end user. Thicket provides programmatic analysis capabilities but only focuses on multi-run analysis.

We propose a Python-based API, Chopper, to overcome these limitations. Chopper facilitates performance analysis by simplifying several single-run and multi-run performance analysis tasks and making them easier and more intuitive to perform. We utilize Hatchet’s programmatic interface and visualization capabilities to implement the analysis tasks. With the Chopper API the users can identify performance problems in their parallel programs by writing only a few lines of Python code. Chopper reduces the effort and time spent on performance analysis.

<table>
<thead>
<tr>
<th></th>
<th>Hot Path Analysis</th>
<th>Load Imb. Analysis</th>
<th>Programmatic API</th>
<th>Flat Profile</th>
<th>Speedup Analysis</th>
<th>Correlation Analysis</th>
<th>Multirun Analysis</th>
<th>Perf. Modeling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube</td>
<td>Manual</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Manual</td>
<td>x</td>
<td>Manual</td>
<td>x</td>
</tr>
<tr>
<td>Extra-P</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>Manual</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ParaProf</td>
<td>Manual</td>
<td>Manual</td>
<td>Guided</td>
<td>Manual</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>x</td>
</tr>
<tr>
<td>Thicket</td>
<td>Manual</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>x</td>
</tr>
<tr>
<td>This work</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>Guided</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 1: Capabilities in different profile analysis and visualization tools.
Chopper, we considered them and the design issues surrounding them separately. Chopper provides a unified interface for invoking functions from either category. However, single-run analysis tasks can also be invoked from a GraphFrame object in Hatchet.

4 CHOPPER: A PYTHON API FOR PERFORMANCE ANALYSIS

We describe the design of API we implemented, called “Chopper” because it helps manipulate calling context trees (and call graphs). Chopper facilitates a range of analysis tasks for both single (subsection 4.1) and multiple executions (subsection 4.2).

4.1 Analyzing a Single Execution

Through Chopper, we add higher-level performance analysis operations to the lower-level performance metrics and manipulations offered by Hatchet. To provide a seamless experience, we augment the Hatchet GraphFrame (pandas dataframe + graph) object so Chopper methods can be called directly. These methods are also available directly through the Chopper API by passing the GraphFrame object. We describe Chopper’s single-run capabilities below.

```python
import hatchet as ht
load_ext hatchet.vis.loader
gf = ht.GraphFrame.from_literal("simple-cct")
ccf = gf.Hatchet’s Jupyter CCT visualization (a)
callgraph_graphframe = gf.to_callgraph() # (b)
```

Figure 1: Creating a callgraph from a CCT using the `to_callgraph` function. Hatchet’s Jupyter notebook visualization is used to visualize the CCT (a). The call graph (b) is visualized externally.

```python
gf = ht.Chopper.load imbalance(metric_column="time", verbose=True)
```

Figure 2: Calculating the load imbalance of a 512 process execution for LULESH by using the `load imbalance` function. The resulting DataFrame is sorted by the `time imbalance` column which shows the imbalance value for each CCT node.

**load imbalance:** Load imbalance is a common performance problem in parallel programs. Developers and application users are interested in identifying load imbalance so they can improve the distribution of work among processes or threads. The `load imbalance` function in Chopper makes it easier to study load imbalance at the level of individual CCT nodes.

Algorithm 1 summarizes the `load imbalance` function. The input is a GraphFrame along with the metric on which to compute imbalance. Optional parameters are a threshold value to filter out inconsequential nodes and a flag for calculate detailed statistics about the load imbalance. The output is a new GraphFrame with the same graph object but additional columns in its DataFrame to describe load imbalance and optionally the verbose statistics. A full example of `load imbalance` is shown in Figure 2.

To calculate per-node load imbalance, we use pandas DataFrame operations to compute the mean and maximum of the given metric across all processes (line 4 and 5). Load balance is then the maximum divided by the mean (line 15). A large maximum-to-mean ratio indicates heavy load imbalance. The per-node load imbalance value is added as a new column in DataFrame.
The threshold parameter is used to filter out nodes with metric values below the given threshold (line 13). This feature allows users to remove nodes that might have high imbalance because their metric values are small. For example, high load imbalance may not have significant impact on overall performance in the time spent in the node is small.

