Online Performance Observation for HPC Applications

by

Dewi Yokelson

A dissertation accepted and approved in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
in Computer Science

Dissertation Committee:
Allen Malony, Chair
Boyana Norris, Co-chair
Hank Childs, Core Member
Marina Guenza, Institutional Representative

University of Oregon
Spring 2024

DISSERTATION ABSTRACT

Dewi Yokelson

Doctor of Philosophy in Computer Science

Title: Online Performance Observation for HPC Applications

The exascale computing era is providing faster and more powerful systems for advanced HPC applications. However, it is increasingly challenging for programmers to utilize the range of hardware resources that make up these platforms to their fullest extent. Enabling larger, faster, and more diversified simulations requires performance monitoring tools that can integrate seamlessly with applications and operate efficiently in all desired configurations. In addition to critical computational bottlenecks, data movement and I/O performance issues are also important to monitor as data can quickly grow to terabytes and beyond. Thus, a major challenge in high-performance computing is maximizing the performance of many diverse simulations on expensive, energy consuming, and heterogeneous hardware. Furthermore, the landscape of scientific simulations is changing to include increasingly diverse and complex systems, such as coupled applications and workflows. This creates additional considerations in the performance analysis space, where dependencies and task scheduling can play a larger role. This dissertation presents an approach to addressing these issues, wherein we enable performance observability during runtime for different applications and workflows running on heterogeneous architectures. The framework we have created to support this valuable functionality is called Service-based Observability, Monitoring, and Analytics (SOMA). We show how it addresses diverse application and workflow needs across systems, while supporting many useful performance monitoring capabilities with reasonable overhead.

This dissertation includes previously published and unpublished coauthored material.

CURRICULUM VITAE

NAME OF AUTHOR: Dewi Yokelson

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, Oregon, USA
Gonzaga University, Spokane, Washington, USA

DEGREES AWARDED:

Master of Science, Computer Science, 2022, University of Oregon
Bachelor of Science, Mathematics, 2013, Gonzaga University

AREAS OF SPECIAL INTEREST:

High Performance Computing
Online Performance Monitoring and Analysis
Performance Visualization
Performance Modeling and Benchmarking

PROFESSIONAL EXPERIENCE:

Computing Graduate Student Intern, Lawrence Livermore National Laboratory, Computation and Scientific Computing Division, Advisors: Stephanie Brink, David Boehme, Olga Pearce, May 2023 - September 2023

Graduate Research Assistant, Los Alamos National Laboratory, Applied Computer Science, Advisors: Ying Wai Li, Marc Robert Joseph Charest, January 2022 - January 2023

Parallel Computing Summer Intern, Los Alamos National Laboratory, Computational Physics, Advisors: Robert Robey, Ying Wai Li, June 2021 - August 2021

Graduate Research Assistant, University of Oregon, Advisors: Brittany Erickson, Allen Malony, Boyana Norris, Kevin Huck, Spring 2020 - Spring 2024

Graduate Teaching Assistant, University of Oregon, Operating Systems, Introduction to Data Science, Introduction to Programming, Fall 2018 - Winter 2020, Fall 2023

Software Development Supervsior, Expeditors International of Washington Inc., July 2015 - August 2018

Full-Stack Java Developer, Expeditors International of Washington Inc. February 2013 - July 2015

GRANTS, AWARDS AND HONORS:

Gurdeep Pall Graduate Student Fellowship, 2023

General University Scholarship, 2023, 2021, 2019

Selected as a Lead Student Volunteer, Supercomputing Conference (SC), 2022, 2023

Selected as a Student Volunteer, Supercomputing Conference (SC), 2021

Travel Grant Awarded to attend Supercomputing Conference (SC), 2021, 2022, 2023

NSF travel grant awarded to attend International Conference on Software Engineering (ICSE), 2020

J. Donald Hubbard Family Scholarship in Computer Science, 2019

Promising Scholar Award, 2018

PUBLICATIONS:

Enabling Performance Observability for Heterogeneous HPC Workflows with SOMA; Forthcoming (Accepted) at International Conference on Parallel Processing, August 2024; Dewi Yokelson, Mikhail Titov, Srinivasan Ramesh, Ozgur Kilic, Matteo Turilli, Shantenu Jha, Allen D. Malony

