Analyzing the Performance of Applications at Exascale

Dragana Grbic
Department of Computer Science
Rice University
Houston, TX, USA
dg76@rice.edu

John Mellor-Crummey
Department of Computer Science
Rice University
Houston, TX, USA
johnmc@rice.edu

Abstract

The Linux Foundation's HPCToolkit is a measurement tool that enables profiling and tracing HPC applications on heterogeneous exascale supercomputers. Using HPCToolkit, users can measure applications at scales up to thousands of GPU-accelerated compute nodes and collect fine-grained, instruction-level measurements of CPU and GPU code. However, analyzing performance measurements from such large-scale program executions can be challenging. Performance databases for exascale executions can grow to terabytes, making manual inspection using a graphical user interface difficult and time-consuming. Methods for automated and programmatic data inspection are necessary to inspect the vast measurement data and identify characteristics of interest.

In this paper, we present a new approach to analyzing the performance of applications measured with HPCToolkit, with a focus on measurement data from large-scale executions. Our analysis framework provides access to HPCToolkit data with Python and analyzes it with the Pandas library. Analysis tasks can be automated, integrated, and saved inside Jupyter notebooks. When extracting data from persistent storage, users can employ techniques for automatically pruning code regions with low information content and sampling large-scale executions using specialized query expressions. We tested and evaluated our solution by analyzing executions with several thousands of profiles and traces.

CCS Concepts

• **Software and its engineering** → **Software organization**; • **Computing methodologies** → *Parallel computing methodologies*; Performance analysis.

ICS '25, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-1537-2/25/06
https://doi.org/10.1145/3721145.3730417

Keywords

Performance analysis, Exascale supercomputers, GPU-accelerated applications, Performance profiles, Calling context trees, Parallel execution traces, Programmatic analysis

ACM Reference Format:
Dragana Grbic and John Mellor-Crummey. 2025. Analyzing the Performance of Applications at Exascale. In *2025 International Conference on Supercomputing (ICS '25), June 08–11, 2025, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3721145.3730417

1 Introduction

Applications running on exascale platforms pose significant challenges to performance measurement tools. An aim of the US Department of Energy's Exascale Computing Project (ECP) was to build a software ecosystem for exascale supercomputers (Frontier [25, 26], Aurora [4, 5], and El Capitan [21]) whose compute nodes are accelerated with Graphic Processing Units (GPUs). With the support from ECP, Rice University extended the HPCToolkit performance tools to support the measurement and analysis of programs that run on thousands of GPU-accelerated compute nodes [1, 30]. However, the project team observed that HPCToolkit performance databases for exascale executions can be huge [1], making them difficult to analyze and interpret.

The traditional approach to analyzing the performance data collected by HPCToolkit involves using its graphical user interface (GUI). However, visual inspection using a GUI has its limits. When a user measures a program that executes on thousands of compute nodes, profiles and traces become massive. Manually analyzing large traces using a GUI is difficult. Although the gross application characteristics may be visually evident, understanding the aggregate impact of fine-grained characteristics is hard without automation. Users need support for programmatic and automated inspection of large-scale performance datasets.

This paper presents a novel approach to analyzing the performance of applications at scale, specifically those measured with HPCToolkit. We developed a framework that enables users to access HPCToolkit data with Python and analyze it with the Pandas [22] library. The framework enables users to write programs that perform the desired analysis tasks, automating the workflow for greater efficiency. One can write

programs inside Jupyter notebooks and reuse them on different datasets. Because the goal was to develop a scalable and efficient solution for processing large-scale out-of-core datasets, we implemented techniques for pruning code regions with low information content and sampling large-scale executions using specialized query expressions.

The paper makes the following contributions:

- it describes the framework and its interface for sampling and analyzing profiles and traces of large-scale executions collected by HPCToolkit,
- it explains the format in which extracted data are stored inside Pandas DataFrame tables and how users can leverage DataFrame operations to manipulate the data,
- it outlines how the interface for sampling large-scale data accommodates scalable and efficient analysis, and
- it demonstrates the framework's efficiency in analyzing large-scale executions using several case studies.

The paper is organized as follows. Section 2 provides an overview of the HPCToolkit performance tool for measuring applications at exascale. Section 3 reviews work related to the framework we developed for analyzing performance at exascale. Section 4 describes our new framework, including its architectural design, methodology for storing and sampling large-scale performance datasets, and its use of the Pandas library for manipulating and exploring extracted data. Section 5 presents several case studies conducted on the largest supercomputers, where we utilized the new framework to process and analyze performance datasets containing thousands of profiles and traces. Finally, Section 6 summarizes our findings and outlines some potential next improvements for analysis of application performance at exascale.

2 HPCToolkit Overview

HPCToolkit [1, 2, 30]—an open-source project by the Linux Foundation—is a measurement tool for profiling and tracing applications across a wide range of systems, from desktop computers to GPU-accelerated supercomputers. It captures detailed, instruction-level measurements of CPU and GPU code during executions that may involve tens of thousands of processes. HPCToolkit is designed to provide efficient, low-overhead performance measurement at various scales, enabling users to evaluate their applications with minimal disruption to their execution. One of its key advantages is that users can measure the performance of their applications without modifying the source code or recompiling; they only need to make a few adjustments to the command line used to launch their applications. This eliminates the need for time-consuming annotations of code regions to measure.

HPCToolkit collects calling context trees that contain detailed information about program execution. Calling context trees (CCTs) store information about the hierarchical structure of program execution. Each node in the tree represents a static program context (e.g., a procedure, inlined function, loop, or statement) or a dynamic context (e.g., a function call), and the sequence of contexts along a path from the root of the tree to any node represents the calling context of that node. HPCToolkit's detailed calling context trees support precise attribution of program performance.

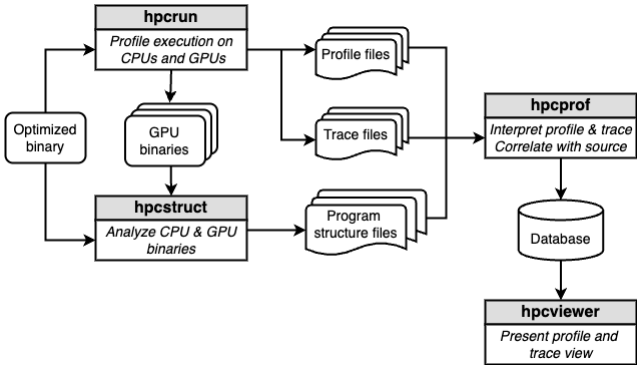

Figure 1: HPCToolkit's workflow

HPCToolkit records performance profiles that consist of calling context tree nodes annotated with metric values. By examining a calling context tree annotated with metrics, users can identify performance bottlenecks, understand program behavior, and optimize it in various ways, such as replacing expensive algorithms, avoiding unnecessary copies, or reducing load imbalance. When measuring parallel program executions, HPCToolkit records local calling context trees for each process, thread, and GPU stream. Local calling context trees annotated with metric values form parallel performance profiles. During postmortem analysis, HPCToolkit unifies local trees for each parallel execution context to construct a global calling context tree and generates a summary profile by annotating the global tree with derived metrics. Summary-derived metrics aggregate metrics across individual parallel profiles using an aggregation function: sum, average, min, max, and/or variance.

When measuring an application with HPCToolkit, users can optionally collect traces for each parallel execution context. Traces provide insight into how program behavior changes over time. A trace for an execution context is paired with a profile containing a local calling context tree. Traces are recorded as a sequence of events, each described with an ID of a calling context tree node and a timestamp.

Fig. 1 shows the traditional workflow of measuring and analyzing applications with HPCToolkit. The workflow consists of four steps:

- **measurement**: *hpcrun* measures performance as the application executes and records profiles, traces, and copies of GPU binaries observed during execution.
- **binary analysis**: *hpcstruct* analyzes CPU and GPU binaries, recovers information on how machine instructions map to source code, and records it in program structure files.
- **postmortem analysis**: *hpcprof* reads profiles, traces, and program structure files, correlates performance metrics with program structure, and generates a performance database containing sparse representations of profiles and traces.
- **presentation**: the *hpcviewer* GUI enables users to explore and analyze profiles and traces in a performance database.

This paper describes an alternative to the final step in HPCToolkit's traditional workflow, where users analyze collected data with a GUI, with a new framework for programmatic inspection using Python and the Pandas library. This new approach enables users to write programs that inspect HPCToolkit performance data and automate the analysis using modern techniques and frameworks.