The verbose option calculates additional statistics. If enabled, the function adds a new column to the resulting DataFrame with each of the following: the top five ranks that have the highest metric values (line 7), values of 0th, 25th, 50th, 75th, and 100th percentiles of each node (line 8), and the number of processes in each of ten equal-sized bins between the 0th (minimum across processes) and 100th (maximum across processes) percentile values (line 9).

Algorithm 1 Pseudocode of load_imbalance

1: function load_imbalance(graphframe, metric, threshold, verbose)
2:   dataframe ← graphframe.dataframe
3:   for nodes ∈ dataframe do
4:     dataframe[metric.max] ← max across processes
5:     dataframe[metric.mean] ← mean across processes
6:     if verbose then
7:       dataframe[metric.percentile] ← top five ranks
8:       dataframe[metric.ranks] ← percentile values
9:     end if
10:   end for
11: end function

Figure 3: Identifying the hot path of a simple CCT using the hot_path function in Chopper. The red-colored path with bigger, labeled nodes represents the hot path.

hot_path: A common task in analyzing a single execution is to examine the most time-consuming call paths in the program or some subset of the program. Seeking out the latter call paths can be tedious in a GUI, especially if the CCT is large and complex. Chopper’s hot_path function retrieves the hot path from any subtree of a CCT given its root. The input parameters are the GraphFrame, metric (and optional stopping condition), and the root of the subtree to search. Starting at the given subtree root, the method traverses the graph it finds a node whose metric accounts for more than a given percentage of that of its parent. This percentage is the stopping condition. The hot path is then the path between that node and the given subtree root. The function outputs a list of nodes using which the DataFrame can be manipulated.

By default, the hot_path function uses the most time-consuming root node (in case of a forest) as the subtree root. The default stopping condition is 50%, which we chose based on its utility as identified by Adhianto et al. [2]. The resulting hot path can be visualized in the context of the CCT using the interactive Jupyter visualization in Hatchet. We validated our implementation by comparing our results with hpcviewer.

Figure 3 shows the hot path for a simple CCT example, found with a single Chopper function call (line 5) and visualized using Hatchet’s Jupyter notebook visualization (line 6). The red-colored path with the large red nodes and additional labeling represents the hot path. Users can interactively expand or collapse subtrees to investigate the CCT further.

correlation_analysis: Profiling data may include numerous metrics and CCT nodes and it is important to analyze correlation between them to understand the program behaviour. To facilitate this analysis, the Chopper API provides two main functions: correlation_analysis and pairwise_correlation. The correlation_analysis function calculates the correlation between different performance metrics such as time, cache misses, and branch misses. It accepts a GraphFrame, list of metrics, and a method to calculate correlations (e.g., Pearson, Spearman, Kendall). It outputs the correlation matrix. In order to simplify the analysis, Chopper provides the filter_correlation_matrix function that filters the correlation matrix based on the correlation value. The pairwise_correlation function provides a more granular view, examining the relationship of two metrics at the level of individual CCT nodes. This function performs linear regression and fits a linear model to the data, assuming linear relationship between two performance metrics. The CCT nodes that diverge significantly from the fitted line might imply unusual behavior within the program and aid users to identify potential issues. The pairwise_correlation function adds the values on the regression line and the distances of each CCT node to the GraphFrame’s DataFrame.

4.2 Comparing Multiple Executions

Performance often only makes sense in the context of multiple executions, for example, understanding weak or strong scaling. However, GUI-based performance tools are often focused on single executions. While Hatchet has a few simple pairwise operations on two GraphFrames (i.e., two executions), three or more executions require programming ad hoc analyses. Chopper implements several capabilities for comparing performance across several executions, targeting common analyses such as those in studies of scaling scaling or performance variability. These are implemented as static functions of the Chopper API.

construct_from: Ingesting multiple datasets is the first step to analyzing them. It is laborious and tedious to specify and load
datasets = glob.glob("list_of_lulesh_profiles")
gfs = hatchet.GraphFrame.construct_from(datasets)
table = hatchet.Chopper.multirun_analysis(gfs)
print(table)

Figure 4: The multirun_analysis function returns a pivot table containing node names and time values of the nodes in each profile. We show a truncated example of the returned pivot table from a set of LULESH weak scaling executions (64, 125, 216, and 512 processes).

datasets = glob.glob("list_of_lulesh_profiles")
gfs = hatchet.GraphFrame.construct_from(datasets)
print(gfs)

datasets = glob.glob("list_of_lulesh_profiles")
print(gfs)

gf2

Figure 5: GraphFrames before and after unification by the unify_multiple_graphframes function. The resulting GraphFrames include all nodes from the given GraphFrames but retain their original metric values.