SOMA: Observability, Monitoring, and In Situ Analytics for Exascale Applications; Concurrency and Computation: Practice and Experience, Special Issue Paper:e8141, June 2024; Dewi Yokelson, Oskar Lappi, Srinivasan Ramesh, Miika Vaisala, Kevin Huck, Touko Puro, Boyana Norris, Maarit Korpi-Lagg, Keijo Heljanko, Allen D. Malony

HPC Application Performance Prediction with Machine Learning on New Architectures; PERMAVOST Workshop at HPDC, June 2023; Dewi Yokelson, Marc Robert Joseph Charest, Ying Wai Li

Observability, Monitoring, and In Situ Analytics in Exascale Applications; Cray User Group, May 2023; Dewi Yokelson, Oskar Lappi, Srinivasan Ramesh, Miika Vaisala, Kevin Huck, Touko Puro, Boyana Norris, Maarit Korpi-Lagg, Keijo Heljanko, Allen D. Malony

Performance Analysis of CP2K Code for Ab Initio Molecular Dynamics on CPUs and GPUs; Journal of Chemical Information and Modeling, April 2022; Dewi Yokelson, Nikolay Tkachenko, Robert Robey, Ying Wai Li, Pavel Dub

Enabling Cache Aware Roofline Analysis with Portable Hardware Counter Metrics; International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, November 2021; Brian Gravelle, William D. Nystrom, Dewi Yokelson, and Boyana Norris

ACKNOWLEDGEMENTS

Thank you to my advisors, Dr. Allen Malony and Dr. Boyana Norris, without whom I would not have crossed the finish line. Thank you to my previous advisor, Dr. Stephen Fickas for introducing and guiding me in the many opportunities in the world of computer science research.

Thank you to my mentors, Dr. Srinivasan Ramesh, Dr. Hank Childs, and Dr. Kevin Huck, who provided guidance and support at many critical moments. Thank you to my internship mentors at both Lawrence Livermore National Laboratory and Los Alamos National Laboratory who helped me expand my knowledge. To the administrative staff in the Department of Computer Science — especially Cheri Smith and Nicole Moynahan — you are amazing.

Of course, I could not have achieved this without the unwavering support of my family and friends. Special thanks to my sister, and lucky me, also my best friend. Also to my father, who taught me that if you complete a little bit each day, eventually you will have accomplished a lot. To my brother, for reminding me that getting a PhD is not “stupid," as I sometimes complained. And thanks to my mother for always being an unstoppable positive force. To my partner Zachary for offering the most practical and honest advice. Thanks to all my friends who believed in me and have made my life an absolute joy.

This dissertation is dedicated to all of my grandparents, who are unfortunately no longer with us. I thank each of them for contributing to a family legacy of the never-ending pursuit of knowledge that inspires me to this day. I wish you were here.

TABLE OF CONTENTS

Chapter **Page**

Chapter **Page**

Chapter **Page**

LIST OF FIGURES

Figure **Page**

Figure **Page**

LIST OF TABLES

Table Page

CHAPTER I

INTRODUCTION

1.1 Main Research Question

High-Performance Computing (HPC) is an integral part of most modern scientific innovation. We see it everywhere, from achieving ignition to training the artificial intelligence models that are increasingly incorporated into our day to day lives. Running a simulation on a supercomputer can not only reduce the cost of experimentation, but enable us to study outcomes of experiments that would otherwise be too dangerous or impossible to conduct, i.e. nuclear reactions or deep space calculations. Yet HPC, especially at exascale which can be required for accurate simulations, is not cheap. Supercomputer clusters are expensive to build and maintain, with complex, custom cooling systems and high electricity costs. Moore's Law appears to be ending, which means the gap between resource capabilities and actual compute speed is widening. Using these systems more efficiently allows us to continue to scale and grow simulations and their capabilities for further groundbreaking discoveries while keeping resource usage and costs reasonable.

1.1.1 The Limits of Post Mortem Performance Analysis. There are many tools that profile, measure, and can describe the performance of an HPC application or of specific hardware. However, performance data in HPC is notoriously difficult to manage and make sense of because it is complex, high in volume, and high in dimension. Despite these challenges, performance analysis and visualization has come a long way in the last decade, including many new tools and novel graphs that help scientists on this path. Using existing tools and research and being able to bring more of these supported capabilities online could realize large gains, without requiring reinvention of the wheel.