3 Related Work

Before developing our own framework, we explored existing tools that enable users to programmatically inspect and analyze performance data.

Hatchet [6] is a Python library that enables users to examine performance profiles of parallel programs. It can read, process, and analyze performance data collected by various tools, including HPCToolkit, Caliper [8], and others. Hatchet's main data structure is *GraphFrame*, which consists of a *Graph* object that stores the calling context tree and a Pandas DataFrame table that attributes nodes in the calling context tree with metric values. Performance profiles stored in the DataFrame table are indexed by nodes in the *Graph* object, called a structured index. Using Hatchet, users can filter, group, and aggregate performance metrics and visualize structured profiles in a hierarchical tree view.

Thicket [9] is a Python library that enables users to analyze and compare application performance across multiple experiments. It can analyze multi-dimensional, multi-scale, multi-architecture, and multi-tool performance datasets. With Thicket, users can compare performance profiles collected using various tools and experiments, which enables them to determine the optimal configuration for large-scale application codes. A Thicket profile is a set of Hatchet profiles that represent individual application runs.

Pipit [7] is a Python library that enables users to read and analyze traces from various file formats, including HPCToolkit, Nsight [14], Projections [19], OTF2 [12], and others. It provides a consistent data structure as a Pandas DataFrame table, which can aggregate, filter, and transform events in a trace dataset. The library supports various types of data exploration, manipulation, and visualization. It also provides analysis routines for identifying performance issues in parallel executions. Pipit automates common performance analysis tasks for analyzing both single and multiple executions.

Several issues exist with these tools that make them problematic and inefficient for analyzing and processing large-scale performance datasets collected by HPCToolkit:

- They load entire datasets into memory at once. They don't provide techniques for sampling and extracting slices of large-scale data. This is infeasible when analyzing datasets containing terabytes of performance data for exascale executions. For such cases, users need tools that support reading and analysis of selective slices of data, e.g., for specific contexts in profiles or restricted time intervals in selected traces. In addition, users could benefit from pruning detailed measurements for libraries without source code or measurements of code regions that fall below a specified threshold.
- They store data in a dense format. Metrics in HPCToolkit performance profiles are sparse because different code regions are annotated with different sets of metrics. A case study by the HPCToolkit team [3] showed that in some cases, dense representations of metrics in profiles of GPU-accelerated applications on thousands of MPI ranks are over 1000× larger than sparse ones.
- They limit data exploration to either profiles or traces. Hatchet can analyze performance profiles from single application runs, Thicket can analyze performance profiles from multiple application runs, and Pipit can analyze parallel execution traces. This limits users to performing more complex analysis tasks, which include correlating information between different types of performance data. An example is reconstructing the calling context tree for a specific time interval within a trace.

NVIDIA Nsight Systems [24] also enables users to write programs to analyze execution traces collected by the tool. Their method requires converting an entire performance database collected by their tool into a SQLite format. In contrast, our tool enables users to selectively explore and read slices of large-scale data without reprocessing it all.

Google-Wide Profiling [27] periodically collects performance data from Google's data centers. This ongoing process involves sampling and measurement, continuously adding to existing databases. The data is stored in highly distributed and scalable databases optimized for fast data ingestion and

query performance. Users analyze collected data at their convenience by submitting queries that execute across the entire database. In our workflow, we focus on using specialized query expressions to sample and extract slices of data from a large-scale database. Once the slices are extracted, users can inspect them and gain insight into program behavior without examining the entire database.

We also acknowledge graph-based approaches for analyzing performance data, such as Perflow [17, 18] and Scalana [15, 16]. These tools provide an alternative methodology to automate performance analysis by converting program execution into a Program Abstraction Graph (PAG) for automated root cause identification and optimization. However, they lack interactive methods for sampling and extracting slices of large-scale data. Our focus is on providing direct, interactive access to massive performance datasets collected at exascale, emphasizing user-driven data slicing and exploration, prior to in-memory import. Additionally, these graph-based approaches consider only profiles, which limits their ability to analyze time-varying behavior.

To support programmatic analysis of large-scale performance data collected by HPCToolkit, we developed a framework that enables users to analyze profiles and traces, selectively explore and read slices of large-scale data, reduce large calling context trees using techniques for pruning code regions with low information content, and perform detailed and complex analysis tasks. The main goal of the work described in this paper was to create a scalable and efficient solution for processing and analyzing large-scale out-of-core datasets in a fast and responsive way.

4 Our Solution

We developed the *hpcanalysis* framework [28], which enables users to access HPCToolkit performance data using Python and analyze it with the Pandas library. The framework is structured hierarchically and consists of several layers. Fig. 2 shows the structure of the framework: users extract data with *Read API* and *Query API* interface and analyze it using Pandas DataFrame operations or functions implemented in the *Data Analysis* layer. When extracting data, users can employ techniques for pruning code regions with low information content or selectively exploring and reading slices of large-scale data.

hpcanalysis is designed to be memory-efficient. When users open a database for analysis, they are presented with an object that enables them to examine the data. However, data is fetched and stored in memory only upon request. The size of the extracted data on the first access depends on the type of requested data. Small data sections are extracted in their entirety, while large data sections are sampled. Sampling is performed by pruning code regions with low information

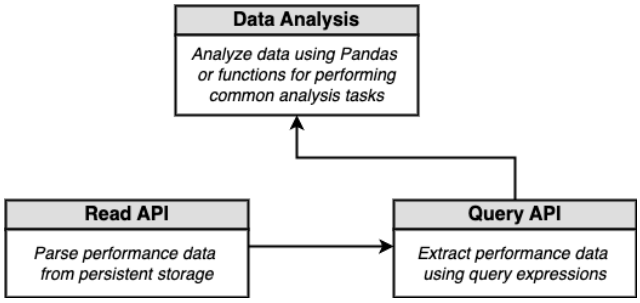

Figure 2: Structure of the *hpcanalysis* framework

content or by extracting slices of large-scale data using query expressions. Fetching upon request and extracting slices relevant to the current request ensure scalability, efficiency, and responsiveness.

4.1 Small Metadata

HPCToolkit collects various performance metrics. It can provide information about CPU time in seconds or CPU cycles. For applications that use GPUs, it collects GPU metrics that vary based on the vendor (Intel, AMD, or NVIDIA). It can also collect hardware performance counter metrics using tools such as PAPI [23] and perf [11]. When generating the summary profile, it derives summary metrics that aggregate metrics across all execution contexts. Descriptions of metrics measured across all execution contexts are stored in a separate section within the performance database. This section typically occupies a few megabytes of data, making it feasible to extract the entire section on the first access. *hpcanalysis* stores extracted metric descriptions in a separate Pandas DataFrame table, as shown in Fig. 3.

For each parallel execution in the program, HPCToolkit records a description that specifies the execution. This includes details about MPI ranks for applications that execute on several ranks, CPU threads when using multithreading, and GPU streams when offloading computations to GPUs. HPCToolkit collects distinct information for each of these contexts. MPI tasks are identified by the compute node ID and task rank ID, CPU threads are identified by thread ID, and GPU streams are identified by stream ID. This information uniquely distinguishes each performance profile and its corresponding parallel execution trace. Descriptions of profiles are stored in a separate section within the performance database. This section typically occupies a few megabytes of data, making it feasible to extract the entire section on the first access. *hpcanalysis* stores extracted profile descriptions in a separate Pandas DataFrame table, as shown in Fig. 3.