By default, multirun_analysis builds a unified "pivot table" of the multiple executions for a given metric. The index (or "pivot") is the execution identifier. Per-execution, the metrics are also aggregated by the function name. This allows users to quickly summarize across executions and their composite functions for any metric.

multirun_analysis allows flexibly setting the desired index, columns (e.g., using file or module rather than function name), and metrics with which to construct the pivot table. It also provides filtering of nodes below a threshold value of the metric. The code block in Figure 4 demonstrates multirun_analysis with default parameters (line 3) and its resulting table.

As we will show in Section 7, the multirun_analysis function makes it straightforward to analyze multiple executions and significantly reduces end-user effort. Most importantly, users can easily manipulate the pivot table programmatically or generate different ones for different analysis tasks such as scaling and variability. In addition, it is possible to plot the data in this pivot table with only a single line of Python code. This is normally a laborious task to perform using only a GUI.

unify_multiple_graphframes:
Fine-grained analysis tasks may require preserving those individual metrics and CCT topology in order to match them across CCT nodes. Combining multiple large parallel profiles takes significant programming effort. We automate this task through the unify_multiple_graphframes function, which takes multiple GraphFrames as inputs and updates each GraphFrame in place.

The unify_multiple_graphframes function creates a union graph object from all input GraphFrames from the collection of unique call paths. The updated GraphFrames point to this new object and the DataFrame of each is updated with the missing nodes. The operation ensures that all input GraphFrames are associated with the same unified graph and have individually updated DataFrames.
Figure 7: Log-log plot of the runtime of the hot_path and load_imbalance, and from_hpctoolkit functions (left) and memory consumption of from_hpctoolkit (right). We observe that all of the functions scale linearly with data size and memory consumption by the Chopper API functions does not exceed that of file reading.

Figure 6 shows the output DataFrame of efficiency values from a weak scaling (64 to 512 process) experiment of LULESH along with the corresponding code block (line 3). The DataFrame can then be used directly to plot the results.

5 EXPERIMENTAL SETUP

We collected our experiment profiles on a supercomputer with an x86_64 architecture and 36 cores per node. On this machine we collected performance profiles from LULESH 2.0.3 [15] and Quicksilver 1.0 [25] executions on 64 and 128 processes using 32 cores per node (2 to 16 nodes). LULESH is a proxy application that solves a Sedov blast problem and Quicksilver solves a simplified monte carlo problem. In addition, we strong-scaled Tortuga using 32, 64, 128, and 256 processes. Tortuga is a computational fluid dynamics applications provided to us by our collaborators. We use a set of data collected from AMG 1.2 [12] (a parallel algebraic multigrid solver) and Laghos for a study on variability in [23]. This data was collected on the same machine and for several applications run on 512 processes with the same configuration for almost a year. We used a subset of the data that includes six months of HPCToolkit profiles for AMG and Laghos executions.

We built each tool and application using GCC 8.3.1 and Open MPI 3.0.1. We used Score-P 7.1 to profile Tortuga and all other applications was profiled using HPCToolkit 2021.05.15.

6 PERFORMANCE EVALUATION OF CHOPPER

In this section, we evaluate the performance of some of the single execution analysis functions in Chopper.

6.1 API Performance for Single Executions

We measure the runtime of the functions by using them with a set of HPCToolkit profiles. The smallest GraphFrame is created from an AMG execution on 64 processes (2 nodes) and contains 1,893,504 rows in the DataFrame and 29,586 CCT nodes in the graph object. The biggest is created from a MILC execution on 256 processes (8
nodes) and contains 121,177,088 rows in the DataFrame and 473,348 CCT nodes in the graph object. The data points in between are from AMG executions on 128, 256, and 512 and a MILC execution on 128 processes.