Launching into the era of exascale, simulations are growing, as is the accompanying data that they generate. This can be in the form of the actual scientific simulation data, or the performance data gathered. One major bottleneck in such cases is moving the data across compute nodes, and storing it for post mortem processing. There are some online approaches for reducing the amount of data that needs to be moved and stored. One approach is to generate intermediate results during runtime, which would allow for changes to, or ending the simulation. Another approach is to filter or aggregate the data through such intermediate results and only store a subset of the initial

output. This could amount to something similar to a lossy compression algorithm, with result-aware compression rules when generic compresssion algorithms do not suffice.

Visualizations are critical in presenting performance data in an understandable format. Yet knowing what data to present, and how is a challenge in and of itself. As the volume of performance data grows, there is an increasing need to be able to do more with less data. As one example, if a full simulation takes days to run, it can be unreasonable to do many runs to tune parameters, i.e., different libraries, number of ranks, or number of threads. The ability to know if we have better performance earlier in the simulation would speed up productivity immensely. Another example is when a simulation is run on hundreds or thousands of nodes, with multiple of processes and threads each, and per rank, per thread metrics are gathered. This can actually create I/O or data storage problems, especially if analysis is done *post mortem* (after completion). Thus being able to conduct *in situ* (during execution) performance analysis becomes necessary.

1.1.2 Performance Observation as a First-Class Citizen. In many cases, it is no longer enough to only write a parallel version of a program to speed up an otherwise serial code. On increasingly diverse systems, with new and powerful programming models, more detailed input data, and longer running simulations, code performance becomes one of the most important things, sometimes even at the sake of better accuracy. This has elevated the need to consider performance from the outset, a paradigm shift that has been a long time coming.

Instead of creating a single tool that can do-it-all, and trying to force simulation developers and users to conform their use cases to that tool, we propose a different approach. Instead, this dissertation proposes a framework, a new model if you will, that acknowledges the diverse needs for each specific use case, and enables choice and flexibility. With the increasing importance of code performance, this framework supports the elevation of that analysis as a top concern. We address the very real needs for such a framework in an HPC ecosystem to be effective, and we demonstrate that it works.

1.1.3 Thesis Topic. In the world of exascale computing, with the increasing need for observation of the performance of the simulations, we must elevate this need to being a primary consideration for every simulation. My main aim is to answer the high level research question (RQ) and implementation-specific supporting question (SQ):

RQ What are the requirements for an approach to online observation of simulation performance in an HPC environment?

SQ What implementation choices for a monitoring system can support the requirements identified by answering RQ?

For clarity, we offer a definition of online performance observation in the context of this work.

Online Performance Observation The availability of performance metrics during the runtime of an application for viewing, storing, and analyzing by a human or computer.

Section 1.2 offers a brief overview of Chapter II, which addresses the conceptual approach for answering the main question (RQ), including background information and challenges for creating an observation framework. Sections 1.3, 1.4 and 1.5 introduce the main points of Chapter III, Chapter IV, and Chapter V, respectively. These chapters address the specifics for answering the supporting question (SQ) based on the requirements identified in Chapter II. They aim to answer the following questions:

- **Chapter III:** How can we run on heterogeneous HPC ecosystems?
- **Chapter IV:** How do we support different input and output?
- **Chapter V:** How do we minimize the overhead?

1.2 Requirements for Supporting Performance Observation

HPC simulations are anything but homogeneous; from which language they are written in, to whether they contain memory-bound or compute-bound regions, to whether they utilize only CPUs or are GPU accelerated, these are just some of the many ways that they can differ. In this respect, forcing a one-size-fits-all solution, by creating a single performance monitoring tool with specific analysis and visualization functionality would be doomed from the start. Many performance measurement tools exist that attempt to do just this, but the result is that they excel in some aspects, or are useful for some codes, but they lack enough features to be the best solution across the board. An example stems from the benefit fully instrumented codes that generate detailed traces

— if a scientist has taken the time and effort to implement this in their code it would be useful to support new analysis or visualization without requiring a complete refactor to a new tool.