Metric descriptions and profile descriptions tables enable efficient validation of the database. Since they store entire sections of data, users can use them to ensure that proper

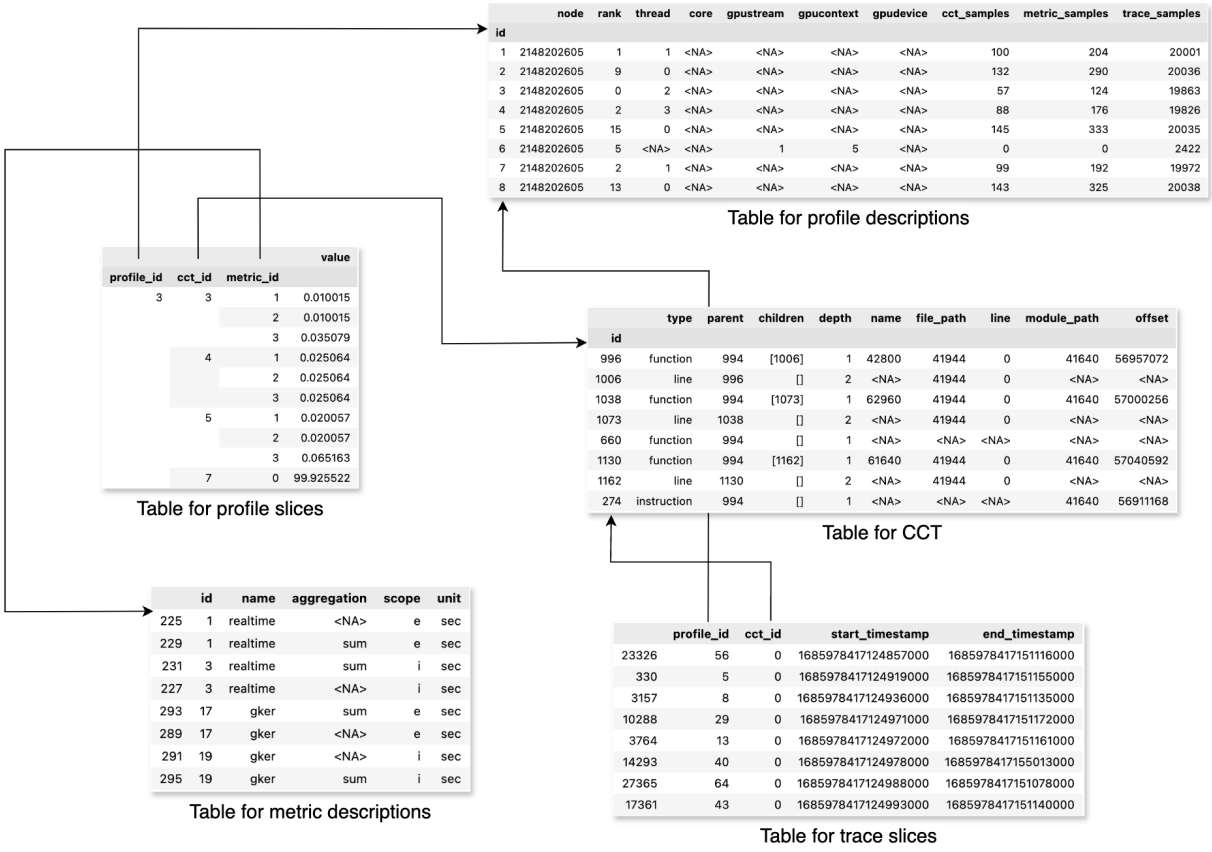

Figure 3: Architecture of the *hpcanalysis* framework. The framework organizes different types of performance data into separate DataFrame tables and correlates them to create valid representations of the collected measurements.

metrics and execution contexts are present in the database. This can be useful when making changes in the measurement methodology: users can validate that the new methodology produces valid measurements. For instance, when measuring applications that use GPUs, they can validate the existence of specific GPU metrics by examining the content of the metric descriptions table. When applications are executed on several compute nodes using multiple ranks on each node, they can validate the existence of specific nodes and specific ranks on each node. We discuss this in more detail in Section 5.1.

4.2 Large Metadata

HPCToolkit records a detailed calling context tree that provides in-depth information about program execution. It can include implementation details of library functions, nodes of various types (functions, line statements, loops, machine instructions), and many code regions with little inclusive cost. For instance, a call to an MPI routine can be recorded as a subtree that contains hundreds of nodes, and each MPI call will generate a substantial subtree. In addition, when applications are executed at larger scales, HPCToolkit gathers more unique samples in low-portability code regions. This means that if we increase the number of ranks when measuring an application, its calling context tree will grow and contain more nodes with little inclusive cost.

When an application is measured on several thousands of compute nodes, its calling context tree can become huge, making it inefficient to parse and store the entire tree in memory. To address this issue, when users open a database for analysis, *hpcanalysis* enables them to import the subtree of interest by specifying code regions they want to prune from the global calling context tree. We implemented several strategies that enable users to prune code regions with low information content. For instance, users can remove implementation details of functions in MPI, OpenMP, and CUDA libraries. They can also set a threshold to eliminate nodes with an inclusive cost less than a specified percentage of the total application time. This can significantly accelerate

reading the data because subtrees of nodes marked for removal can be skipped, and they can constitute a substantial portion of the entire tree stored in the database. Users can also choose to remove nodes that are not functions (line statements, loops, machine instructions) or instructions in a binary where mappings to source lines have been stripped. We discuss this in more detail in Section 5.2.

hpcanalysis stores the extracted calling context tree in a separate Pandas DataFrame table, as shown in Fig. 3. The hierarchical structure of the tree is represented using the *parent* and *children* fields. *hpcanalysis* uses them to visualize the tree in a hierarchical tree view. Long strings are stored in separate tables: columns *file_path*, *module_path*, and *name* store the index of string representation from the string table.

Metadata tables enable users to sample and extract slices of large-scale performance data. Users may want to retrieve performance data for a specific range of MPI ranks, CPU threads, or GPU streams, select every 100th profile or trace, extract trace lines for particular time intervals, among other requests. This sampling is performed using query expressions, outlined in Section 4.4. Users submit queries that specify desired slices of profiles or traces, queries are associated with unique IDs stored in metadata tables, and IDs are then used to access and extract selected slices from the file.

4.3 Large Performance Data

For each parallel execution context, HPCToolkit records a local calling context tree that contains code regions executed within that context. Once nodes in a local calling context tree are annotated with metrics, they form a performance profile: profiles are calling context trees annotated with metric values. HPCToolkit performance profiles are sparse in two ways: a set of contexts present in different profiles and a set of metrics present for different contexts. Context sparsity exists because local calling context trees for individual profiles may differ. For instance, threads may have different local calling context trees because they may execute different code. Metrics sparsity exists because different code regions are annotated with different metrics: CPU code is annotated with CPU metrics, while GPU code is annotated with GPU metrics.

During postmortem analysis, *hpcprof* constructs the global calling context tree outlined in Section 4.2 and generates a summary profile. Global calling context tree represents a union of local trees for each parallel execution context. Global calling context tree is annotated with summary profile metrics that aggregate metrics across individual parallel profiles. Given that profiles for executions performed on several thousands of compute nodes are huge, users need techniques for sampling. *hpcanalysis* supports the following strategies for sampling profiles.

- **Sampling profiles by execution context.** Extract profiles for specific execution contexts. A query might specify a range of MPI ranks, CPU threads, or GPU streams.
- **Sampling calling contexts in profiles.** Extract values for specific calling contexts for each selected profile. A query specifies nodes or subtrees of interest (discussed in Section 4.2).
- **Sampling metrics in profiles.** Extract values for specific metrics for calling context tree nodes of interest within or across profiles.

HPCToolkit organizes profiles into an array where profiles are indexed by their ID and sorts CCT nodes and metrics within individual profiles by their ID. This file format enables *hpcanalysis* to perform sampling efficiently: query for execution context is mapped to a list of profile IDs to access specific profiles from the file, and we use binary search to access specific values within selected profiles. In addition, *hpcanalysis* employs parallelism when parsing data for individual profiles. The number of parallel tasks it spawns is proportional to the number of extracted profiles. *hpcanalysis* stores extracted slices of profiles in a separate Pandas DataFrame table, the format of which is shown in Fig. 3. This format accommodates the sparse format of profiles: both the CCT node ID and the metric ID are stored inside the DataFrame index. Storing them as separate columns, as Hatchet does, would store empty cells for sparse elements.

When measuring applications with HPCToolkit, users can also enable tracing and collect traces for each parallel execution context. Individual traces are represented as a sequence of events, each described with a CCT node ID and a timestamp. Given that traces for executions performed on several thousands of compute nodes are huge, users need techniques for sampling. *hpcanalysis* supports the following strategies for sampling traces.

- **Sampling traces by execution context.** Extract traces for specific execution contexts. A query might specify a range of MPI ranks, CPU threads, or GPU streams.
- **Sampling time intervals in traces.** Extract events for specific time intervals for each selected trace. User provides time intervals of interest.

HPCToolkit organizes traces into an array where traces are indexed by their ID and sorts events within individual traces by their timestamp. This file format enables *hpcanalysis* to perform sampling efficiently: query for execution context is mapped to a list of trace IDs to access specific traces from the file, and we use binary search to access specific events within selected traces. As when extracting profiles, *hpcanalysis* employs parallelism when parsing data for individual traces.