Figure 7 (a) shows the runtime of `load_imbalance` and `hot_path` for each data size. The runtime of the HPCToolkit reader function, Hatchet’s `from_hpctoolkit`, is included to illustrate the total time required as reading into Hatchet is necessary to use Chopper. The results demonstrate that all the functions work efficiently for large profiles in terms of number of rows in the DataFrame. The lowest function, `load_imbalance`, takes 9.81 seconds for the smallest and 470.94 seconds for the largest data size. This increase in running time is expected due to the significant increase in size of the profiles. Figure 7 (b) shows only the memory consumption of the `from_hpctoolkit` function because the Chopper functions consume less memory than the file reading.

7 CASE STUDIES

We demonstrate the usability and flexibility of the analysis functionalities we provide in Chopper by analyzing profiles from single and multiple executions.

Finding the hot path: Hot path analysis helps explore the most time-consuming call path in the program. It may help to pinpoint potential bottlenecks. Figure 8 shows the hot path we found in a LULESH execution on 64 processes. We call the `hot_path` function and find the hot path. Chopper identified `CalcEnergyForElems` as the hot node, indicating that each of the nodes between the root and `CalcEnergyForElems` account for 50% or more of the inclusive time of their parents. Further exploration can be done by examining the children and parent of the hot node.

To visualize the hot path, we added capability to highlight the hot path in the Jupyter visualization, which simplifies analyzing and presenting the hot path. The visualization highlights the hot path by coloring the nodes and edges in red and making the nodes bigger and edges thicker than normal (Figure 8). The user can easily visualize the tree (line 4) and manipulate it interactively (e.g., selecting nodes, expanding or collapsing subtrees) for further examination and export tree state back to Python via query. The code block in Figure 8 demonstrates that Chopper makes this analysis effortless with a few lines of Python code and enables further investigation.

Detecting load imbalance: In this case study we use Quicksilver proxy application to analyze load imbalance in an execution on 128 processes. The code block in Figure 9 demonstrates how the load imbalance analysis can be performed using Chopper. We create the GraphFrame (line 1) and call the `load_imbalance` function with the time metric and verbose parameters (line 2). The DataFrame associated with the returning GraphFrame is sorted by the `time.mean` column, so that we can investigate the load imbalance of the most time-consuming CCT nodes. Then, we create a smaller DataFrame, `df_imb` by filtering out the top 50 nodes and sorting them by `time.imbalance` (line 4). Then, we focus on the top four nodes (line 5) that have the highest imbalance values since the rest of nodes do not have significant imbalance.

Figure 9 (a) shows the resulting DataFrame. The highest imbalance (4.199) occurs from `MacroscopicCrossSection:22`. The five ranks (process IDs) with the highest time value are shown in `time.ranks`. The `time.percentiles` column shows the 0th, 25th, 50th, 75th, and 100th percentiles. Using these two columns, we observe that rank 39 has the most load imbalance and spends 39.1 seconds in this function. The `time.hist` column shows the number of processes for each node in ten equal-sized bins space across the full range of time values. Additionally, users can easily investigate the parent of `MacroscopicCrossSection:22` (e.g., `df_imb.index[0].parents`) to see where it is being called. Using this to examine the source code, we observe that many load and store operations are performed in `macroscopicCrossSection` and it is called in a for loop inside of the `CollisionEvent` function. Therefore, uneven distribution of load/store operations across processes may be the cause of this imbalance.

As shown in Figure 9 (b), most processes spend between 6.296 to 9.577 seconds, while only a few spend more than 32.541 seconds. This case study shows that a detailed analysis of load imbalance at per node-granularity can be trivially accomplished using Chopper.

Analyzing correlation between metrics and CCT nodes: In this case study, we analyze the relationship between performance
```
graphframe = hatchet.GraphFrame.from_hpctoolkit("qs_profile_128")
graphframe_imbalance = graphframe.load_imbalance( verbose=True)
# sort the top 50 nodes that have the highest mean value by imbalance
df_imb = graphframe_imbalance.dataframe.head(50).sort_values("time.imbalance", ascending=False)
print(df_imb.head(4))
```

Figure 9: Demonstration of load imbalance analysis and the results of the case study. The most imbalance is caused by MacroscopicCrossSection:22. Chopper's load_imbalance function provides detailed statistics about the imbalance (a) that can be easily plotted by using Python libraries (b). We use Quicksilver execution on 128 processes.

```
correlation_matrix = gf.correlation_analysis(metrics=['time', 'cache_misses', 'branch_misses', 'instructions', 'page_faults', 'cycles'], method='spearman')
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm')
gf_corr = gf.pairwise_correlation(metric1="instructions", metric2="time", logscale=True)
plt.plot(gf_corr.dataframe["instructions"], gf_corr.dataframe["predicted"], 'o')
plt.plot(gf_corr.dataframe["predicted"], np.exp(gf_corr.dataframe["predicted"]), '--k')
```

Figure 10: Demonstration of correlation analysis performed by using Chopper. We first calculate the correlation matrix of all the performance matrix (a). Then, we investigate the relationship of instructions and time metrics at the individual CCT node level (b). We use the Tortuga execution on 1024 processes.