Background information and related work are described within Chapter II. Then, the three key requirements that were identified and inspired the three chapter research questions above are listed as follows, and then they are described in further detail after:

1. Configurable: Accommodating many different setups in order to support the diverse nature of HPC simulations and ecosystems.

2. Flexible: Both for input — the ability to measure many different metrics and simulations — and output — the ability to analyze and visualize many different types of data.

3. Performant: Efficient in HPC ecosystems, and producing a reasonable amount of overhead when observing and analyzing.

Our proposed solution for enabling in situ and online observation is to utilize existing performance measurement and analysis tools, and enable their online use via a microservice-based framework. We needed something that would be flexible, configurable, and performant. We propose the Service-based, Observation, Monitoring, and Analytics (SOMA) framework as a solution for bringing existing tools for analysis of HPC applications online or in situ as debuted in [157, 158]. See Figure 1 for a high-level view of the interaction between SOMA and existing performance tools, acting as an enabler for access to online performance data.

1.2.1 Configurable. Configurability is crucial because different simulations will have different amounts or types of data and our system must be able to adjust to serve it well. We need to be able to monitor different metrics across HPC ecosystems at different times. Additionally, running even the same simulation on different hardware with different network connections may necessitate changes to any of these settings. These requirements inspired the question “How can we run on heterogeneous HPC ecosystems?" Section 1.3 details the configurable aspects of our service-based approach in SOMA, and how we used it to support the diverse needs of online observation of HPC applications.

1.2.2 Flexible. Flexibility is important because only an adaptable service will be able to interface with the existing tools, and in varying combinations. For example, some performance measurement tools are geared towards measuring memory usage, while others might focus on I/O.

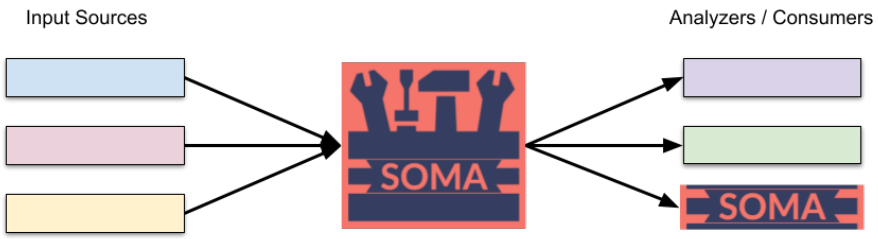

Figure 1. SOMA enables online data access from many diverse input sources such as performance profilers, or the application itself. It make the data available for use by many diverse consumers for analysis or visualization.

From an analysis perspective, some tools might generate excellent roofline plots, while others provide algorithms for identifying memory leaks or suggesting kernel optimizations. Being able to integrate with multiple tools and support the online movement of the data they generate or ingest is necessary for widespread adoption of an online observation framework. These requirements inspired the question “How do we support different input and output?" Section 1.4 describes our approach to providing a canonical data model structure which supports many different performance metrics.

1.2.3 Performant. Performance is important because we need to be able to manage the large amounts of data generated by the applications with little enough overhead that it is worthwhile to collect and analyze said data. Making use of existing HPC technologies that support efficient performance is critical for this endeavour. Since what constitutes an acceptable amount of overhead may be somewhat subjective based on the developer, we must take this into account. We must consider the trade-offs between the amount and granularity of data we can collect with the impact to the application. These requirements inspired the question “How do we minimize the overhead?" Section 1.5 discusses our approaches to reducing the monitoring overhead.

Co-author Acknowledgement Chapter II contains both unpublished and published material with and without co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Allen Malony, Dr. Boyana Norris, Oskar Lappi, Dr. Srinivasan Ramesh,

Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, Dr. Mikhail Titov, Dr. Ozgur Kilic, Dr. Matteo Turilli, and Dr. Shantenu Jha.

1.3 A Microservice Approach

This section outlines how Chapter III answers the question, "How can we run on heterogeneous HPC ecosystems?" While all three requirements and questions from section 1.2 contributed, the configurability requirement especially, led us to the decision to use a microservice inspired architecture that is supported by the existing MOCHI software stack. This eases the composition of these services by providing the client/server framework and the remote procedure call (RPC) infrastructure needed for a distributed HPC environment [114]. We can pick and choose from the SOMA client types to use based on the performance data sources that are needed for the particular simulation. On the server side, any number of analysis backends can then be initialized to act on the RPC data stream.