Table 1: Query/Read API Interface

Query API method	Arguments	Read API method	Arguments
query_metric_descriptions	query for metrics	**read_metric_descriptions**	none - entire section of data is fetched
query_profile_descriptions	query for profiles	**read_profile_descriptions**	none - entire section of data is fetched
query_cct	query for CCT	**read_cct**	none - entire section of data is fetched
query_profile_slices	queries for profiles, CCT, and metrics	**read_profile_slices**	argument that specifies slices of profiles with profile IDs, CCT IDs, and metric IDs
query_trace_slices	query for profiles and time intervals	**read_trace_slices**	argument that specifies slices of traces with trace IDs and time intervals

hpcanalysis stores extracted slices of traces in a separate Pandas DataFrame, as shown in Fig. 3.

4.4 Interface for Extracting Data and Query Expressions

Users request specific types of performance data using *Query API* interface, a high-level API that enables requesting data using query expressions. *Read API* is a low-level API that parses raw data from storage and is called by *Query API* when a request is made for data that was not previously parsed and stored in memory. Extracted data are stored in Pandas DataFrame tables, which are initially empty. Depending on the type of requested data, different subsets are extracted and stored in memory. Table 1 specifies the *Read API* and *Query API* interface. When the user requests metadata (metric descriptions, profile descriptions, calling context tree), it provides a single query that specifies the subset of interest, and, on the first request, *Read API* will fetch the entire section. When requesting the calling context tree, specific nodes/subtrees will be excluded, which is configured by the user when it opens the database and specifies the subtree of interest. When the user requests profiles or traces, it provides several arguments/queries that specify slices of interest. When a request is made for a profile or trace with an execution context that was not previously parsed and stored in memory, *Read API* will access the requested slices in the file and perform sampling as specified, such as sampling by calling context and set of metrics for profiles, and sampling by time intervals for traces. After the data is extracted and stored in memory, slices specified by queries are presented to the user.

Listing 1 shows how users can request metadata. Queries for calling context tree extract specific call paths from the tree. Call path queries can specify different types of nodes: functions, line statements, loops, and machine instructions. They can also use wildcards for call path fragments or node attributes, such as function names. This simplifies complex queries, such as extracting all calling contexts that invoke GPU operations. Queries for metric descriptions extract descriptions of metrics with a specific name and/or scope. The two most common types of scope are inclusive and exclusive scope. The inclusive cost of a node refers to the total cost of the node along with all its descendants in its subtree.

```python
import hpcanalysis

query_api = hpcanalysis.open_db("path/to/database")._query_api

# extract CPU time metric with inclusive and exclusive scope
query_api.query_metric_descriptions("cputime_(i,e)")

# extract profiles with rank ID 0 and thread IDs 1,2
query_api.query_profile_descriptions("rank(0).thread(1-2)")

# extract specific call path from the calling context tree
query_api.query_cct("function(foo).line(foo.c:12)")
```

Listing 1: Extracting metadata using queries

```python
import hpcanalysis

query_api = hpcanalysis.open_db("path/to/database")._query_api

# extract slices of profiles for rank ID 0 and thread IDs
    1,5,6,7, for MPI functions, and for inclusive CPU time
query_api.query_profile_slices(
    "rank(0).thread(1,5-7)",
    "function(MPI_*)",
    "cputime_(i)",
)

# extract slices of traces for rank ID  0 and thread IDs
    1,5,6,7, and for a specific time interval
query_api.query_trace_slices(
    "rank(0).thread(1,5-7)",
    (1713838621,1713910621),
)
```

Listing 2: Extracting performance data using queries

The exclusive cost of a node refers to the cost of the node itself, excluding the cost of its subtree. Queries for profile descriptions extract descriptions of profiles with a specific execution context. Execution contexts can be specified with a range of values for MPI rank IDs, CPU thread IDs, or GPU stream IDs. Wildcards can be used in both queries for metrics and queries for profiles.

Listing 2 shows how users can request profiles and traces. When requesting profiles, the user provides a query for execution context, a query for calling context, and a query for metrics. When requesting traces, the user provides a query for execution context and a time interval.

In Fig. 3, we presented the overall architecture of the *hpcanalysis* framework and illustrated the correlation between different types of performance data. Metadata tables include metric descriptions, profile descriptions, and the calling context tree. They store entire sections of data, with the exception of the global calling context tree, which can be pruned

using heuristics declared when opening the database. Metadata tables do not reference other tables; all necessary data is contained within them, with the exception of long strings, which are stored in separate string tables. In contrast, tables for slices of profiles and traces store sampled sections of data. They contain slices of data that the user requests over time, utilizing specialized query expressions. Tables for profiles and traces provide context for the collected metadata, associating them with measured metrics or trace events. These tables contain IDs that link back to the metadata tables, enabling them to be referenced and connected with appropriate values. This architecture facilitates efficient querying of performance data, as queries that request slices of profiles or traces can be easily mapped to the IDs stored in metadata tables. Once these IDs are retrieved, they can be used to either extract slices from the performance files, if they were not previously requested and stored in memory, or to retrieve them from memory if they had been stored earlier. After the slices are extracted and stored in memory, they are linked with metadata, creating a clear reference between the metadata and the collected metric values or trace events.

4.5 Performing Common Analysis Tasks

After users request data using the *Query API* interface, they are presented with a DataFrame table containing the result. They can comprehend program behavior and performance using DataFrame operations, such as filtering, pruning, and aggregating. However, some tasks are performed more frequently and can benefit from automation. Such tasks are creating flat profiles, detecting load balance, detecting GPU idleness, visualizing calling context tree, etc. Therefore, we implemented an additional *Data Analysis* layer atop *Query API* that exposes several functions that automate the execution of common analysis tasks. They internally use the *Query API* interface to extract relevant subsets and perform additional transformations, such as mapping with metadata tables or grouping by execution context.

One utility in *Data Analysis* layer is *hpcreport*, which automatically analyzes the summary profile of program execution and generates a high-level overview of its behavior. *hpcreport* breaks down time spent in the program into three main categories: MPI, OpenMP, and GPU execution. It also breaks down time into several subcategories, such as timings for specific subsets of MPI operations, OpenMP, and GPU operations. For instance, for GPU operations, it reports timings for kernel execution, memory allocation, memory deallocation, memory setting, explicit data copy, implicit data copy, and synchronization. One of the strengths of *hpcreport* is that it can automatically detect whether a program execution is compute bound, memory bound, communication bound, or I/O bound. Table 2 shows an example of *hpcreport* output.

Table 2: *hpcreport* output

Major	Minor	Time (sec)	Percentage (%)
CPU	CPU Total	1.18e+04	100.00
	OpenMP Idle	8.81e+03	74.55
	MPI Collective	8.32e+01	0.70
	MPI Point-to-Point	2.62e+01	0.22
	OpenMP Wait	1.48e+01	0.12
	MPI Environmental Inquiry Functions and Profiling	1.80e-01	0.00
GPU	GPU Total	6.91e+02	100.00
	GPU Kernel Execution	5.95e+02	86.04
	GPU Explicit Data Copy	9.65e+01	13.96

While the functionality of *hpcreport* is similar to Intel VTune Application Performance Snapshot (APS) [13], *hpcreport* produces summaries from detailed information collected at runtime, whereas APS only collects summaries. This means that if the summary from *hpcreport* is concerning, one can write specialized queries to look deeper into areas of concern; with APS, one cannot.

5 Case Studies

This section describes several case studies that showcase analysis tasks using *hpcanalysis* framework. We evaluated the efficiency and scalability of processing large-scale out-of-core datasets by analyzing executions performed on thousands of compute nodes.

5.1 Validating Performance Metadata

For the first several case studies, we analyzed the performance data for a large-scale execution of LAMMPS [29]—a molecular dynamics code developed by Sandia National Laboratories—measured on Frontier. The execution contains measurement data for 8,192 compute nodes, with 8 MPI ranks per node, yielding a total of 65,536 ranks. The execution did not employ multithreading, so each rank operated with a single CPU thread. The total size of the file containing sparse profiles was nearly 100 gigabytes. The entire performance database contained approximately 4TB of data, most of which represented parallel execution traces. Analyzing such large-scale performance data enabled us to evaluate the efficiency of *hpcanalysis* at scale.