We manually annotated and profiled Tortuga by using Score-P.

Figure 10 (a) shows the correlation between each metric. We first examine the correlation between performance metrics by using the correlation_analysis function with the Spearman method.
datasets = glob.glob("list_of_amg_profiles")
graphframes = hatchet.GraphFrame.construct_from(datasets)
pivot_table = hatchet.Chopper.multirun_analysis(graphframes=graphframes, threshold=threshold)
pivot_table.loc[:, :].plot.bar(stacked=True)

Figure 11: Demonstration of variability analysis by using multiple executions. The figure shows the executions slowest to fastest from left to right. We create GraphFrames for slowest, average, and fastest (a) AMG and (b) Laghos runs. Then, we use the multirun_analysis function to compare CCT nodes on these multiple executions and easily create plots by using the output of the function. As shown, the variation comes from communication libraries in both cases. All executions use 512 processes and has the same configuration.

query = ["*", {"name": "MPI_File_write_all"}]
filtered_test = graphframe.filter(query)
print(filtered_test.tree())

Figure 12: Call paths of the problematic portions of the program before (left) and after (right) the optimization. The time spent in writeSingleField reduced from 7.033 to 2.088. The 1024 process count execution is used.

(line 1). We create a heatmap of correlation values by using the seaborn library in Python (line 2). Interestingly we observe that time and other metrics are not highly correlated. To investigate further, we use the pairwise_correlation function (line 4). We then create a scatter plot with a regression line by using the output of the pairwise_correlation function (line 5 and 6). As shown in Figure 10 (b), there are a few outliers CCT nodes. The time-loop node represents the main for loop that includes all the operations and functions calls on the program. The program spends relatively significant amount of time on write_data_cvnoVector and write_data_cvnoScalar although they don’t execute as many instructions. Both of these functions perform parallel IO write operations. Therefore, we observe that they perform less instructions but they have more wait time due to IO operations. This case study demonstrates that the users can easily examine correlation between different performance metrics and investigate outliers or potential issues by performing analyses at CCT node level. Chopper also enables to easily plot the results by using the Python libraries.
7.2 Comparing Multiple Executions

More advanced analysis tasks, such as studying scalability and variability, require analyzing multiple executions of the same program with different parameters. In this case, the user needs to analyze more than two CCTs. We show that Chopper can significantly simplify these analysis tasks.

Identifying performance variability: We analyze data collected in [23] that focuses on two applications, AMG and Laghos. The data was collected over a period of six months, during which the applications were executed repeatedly on a fixed number of nodes using fixed input parameters to study performance variability. In this case study, we demonstrate how we use Chopper to quickly identify the sources of variability. For both applications, we identify the runs that have fastest, slowest, and average execution time and analyze profile of these runs.

Figure 11 illustrates our analysis methodology and the resulting plots. First, we create GraphFrames for each profile (line 2) and pass them to the multirun_analysis function. Using the time metric, we apply a threshold for each of the three executions to remove the insignificant CCT nodes. Using the table that the multirun_analysis function calculates efficiency relative to the baseline execution. We filter out the CCT whose efficiency values are greater than 0.7 (line 4) and plot the results by using the resulting DataFrames (line 5). In addition to efficiency values and node names, the user can access the corresponding file and line number from the DataFrame.

The efficiency plot (Figure 13) shows the nodes that use more than 10% of the total execution time and have efficiency values lower than 0.7. endGhostCvsInterfaces perform the communication of ghost cells in the program. Therefore, the decreasing efficiency on these nodes indicates inefficient communication. spectralRadius is called in every iteration of the main for loop of the program. It calculates spectral radius of a 3-dimensional tensor and calls both MPI_Bcast and MPI_Reduce. MPI_Bcast run is a large function (772 lines of code) that includes the main for loop and many IO operations. time-loop represents the main for loop. writeSingleField includes file write operations using all the MPI processes used.