We built on top of the successful ideas behind the SYMBIOMON [110] online monitoring and SERVIZ [109] shared visualization service projects. We extended the configurability of these solutions by introducing frequency of data collection and publication in numerous capacities. SOMA [157, 158] also has the ability to initialize any number of clients or server endpoint instances to meet the demands of the amount of data being generated and published. The ability to very closely configure the number of clients, servers, and RPC calls can be especially useful for preventing disruption by the monitoring system if a simulation is already network-bound.

With the data models for metrics as described in section 1.4, we use a SOMA client to publish the collected data during the application execution. This data can then be consumed, aggregated, and analyzed by another SOMA client to generate online results. The benefits of this project not only lie with the potential for data reduction, but also in the ability to end a simulation early and relaunch it with different configurations if a problem is identified. Understanding how an application behaves in real time is telling, when we can identify certain patterns with certain code regions we can identify specific issues and make improvements to the code. This chapter discusses how we support different analysis and visualization procedures with the gathered data. Examples of how our framework works are detailed in this chapter as well as the different computer architectures that we have successfully used.

Co-author Acknowledgement Chapter III contains both unpublished and published material with and without co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Allen Malony, Dr. Boyana Norris, Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, Dr. David Böhme, Dr. Mikhail Titov, Dr. Ozgur Kilic, Dr. Matteo Turilli, and Dr. Shantenu Jha.

1.4 A Canonical Data Model

This section outlines how Chapter IV answers the question "How do we support different input and output?" A key aspect for supporting flexibility in this manner is introducing a Conduit data model [52] into the SOMA framework. Conduit enables any SOMA client to structure data into Conduit hierarchical "node" structure and submit them via RPC to the server endpoints. Now we have a shared data model for any input or client side application, and we can take advantage of the fact that Conduit was made to be performant on HPC ecosystems, with built in serialization capabilities. We implement Conduit data models in four separate domains which are: performance profiles, application diagnostics, hardware metrics, and workflow states. These can be used on their own or in combinations together to contribute to an in-depth understanding of application performance in HPC ecosystems.

One other major benefit of enabling the SOMA service to collect, monitor, and analyze different performance data online is the potential for providing feedback to the simulation or workflow. We demonstrate how we collect and analyze data that can be used in this regard for both the Astaroth codes, and for a RADICAL-Pilot (RP) workflow. In the case of Astaroth, we identified application diagnostic metrics that indicated when the simulation is becoming unstable, this could allow the simulation to be corrected and continue, or stopped if it is too late. Section 1.4.4 introduces the integration between the RP pilot system and SOMA to collect and analyze not only scientific and performance data, but workflow state data. We demonstrate case studies enabling online feedback for making changes to the workflow runtime parameters. For the RP workflow we collect and analyze data about the state of each task and the corresponding hardware behavior on those compute nodes.

1.4.1 Performance Profiles.

The ability to leverage existing tools, where *post mortem* data processing limitations can start to be felt, and bringing them online for faster analysis is a major use case for the SOMA framework. This section describes two such integrations, showcasing