The first step we took for the LAMMPS experiment was to conduct basic validations of the database. As discussed in Section 4.1, tables that store metric descriptions and profile descriptions are typically small, making it feasible to extract them in entirety upon the first access. Once extracted, we can use them to verify the accuracy of collected metadata. Listing 3 shows the code we used to ensure the correctness of the execution contexts recorded in the LAMMPS database. We verified that the database contains measurements for the proper number of compute nodes, ranks on each node, cores, and threads.

```python
import hpcanalysis

hpc_api = hpcanalysis.open_db("path/to/database")
profs = hpc_api._query_api.query_profile_descriptions("rank")

# there are in total 65,536 execution profiles
assert len(profs)==65536
# execution is performed on 8,192 compute nodes
assert len(profs["node"].unique())==8192

for node in profs["node"].unique().tolist():
    # each compute node has 8 ranks executing on it
    assert len(profs[profs["node"]==node]["rank"].unique())==8
    # each rank is executed on a separate core
    assert len(profs[profs["node"]==node]["core"].unique())==8
    # each rank has a single CPU thread executing on it
    assert len(profs[profs["node"]==node]["thread"].unique())==1
```

Listing 3: Verifying performance metadata in the LAMMPS experiment

5.2 Pruning Code Regions

We continued examining the LAMMPS experiment by analyzing its calling context tree. Listing 4 shows the interface we implemented for specifying strategies for pruning code regions from large calling context trees. The *CCTReduction* object from Listing 4 can receive an arbitrary set of user-defined filters for pruning the calling context tree. We defined some strategies for pruning the tree, but users can define their own filters as long as they implement the appropriate interface. One strategy we implemented is to set a threshold that will eliminate all nodes with inclusive time below a specific percentage of the total application time, along with their entire subtrees. Since the LAMMPS experiment was conducted on 65,536 ranks and the HPCToolkit collects more unique samples in low-portability code regions at larger scales, we anticipate that the resulting tree will contain numerous code regions with little inclusive cost. Another strategy is to remove implementation details of commonly used libraries, such as MPI. Even in smaller-scale experiments, we encountered subtrees of MPI functions that contained several hundreds of nodes. Since each call to a library routine generates a substantial subtree, a significant portion of the database can consist of the implementation details of library functions. Users can also choose to eliminate nodes that are not functions (line statements, loops, machine instructions). We tested various combinations of these pruning techniques and analyzed the trees they produced.

Table 3 shows the size of the calling context tree for the LAMMPS experiment after pruning various code regions. The entire tree contains 1,006,440 nodes. Interpreting performance using a tree that contains more than a million nodes is difficult and time-consuming. We experimented with setting different thresholds to remove regions with little inclusive cost. We evaluated pruning regions that accounted for less than 0.0001%, 0.001%, 0.01%, 0.1%, and 1% of the total application time. For each of these thresholds, we also examined

```python
import hpcanalysis

cct_reduction = CCTReduction()

# remove nodes that are not functions
cct_reduction.add_reduction(FunctionReduction())

# remove nodes with little inclusive cost
cct_reduction.add_reduction(TimeReduction(percentage_threshold
    =1))

# remove implementation details of MPI
cct_reduction.add_reduction(MPIReduction())

# remove implementation details of OpenMP
cct_reduction.add_reduction(OpenMPReduction())

hpc_api = hpcanalysis.open_db("path/to/database", cct_reduction=
    cct_reduction)
```

Listing 4: Specifying strategies for pruning the calling context tree

Table 3: Size of the calling context tree in the LAMMPS experiment with different pruning techniques

Threshold	None	0.0001%	0.001%	0.01%	0.1%	1%
Nodes in the entire calling context tree	1,006,440	15,981	5,222	2,047	618	165
Remove implementation details of MPI library	791,035	10,270	3,059	1,102	310	98
Remove nodes that are not functions	248,022	6,896	2,557	1,056	372	112
Remove implementation details of MPI and nodes that are not functions	196,851	4,577	1,571	596	200	63

Table 4: Percentage of specific code regions in the LAMMPS experiment with different pruning thresholds

Threshold	None	0.0001%	0.001%	0.01%	0.1%	1%
Implementation details of MPI	21%	36%	41%	46%	50%	41%
Nodes that are not functions	75%	57%	51%	48%	40%	32%
Implementation details of MPI and nodes that are not functions	80%	71%	70%	71%	68%	62%

the effects of removing implementation details of MPI and eliminating nodes that are not functions. Table 4 shows the percentages of MPI implementation details and nodes that are not functions within the calling context tree for each pruning threshold.

An interesting finding is that if we apply a 0.0001% threshold, which is relatively low, we reduce the tree size from one million nodes to only fifteen thousand. With larger thresholds, we reduce the size even further, reaching only 165 nodes for the 1% threshold. These thresholds enable users to focus their analysis tasks on the most time-consuming regions. When we tested removing MPI implementation details, we observed that 20-50% of the tree can consist of subtrees rooted in MPI functions. Eliminating these regions can significantly accelerate the analysis workflow by reducing loading

Figure 4: Pruned calling context tree for the LAMMPS experiment, showcasing the most time-consuming functions

times. When inspecting performance data from multiple programs, we found that nodes that are not functions account for 30-70% of the calling context tree. Pruning nodes below the function level accelerates high-level analysis tasks.

Removing both MPI implementation details and nodes that are not functions for each pruning threshold for the LAMMPS experiment reduced the tree size from 1,006,440 nodes to 4,577 for the 0.0001% threshold, 1,571 for the 0.001% threshold, 596 for the 0.01% threshold, 200 for the 0.1% threshold, and 63 for the 1% threshold. These results indicate that a significant portion of the database consists of regions with low information content. Analyzing performance for such regions adds unnecessary complexity when trying to understand the program behavior over time or identify opportunities for improving performance. Pruning strategies such as those we implemented help thin the representation and focus analysis on important program regions.

We visualized the calling context tree after applying a 1% threshold and removing MPI implementation details and nodes that are not functions. We used the *visualize_cct* function we implemented in the *Data Analysis* layer, which displays a hierarchical view of a calling context tree. Fig. 4 shows the pruned calling context tree for the summary profile of the LAMMPS execution. Users can expand and collapse nodes and display metrics from the summary profile. In Fig. 4, we expanded the nodes that consume the most time and displayed the total execution time for each. This provides an overview of the most time-consuming regions, which we can use to guide further exploration of program performance.

```
fn = [
    'LAMMPS_NS::VerletKokkos::run',
    'LAMMPS_NS::CommKokkos::forward_comm_device',
    'MPI_Send',
    'MPI_Wait',
    'LAMMPS_NS::NeighborKokkos::build_kokkos',
    'LAMMPS_NS::CommKokkos::exchange_device',
    'LAMMPS_NS::CommKokkos::borders',
    'LAMMPS_NS::ModifyKokkos::final_integrate',
    'LAMMPS_NS::LAMMPS::LAMMPS',
]

load_bal = hpc_api.load_balance

load_bal(fn, "rank(0-65536:4096)") # extract every 4096th rank
load_bal(fn, "rank(0-65536:512)")  # extract every 512th rank
load_bal(fn, "rank(0-65536:64)")   # extract every 64th rank
load_bal(fn, "rank(0-65536:8)")    # extract every 8th rank
load_bal(fn, "rank(0-65536:1)")    # extract every rank
```

Listing 5: Detecting load balance in the LAMMPS experiment by extracting different numbers of ranks

5.3 Sampling Performance Profiles

After pruning the calling context tree and identifying the most time-consuming regions, we analyzed their distribution across MPI ranks. This analysis enables us to determine whether the application workload is evenly distributed or if certain ranks are performing more work than others. We implemented the *load_balance* function in the *Data Analysis* layer, which calculates the load balance of a specific calling context, such as an MPI routine, across selected execution contexts, such as MPI ranks. It estimates the load balance by dividing the average execution time of the routine across the selected ranks by the maximum time it spent across those ranks. A ratio close to 1 indicates that the routine is well-balanced, while a lower ratio signifies a greater imbalance. When the user selects a range of ranks to examine, it can set a step value to sample the range and estimate results on a smaller subset. For instance, when analyzing an execution on 100,000 ranks, it can set a step of 100 to extract every 100th rank and examine 1,000 ranks instead of all 100,000. We will also add support for the probability of examining a rank so that one can examine a random subset rather than a strided subset. This approach enables users to sacrifice some accuracy for faster analysis.