After getting this efficiency results, we decided to focus on the writeSingleField function because it is one of the functions that has significantly decreasing efficiency. We further annotated this function to identify the code block that cause this scalability issue. We identified the MPI_File_write_all function as a cause of this problem. It is a collective and blocking function that uses all the processes on the program to write to a file. Instead of using this collective and blocking function, we used the nonblocking MPI_File_iwrite function and leveraged asynchrony to optimize the function. Figure 12 demonstrates the unoptimized (left) and optimized (right) version of the corresponding call path. The time spent on writeSingleField reduced from 7.033 to 2.088 on 1024 processes. The figure also demonstrates how to easily get the corresponding call paths by using Hatchet’s query language.

This study shows that Chopper significantly simplifies this scalability analysis at per-node granularity by providing functions that which is expected due to network congestion mentioned in the previous paper [23].

The Chopper API enables the analysis of multiple executions using a single function call and presents the results in an easy-to-plot format. This is a tedious and fraught task without programmatic analysis capabilities as it requires comparing performance nodes from multiple runs simultaneously. Furthermore, to the best of our knowledge, this is the first study that uses CCT data to identify performance variability.

Identifying scalability bottlenecks: In this case study, we analyze data from a strong scaling experiment using Tortuga executions on 64, 128, 256, 512, and 1024 processes. The executions use 2, 4, 8, and 16 full nodes on the supercomputer, respectively. The efficiency at 128, 256, 512, and 1024 process counts is calculated relative to the baseline, which is the execution with 64 processes. We used the code that we manually annotated using Score-P. After getting this efficiency results, we decided to focus on the writeSingleField function because it is one of the functions that has significantly decreasing efficiency. We further annotated this function to identify the code block that cause this scalability issue. We identified the MPI_File_write_all function as a cause of this problem. It is a collective and blocking function that uses all the processes on the program to write to a file. Instead of using this collective and blocking function, we used the nonblocking MPI_File_iwrite function and leveraged asynchrony to optimize the function. Figure 12 demonstrates the unoptimized (left) and optimized (right) version of the corresponding call path. The time spent on writeSingleField reduced from 7.033 to 2.088 on 1024 processes. The figure also demonstrates how to easily get the corresponding call paths by using Hatchet’s query language.

This study shows that Chopper significantly simplifies this scalability analysis at per-node granularity by providing functions that

```python
1 datasets = glob.glob("list_of_tortuga_profiles")
2 df = hatchet.GraceFrame.construct_from(datasets)
3 df = hatchet.Cheaper.speedup_efficiency(gfs, strong=True, efficiency=True)
4 df = df.loc[df["1024"] < 0.7]
5 df.T.plot.bar()
```
can automatically unify the profile outputs and calculate efficiency. It also enables easy plotting of the results via Python libraries.

8 CONCLUSION

In this study, we proposed Chopper, a Python-based API for performance analysis, which provides programmatic analysis capabilities that simplifies the performance analysis of single and multiple executions of parallel applications.

We decided to build it on top of Hatchet to leverage its programmatic interface and visualization capabilities. We designed the API in a way that it does not have a steep learning curve so the users can quickly perform their analyses.

In this paper, we used several case studies to demonstrate how Chopper enables performing common but laborious analysis tasks by writing only a few lines of Python code. Specifically, we presented how Chopper simplifies analysis tasks for single and multiple executions such as detecting load imbalance, finding hot paths, identifying scaling bottlenecks, finding correlation between metrics and CNT nodes, and causes of performance variation. We also demonstrated some useful functionalities such as reading multiple profile data at once and unifying multiple GraphFrames. We identified potential performance problems in Tortuga and Quicksilver applications. Additionally, we identified the performance variability problem in AMG and Laghos runs. The effective capabilities that Chopper provides makes the performance analysis tasks easier to perform and significantly reduces the effort.

In the future, we plan to improve correlation analysis by adding predictive modeling capabilities to facilitate performance analysis. To further simplify the analyses and reduce the effort, we plan to support customizable plotting capabilities. Additionally, we will add support for analyzing the performance of GPU applications.

REFERENCES