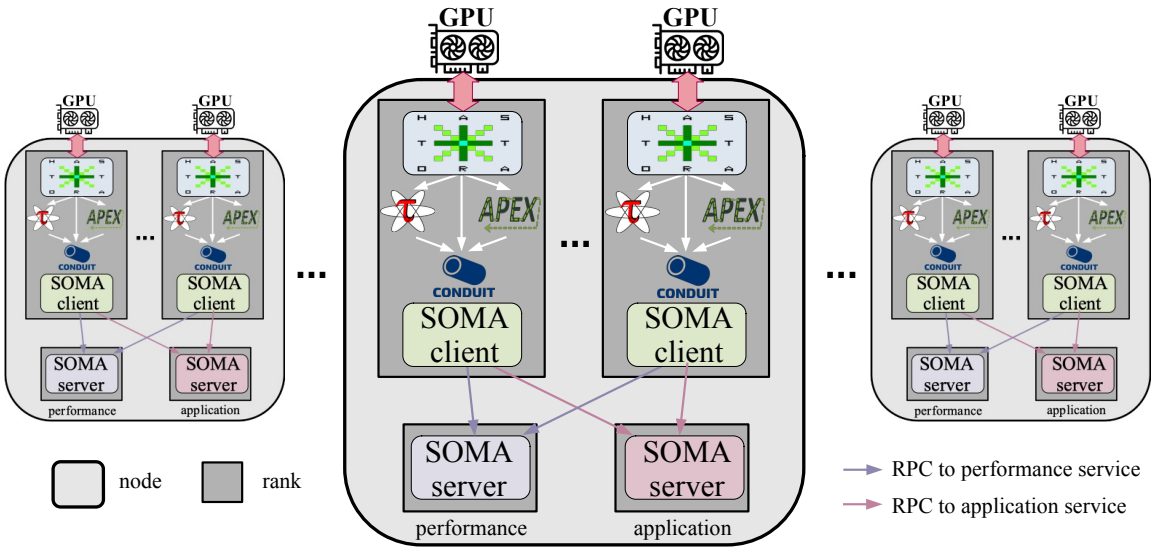

Figure 2. An example use case of the SOMA stack, conducting both performance and application data monitoring and publishing the data to server instances. The data can be used in a variety of different ways with existing tools and interfaces.

the flexibility of SOMA and it's ability to support very different performance analysis tools making use of the canonical Conduit data model.

1.4.1.1 TAU. TAU is a performance analysis library that supports both the sampling of any codes, and more robust instrumentation and tracing [87]. One of the first integrations built into SOMA was a TAU plugin that can translate a TAU profile into a Conduit data node and publish the data an configurable intervals for online analysis [157, 158].

1.4.1.2 Caliper. We propose another such pipeline by creating an integration between the Caliper Performance Analysis Library [23] and the SOMA framework, similar to the integration with TAU. Caliper data has the advantage of being closely linked with the analysis tool thicket [25] for robust analysis features.

1.4.2 Application Diagnostics. Domain scientists may have specific metrics within their codes that they want to monitor to ensure performance. This is not necessarily the intermediate or final results of the simulation, but rather an intermediate diagnostic that might indicate if the simulation is successful, though in some cases they may be one and the same. These diagnostics could indicate the simulation is becoming unstable, or needs to be stopped or reconfigured. We demonstrate how we can collect application-specific diagnostic data via the service-based SOMA framework. We discuss results and overhead for both the LULESH proxy application and large-

scale, multi-GPU astrophysics code, Astaroth [157]. Figure 2 shows an example of the structure of the SOMA clients and servers and how they can relay information from both the performance profile and application diagnostic sources to different destinations.

1.4.3 Hardware Metrics. Associating hardware performance with what is happening in the code can add another layer of understanding to complex HPC ecosystems. We implemented a SOMA client which gathers cpu utilization hardware metrics from `/proc/stat` and structures it into a Conduit node, which allows for a visualization of the cpu utilization over time. This can enable both fine-tuning of code performance as well as insights about where and when tasks are scheduled for improved resource utilization.

1.4.4 Heterogeneous Workflow States. RADICAL-Pilot (RP) is a pilot system for executing ensembles or heterogeneous workflows, that is, multiple concurrent simulations, on HPC resources [97]. These are useful to scientists for a number of reasons, including executing coupled simulations or achieving a faster time to solution by running duplicate, or slightly varied versions of the same simulation at the same time. Creating a SOMA client integrated with RP brings new functionality to both tools. RP can easily launch numerous SOMA servers, and a simulation ensemble concurrently on an HPC cluster, in a configurable manner. This allows us to gather the scientific data, performance data that we normally could with SOMA, but additionally allows for workflow metadata to be collected and analyzed, all online.

Co-author Acknowledgement Chapter IV contains both unpublished and published material with and without co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Allen Malony, Dr. Boyana Norris, Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, Dr. David Böhme, Dr. Mikhail Titov, Dr. Ozgur Kilic, Dr. Matteo Turilli, and Dr. Shantenu Jha.