For the LAMMPS experiment conducted on Frontier, we selected the most time-consuming functions from Fig. 4 and analyzed their load balance across MPI ranks. We evaluated the load balance over the entire range of 65,536 ranks using five different steps: we tested extracting 65,536/4,096 = 16, 65,536/512 = 128, 65,536/64 = 1,024, 65,536/8 = 8,192, and all 65,536 ranks. Listing 5 shows the code we used to sample ranks and examine the load balance among them. Initially, we estimated the load balance when extracting only 16 ranks. We observed that most functions were well-balanced, except for the MPI functions *MPI_Send* and *MPI_Wait*, which had a balance ratio of around 0.5, indicating an uneven distribution.

Table 5: Load balance estimates for the most time-consuming functions in the LAMMPS experiment when extracting different numbers of ranks. Notice that the cost of MPI routines is imbalanced.

	VerletKokkos::run	CommKokkos::forward_comm_device	LAMMPS::LAMMPS	ModifyKokkos::final_integrate	NeighborKokkos::build_kokkos	CommKokkos::borders	CommKokkos::exchange_device	MPI_Send	MPI_Wait
16 ranks	0.99	0.99	0.98	0.99	0.97	0.93	0.95	**0.52**	**0.52**
128 ranks	0.99	0.99	0.98	0.98	0.96	0.91	0.91	**0.52**	**0.52**
1,024 ranks	0.99	0.99	0.98	0.98	0.96	0.90	0.91	**0.51**	**0.51**
8,192 ranks	0.99	0.98	0.98	0.97	0.94	0.87	0.88	**0.51**	**0.50**
65,536 ranks	0.99	0.98	0.98	0.97	0.94	0.86	0.85	**0.46**	**0.44**

Table 6: Times in seconds for pruning and extracting pruned calling context trees for different numbers of MPI ranks in the LAMMPS experiment

Number of MPI ranks	16	128	1,024	8,192	65,536
Serial	0.21	1.05	7.87	67.80	565.80
8 parallel tasks	2.57	0.31	2.03	14.10	124.35
16 parallel tasks	4.29	0.27	1.54	10.59	92.89
64 parallel tasks	0.14	6.78	1.70	12.15	103.71

Table 7: Times in seconds for extracting entire calling context trees for different numbers of MPI ranks in the LAMMPS experiment

Number of MPI ranks	16	128	1,024	8,192	65,536
Serial	18.58	85.63	676.38	> hours	> hours
8 parallel tasks	12.21	17.90	109.99	843.87	> hours
16 parallel tasks	12.04	13.11	71.20	537.43	> hours
64 parallel tasks	11.88	15.52	46.56	345.55	> hours

Subsequently, we tested extracting 128, 1,024, 8,192, and finally, all 65,536 ranks. Table 5 shows the balance estimates for each number of ranks extracted. The table represents balance estimates for inclusive function costs. We found that the results were quite similar for each number of ranks extracted: most functions were well-balanced, except the MPI functions. These findings suggest an irregular distribution of communication time across ranks, which can be further investigated for communication anomalies in the program. Notably, we were able to detect this imbalance even when extracting only 16 ranks.

We evaluated the performance of sampling the LAMMPS execution when analyzing the load balance. Table 6 shows the performance, measured in seconds, for each experiment when extracting different numbers of MPI ranks. We tested using a serial solution and spawning different numbers of parallel tasks. We can employ parallelism when parsing data for individual profiles, as they are stored in distinct sections within the performance file, indexed by profile ID. Extracting all 65,536 ranks with a serial solution took 565.80 seconds, which is nearly 10 minutes. When we spawned 8, 16, and 64 parallel tasks, the extraction time for all 65,536 ranks dropped to 124.35, 92.89, and 103.71 seconds, respectively. However, results from Table 5 show that we can achieve highly accurate load balance estimates even when extracting a smaller number of ranks. Extracting 1,024 ranks took less than 10 seconds with a serial solution, and when employing parallelism, it dropped to only 1-2 seconds. Nevertheless, the estimates obtained when extracting 1,024 ranks were almost the same as those for all 65,536 ranks.

When we sampled different numbers of MPI ranks to analyze the load balance, we extracted parallel profiles for calling context trees pruned using techniques discussed in Section 5.2. *hpcanalysis* prunes the global calling context tree and then attempts to extract the same subset of tree nodes for each selected parallel profile.[1] Table 7 shows the time when extracting different numbers of ranks without pruning the calling context tree. This is much slower than extracting data for pruned trees, as reported in Table 6. As detailed in Section 4.3, *hpcanalysis* employs techniques for efficient access and extraction of slices of profiles from the database. Profiles are organized into an array within the performance file, enabling us to index them by their ID. CCT nodes are sorted by their ID within individual profiles, enabling us to access them through a binary search rather than iterating through entire profiles. If we don't prune the calling context tree, we could end up sampling parallel profiles with local trees that contain up to 1,006,440 nodes, which is the size of the entire global tree. When extracting all 65,536 ranks, we would need to parse $65{,}536 \times 1{,}006{,}440$ values, assuming that the local trees for individual ranks are as large as the global calling context tree. As shown in Table 7, we were unable to extract/parse data for all 65,536 ranks, even after several hours of processing. When we attempted to extract smaller numbers of ranks, such as 1,024 and 8,192, the loading times were several minutes, which is significantly longer than the few seconds it took when we pruned the tree. For large-scale experiments, one might not even have enough physical memory on a single compute node to extract complete calling context trees for all MPI ranks.

[1] Some profiles may not contain some of the nodes of interest.

The significant decrease in loading time—from several hours when extracting all ranks and the entire calling context trees to only a few seconds when sampling ranks, pruning the calling context trees, and utilizing parallelism—demonstrates that *hpcanalysis* enables users to efficiently explore and selectively read slices of large-scale performance data. Users can conduct their analysis tasks in a fast and responsive way without having to reprocess the entire database. In most cases, a modest number of samples should yield results of sufficient accuracy. If concerned about the accuracy of a set of samples, one can extrapolate from the sampled set and compare it with the summary profile. If the difference between the estimate of the summary profile and the actual summary profile is too large, then sample additional profiles.

In the LAMMPS experiment, we investigated the load balance of the most time-consuming functions identified while pruning the calling context tree and highlighted the MPI load imbalance as the most significant finding. Users can, however, examine any metric for any context, such as GPU kernels, by modifying the query expressions used when extracting slices of performance data. Specifically, in the LAMMPS experiment, when we analyzed the performance of GPU kernels, we found that the GPU work was well-balanced and not noteworthy.

5.4 Detecting Communication Anomalies

While analyzing the load balance of the most time-consuming regions in the LAMMPS experiment, we detected an irregular distribution of communication time across MPI ranks. This prompted us to investigate these communication anomalies further. We selected all MPI functions using query *"function(MPI_*)"* and analyzed their values across 1,024 ranks. For each rank in this range, we computed the total time spent in each MPI routine. Subsequently, we calculated the minimum and maximum total time across ranks for each MPI routine. We observed the highest variance between the minimum and maximum total time in *MPI_Bcast*, where the minimum was 0.05 seconds, and the maximum was 101.81 seconds. This indicates that some rank spent 0.05 seconds in *MPI_Bcast* and some spent 101.81 seconds. When we tested extracting 8,192 ranks to determine the variance for a larger set of ranks, the minimum total time was 0.002 seconds, and the maximum was 102.22 seconds. This didn't significantly differ from the results obtained with the initial sample of 1,024 ranks.

After observing a significant variance between the minimum and maximum total time spent in *MPI_Bcast* across ranks, we plotted a histogram of its execution times across ranks. When analyzing the minimum and maximum total times, we reduced the global calling context tree using a 1% threshold. This resulted in only one instance of *MPI_Bcast* appearing in the resulting tree. We selected this instance and plotted a histogram of its execution times across 1,024 ranks. Fig. 5 shows the histogram. We noted an uneven distribution of execution times; some ranks reported values of less than one second, while others reported times close to two minutes. Most values were either under one second or nearly two minutes. We also plotted a histogram when extracting 8,192 ranks, which showed a similar distribution as when extracting 1,024 ranks.

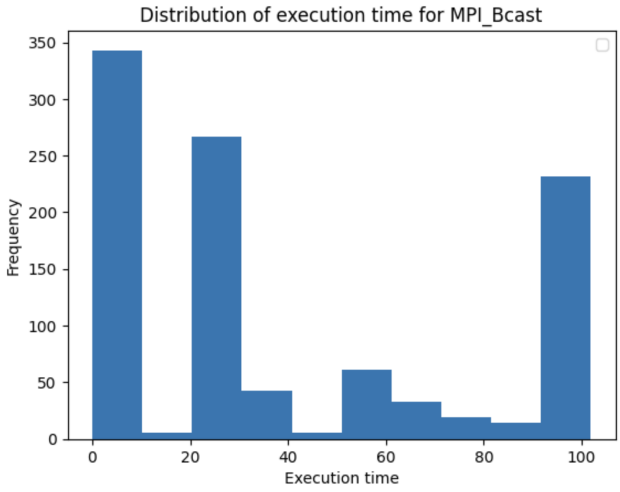

Figure 5: Distribution of MPI_Bcast execution time across 1,024 ranks in the LAMMPS experiment

We tested applying lower thresholds when pruning the calling context tree to examine the balance of additional instances of *MPI_Bcast* that might exist in the entire tree. We decreased the threshold from 1% to 0.0001%, resulting in a tree that contained two additional instances of *MPI_Bcast*. The first instance had a minimum total time of 0.002 seconds and a maximum total time of 3.62 seconds. The second instance had a minimum total time of 0.002 seconds and a maximum total time of 6.08 seconds. When we plotted histograms of their execution times across 1,024 ranks, we observed that the values were relatively evenly distributed. This indicates that the critical instance of *MPI_Bcast*, which exhibited communication imbalance in the program, was detected when applying the 1% threshold. When we used this threshold, we identified the *MPI_Bcast* call that exhibited a wide spread of execution times across ranks, ranging from less than a second to several minutes, signaling a significant communication imbalance in the program possibly caused by workload imbalance.

After identifying the irregular distribution of communication time with *hpcanalysis*, we visualized parallel execution traces using HPCToolkit's *hpcviewer* graphical user interface. Fig. 6 shows the trace view of parallel execution traces

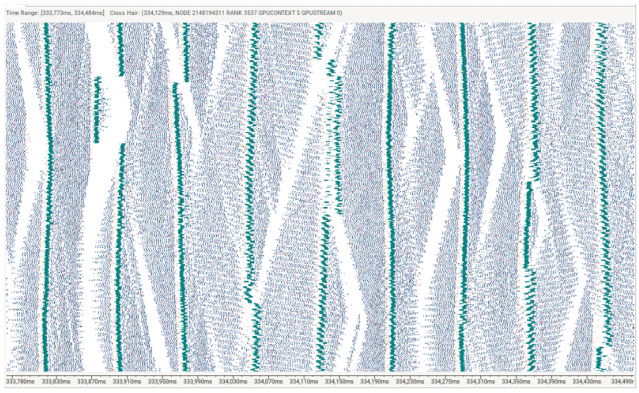

Figure 6: LAMMPS trace view, showcasing strange network delays and patterns of propagating idleness

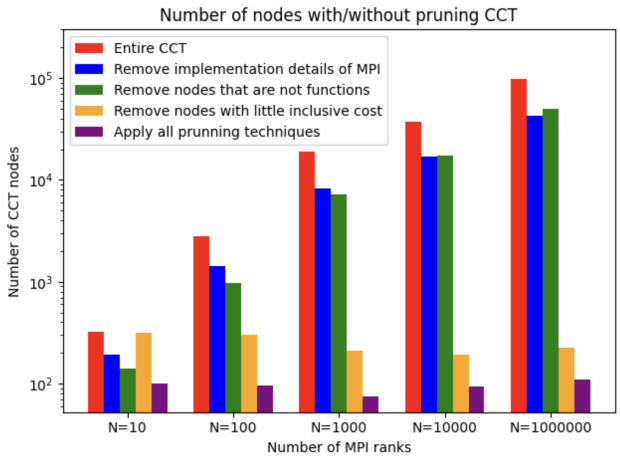

Figure 7: Reducing calling context trees in the scaling study of AMG benchmark

that perform GPU work. We observed unusual delays and peculiar patterns of propagating idleness. One theory is that these delays may stem from congestion in the Slingshot [10] network on Frontier. Given that LAMMPS was measured on a large number of MPI ranks (65,536) with short intervals of computation between communications, it may have stressed the network. To further investigate the underlying causes of these propagating delays, we would need to gather more information than what is currently measured with HPCToolkit. For instance, we could measure the network counters for the Slingshot interconnect during each call to an MPI function and analyze how these values change as we increase the number of execution ranks. By intercepting each MPI routine call, we can collect various counters and identify those that closely correlate with increased congestion. This would provide us with deeper insights into the network behavior and help us understand the reasons behind the observed delays and communication anomalies.

When we used HPCToolkit's graphical user interface, we parsed the entire database and visualized numerous traces to identify strange network delays that were difficult to further examine through a visual inspection. With *hpcanalysis*, we examined these communication anomalies by pruning the calling context tree from one million nodes down to about a hundred, which detected the critical instance of *MPI_Bcast*. After sampling only 1,024 ranks from a total set of 65,536, we noted a significant variance in *MPI_Bcast* execution times across ranks. We could investigate Slingshot network counters and their correlation with *MPI_Bcast* times to analyze network congestion and identify patterns of propagated idleness that we observed on Frontier.

5.5 Experiments on Aurora

We conducted several experiments on the Aurora supercomputer to further investigate the size of calling context trees when measuring large-scale executions and how these trees grow as the number of ranks increases. We performed a scaling study of AMG [20]—an algebraic multigrid benchmark developed by the Lawrence Livermore National Laboratory—measured on Aurora. We executed the benchmark using 10, 100, 1,000, 10,000, and 100,000 ranks. For each test, we allocated 1,000 compute nodes and evenly distributed ranks among them. These experiments enabled us to assess the correlation between the number of ranks used for execution and the size of the resulting tree. In addition, we also evaluated the effectiveness of pruning techniques discussed in Section 5.2, where we reduced the calling context tree for the LAMMPS experiment.

Fig. 7 shows the size of the calling context tree for each experiment. The red bars represent the size of the tree without pruning any code regions. Notice that the size of the tree grows as the number of ranks increases. This happens because HPCToolkit captures more infrequent instructions in low-probability code regions at larger scales, resulting in more tree nodes that have low inclusive cost. Table 8 shows the exact number of CCT nodes for each experiment. When we executed the benchmark on 10 ranks, the size of the tree was 326 nodes. When we performed the same execution using 100,000 ranks, the size increased to 97,833 nodes. We evaluated the impact of removing MPI implementation details (blue bars) and nodes that are not functions (green bars). Both produced trees of similar sizes. Table 9 shows the percentage of these code regions within the entire calling context tree. For the experiments conducted on Aurora, on average, 57% of code regions were attributed to MPI implementation details, and 52% were nodes that are not functions (line statements, loops, machine instructions).

Table 8: Size of the calling context tree in scaling study of AMG benchmark with different pruning techniques

Number of MPI ranks	10	100	1,000	10,000	100,000
Nodes in the entire calling context tree	326	2,792	18,932	37,567	97,833
Remove implementation details of MPI library	139	979	7,255	17,273	49,943
Remove nodes that are not functions	190	1,411	8,330	16,757	42,545
Remove nodes with <1% of total application time	315	303	210	191	224
Apply all pruning techniques	101	96	75	93	110

Table 9: Percentage of specific code regions in the scaling study of AMG benchmark

Number of MPI ranks	10	100	1,000	10,000	100,000	Average
Implementation details of MPI	57%	65%	62%	54%	49%	57%
Nodes that are not functions	42%	49%	56%	55%	57%	52%
Nodes with <1% of total application time	3%	89%	99%	99%	99%	78%

We also evaluated removing nodes with an inclusive time of less than 1% of the total application time (orange bars in Fig. 7). This pruning technique produced trees of similar sizes across experiments at different scales: the height of the orange bars is similar for each experiment. This happens because the strategy eliminates more nodes in larger-scale experiments, which tend to have more regions with low inclusive costs. When we applied the technique to the experiment conducted on 10 ranks, we reduced the tree size from 326 nodes to 315. When we applied it to the experiment conducted on 100,000 ranks, the tree size decreased from 97,833 nodes to 224. Table 9 shows that for most experiments, the percentage of code regions with less than 1% of the total application time was 99%. Finally, we evaluated applying all pruning techniques to each experiment (purple bars in Fig. 7). For the experiment on 10 ranks, we reduced the tree size from 326 nodes to 101. For the experiment on 100,000 ranks, we reduced the tree size from 97,833 nodes to 110. These results demonstrate that pruning techniques effectively simplify the calling context tree by removing uninteresting regions and providing a clear abstraction of the program.

6 Conclusions and Future Work

This paper describes the *hpcanalysis* framework for programmatic analysis of large-scale performance data for exascale executions. Rather than reading all profiles and/or traces into memory for analysis, *hpcanalysis* can extract interesting slices of performance data, focus on key calling contexts, or a sample of profiles.

We demonstrated using *hpcanalysis* to explore and analyze an execution of LAMMPS on 64K MPI ranks. By leveraging techniques for pruning large calling context trees and sampling performance profiles, we detected and examined load balance issues and communication anomalies by analyzing a small fraction of the entire tree and a few samples from the entire set of MPI ranks. Pruning and sampling significantly accelerate the analysis of large-scale out-of-core performance datasets containing gigabytes of profiles.

In the future, we plan to extend *hpcanalysis* to support multi-experiment analysis of application performance. Such a capability would enable users to compare an original code with an optimized version or analyze the same code compiled with different compilers or optimization flags. These comparisons will help users identify the most suitable configurations for their application codes. In addition, we will focus more on trace analysis. This will include comparing traces across different execution contexts, detecting phases within traces, and detecting and comparing iterations within and across traces. We also plan to explore using frameworks such as RAPIDS cuDF or Dask in conjunction with our framework to accelerate the analysis. Furthermore, since we now have an interface for sampling and importing HPCToolkit data into Python, we can train AI models to identify performance patterns and suggest optimization strategies.

Acknowledgments

This research was supported in part by UT-Battelle, LLC subcontract CW54422 (DOE Prime DE-AC05-00OR22275), LLNL subcontract B665301 (DOE Prime DE-AC52-07NA27344), ANL subcontract 4F-60094 (DOE Prime DE-AC02-06CHII357), and a contract from TotalEnergies E&P Research & Technology USA, LLC.

We acknowledge the current members of the HPCToolkit project, whose work on HPCToolkit made this independent research possible: Laksono Adhianto, Jonathon Anderson, Mark Krentel, Marty Itzkowitz, Yumeng Liu, and Yuning Xia. We especially thank Jonathon Anderson for collecting the large-scale measurements of LAMMPS on ORNL's Frontier supercomputer.

This research used resources of the Oak Ridge Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research also used resources of the Argonne Leadership Computing Facility, a DOE Office of Science User Facility at Argonne National Laboratory, and is based on research supported by the U.S. DOE Office of Science-Advanced Scientific Computing Research Program, under Contract No. DE-AC02-06CH11357.

References

[1] Laksono Adhianto, Jonathon Anderson, Robert Matthew Barnett, Dragana Grbic, Vladimir Indic, Mark Krentel, Yumeng Liu, Srđan Milaković, Wileam Phan, and John Mellor-Crummey. 2024. Refining HPCToolkit for application performance analysis at exascale. *The International Journal of High Performance Computing Applications* 38, 6 (2024), 612–632.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. 2010. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience* 22, 6 (2010), 685–701.

[3] Jonathon Anderson, Yumeng Liu, and John Mellor-Crummey. 2022. Preparing for performance analysis at exascale. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 34, 13 pages. https://doi.org/10.1145/3524059.3532397

[4] Argonne Leadership Computing Facility. 2024. Aurora. https://www.alcf.anl.gov/aurora.

[5] Argonne Leadership Computing Facility. 2024. Aurora User Guide. https://docs.alcf.anl.gov/aurora/getting-started-on-aurora.

[6] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. Hatchet: pruning the overgrowth in parallel profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 20, 21 pages. https://doi.org/10.1145/3295500.3356219

[7] Abhinav Bhatele, Rakrish Dhakal, Alexander Movsesyan, Aditya Ranjan, Jordan Marry, and Onur Cankur. 2023. Pipit: Enabling programmatic analysis of parallel execution traces. *arXiv preprint arXiv:2306.11177* (2023).

[8] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. 2016. Caliper: performance introspection for HPC software stacks. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '16)*. IEEE Press, Article 47, 11 pages.

[9] Stephanie Brink, Michael McKinsey, David Boehme, Connor Scully-Allison, Ian Lumsden, Daryl Hawkins, Treece Burgess, Vanessa Lama, Jakob Lüttgau, Katherine E. Isaacs, Michela Taufer, and Olga Pearce. 2023. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing* (Orlando, FL, USA) *(HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 281–293. https://doi.org/10.1145/3588195.3592989

[10] Daniele De Sensi, Salvatore Di Girolamo, Kim H. McMahon, Duncan Roweth, and Torsten Hoefler. 2020. An In-Depth Analysis of the Slingshot Interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14. https://doi.org/10.1109/SC41405.2020.00039

[11] Joao Mario Domingos, Pedro Tomas, and Leonel Sousa. 2021. Supporting RISC-V performance counters through performance analysis tools for Linux (perf). *arXiv preprint arXiv:2112.11767* (2021).

[12] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E Nagel, and Felix Wolf. 2012. Open trace format 2: The next generation of scalable trace formats and support libraries. In *Applications, Tools and Techniques on the Road to Exascale Computing*. IOS Press, 481–490.

[13] Intel Corporation. 2025. Application Performance Snapshot User Guide for Linux OS. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide-application-snapshot-linux/2025-0.

[14] Kumar Iyer and Jeffrey Kiel. 2016. GPU debugging and Profiling with NVIDIA Parallel Nsight. *Game Development Tools* (2016), 303–324.

[15] Yuyang Jin, Haojie Wang, Xiongchao Tang, Zhenhua Guo, Yaqian Zhao, Torsten Hoefler, Tao Liu, Xu Liu, and Jidong Zhai. 2024. Leveraging Graph Analysis to Pinpoint Root Causes of Scalability Issues for Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[16] Yuyang Jin, Haojie Wang, Teng Yu, Xiongchao Tang, Torsten Hoefler, Xu Liu, and Jidong Zhai. 2020. ScalAna: Automating scaling loss detection with graph analysis. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14.

[17] Yuyang Jin, Haojie Wang, Runxin Zhong, Chen Zhang, Xia Liao, Feng Zhang, and Jidong Zhai. 2024. Graph-Centric Performance Analysis for Large-Scale Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems* (2024).

[18] Yuyang Jin, Haojie Wang, Runxin Zhong, Chen Zhang, and Jidong Zhai. 2022. PerFlow: A domain specific framework for automatic performance analysis of parallel applications. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 177–191.

[19] Laxmikant V Kale, Gengbin Zheng, Chee Wai Lee, and Sameer Kumar. 2006. Scaling applications to massively parallel machines using Projections performance analysis tool. *Future Generation Computer Systems* 22, 3 (2006), 347–358.

[20] Lawrence Livermore National Laboratory. 2024. Algebraic Multi-Grid Benchmark. https://asc.llnl.gov/codes/proxy-apps/amg2013.

[21] Lawrence Livermore National Laboratory. 2024. El Capitan: Preparing for NNSA's first exascale machine. https://asc.llnl.gov/exascale/el-capitan.

[22] Wes McKinney et al. 2010. Data structures for statistical computing in Python. In *SciPy*, Vol. 445. 51–56.

[23] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710.

[24] NVIDIA. 2024. Nvidia's Nsight Systems recipes for analyzing traces. https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

[25] Oak Ridge Leadership Computing Facilty. 2024. Frontier. https://www.olcf.ornl.gov/frontier.

[26] Oak Ridge Leadership Computing Facilty. 2024. Frontier User Guide. https://docs.olcf.ornl.gov/systems/frontier_user_guide.html.

[27] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-wide profiling: A continuous profiling infrastructure for data centers. *IEEE micro* 30, 4 (2010), 65–79.

[28] The HPCToolkit Project. 2025. hpcanalysis. https://gitlab.com/hpctoolkit/hpcanalysis.

[29] Aidan P Thompson, H Metin Aktulga, Richard Berger, Dan S Bolintineanu, W Michael Brown, Paul S Crozier, Pieter J In't Veld, Axel Kohlmeyer, Stan G Moore, Trung Dac Nguyen, et al. 2022. LAMMPS—a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales. *Computer Physics Communications* 271 (2022), 108171.

[30] Keren Zhou, Laksono Adhianto, Jonathon Anderson, Aaron Cherian, Dejan Grubisic, Mark Krentel, Yumeng Liu, Xiaozhu Meng, and John Mellor-Crummey. 2021. Measurement and analysis of GPU-accelerated applications with HPCToolkit. *Parallel Comput.* 108 (2021), 102837.