1.5 An Exascale Performer

This section outlines how Chapter V answers the question "How do we minimize the overhead?" The act of observing something inherently generates some amount of overhead. However, we chose to use the existing Mochi and Conduit technologies because they were made to be exceptionally efficient on HPC systems already. Mochi makes use of the high performance networks that are available on each cluster to transfer data from client to server. Conduit has highly efficient serialization procedures for streaming. In addition to the benefits of using these technologies, our configurability

from section 1.3 makes an impact on the performance front as well. With a nearly unlimited options of configurations for number of clients, servers, publication rates, etc. we can tweak them many different ways to reduce monitoring overhead. More is described on this topic in section 1.5.1.

We implemented a robust application programming interface (API) for SOMA that includes the ability to submit both synchronous and asynchronous RPCs. This functionality allows for the client to use either blocking, or non-blocking calls, depending on the needs of the application or system being monitored. The API also allows for each client to commit to it's own “namespace" in order to keep the different data sources separated. Similar to a transactional database commit, we can control the frequency of the RPC calls in this way as well [157, 158].

1.5.1 Monitoring Overhead. We compare usage of different network connections, API calls, publication rates (i.e. changing the frequency of RPC calls), and number of SOMA servers — which can enable faster processing of receieved data. These comparisons are carried out using the LULESH, and Astaroth applications [157, 158]. Results do show there is typically an overhead cost incurred when monitoring with SOMA, but that it can be minimized by applying thoughtful strategies. In this section we explore in depth how different configurations of SOMA affect the measured monitoring overhead.

1.5.2 Adaptive Feedback Potential. One other major benefit of enabling the SOMA service to collect, monitor, and analyze performance data online is the potential for providing feedback to the simulation or workflow. We demonstrate how we collect and analyze data that can be used in this regard for a RADICAL-Pilot (RP) workflow. We collect and analyze data about the DeepDriveMD Mini-app workflow and analyze the state of each task and the corresponding hardware behavior on those compute nodes. This opens up the potential for online reconfiguration of how the workflow is scheduled across resources, or how the workflow tasks are implemented and run. We show how we can access and analyze the data online, the next steps are to “close the loop" and feed the data back to the simulation or workflow manager for online adaptation.

Co-author Acknowledgement Chapter V contains both unpublished and published material with and without co-authorship. Specific contributions are detailed in the beginning of the chapter, but co-authors include Dr. Allen Malony, Dr. Boyana Norris, Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, Dr. Mikhail Titov, Dr. Ozgur Kilic, Dr. Matteo Turilli, and Dr. Shantenu Jha.

CHAPTER II

STATE OF THE ART IN PERFORMANCE OBSERVATION

This chapter contains unpublished and published material with and without co-authorship. Sections 2.1, 2.2, 2.3, 2.4, 2.5 contain material from the departmental requirement, the Area Exam, which is a survey of the state of the art in this research field. For the Area Exam, I was the only author, and I completed all writing, but received guidance, feedback, and suggestions from my dissertation committee members: Dr. Boyana Norris, Dr. Allen Malony, and Dr. Hank Childs.

Sections 2.3, 2.4 and 2.6 contain some published material from the a paper that was initially published at the Cray User Group conference in 2023. An extension of this paper was published in a special edition journal of Concurrency and Computation: Practice and Experience in June 2024. The work was a collaboration between the University of Oregon, NVIDIA Corporation, University of Helsinki, Aalto University, and Academia Sinica. Co-authors include Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Boyana Norris, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, and Dr. Allen D. Malony. I was the first author, but some co-authors helped with writing and figures, all helped with suggestions, and proof-reading. Dr. Malony wrote most of the subsection on the TAU Performance System 2.3.2.1.

Sections 2.4 and 2.6 also contain some unpublished material from a paper that is is under review at a 2024 conference. This work was a collaboration between University of Oregon, NVIDIA Corporation, Brookhaven National Laboratory, and Rutgers University. Co-authors include Dr. Mikhail Titov, Dr. Srinivasan Ramesh, Dr. Ozgur Kilic, Dr. Matteo Turilli, Dr. Shantenu Jha, and Dr. Allen D. Malony. I was the first author for the paper, conducted all experiments and completed the majority of the writing. Some co-authors helped with writing, all helped with suggestions, and proof-reading.

2.1 Introduction

HPC systems have been driving scientific discovery in many domains, yet utilizing them as efficiently as possible still poses a major challenge. The benefits of running simulations on HPC systems are countless. They have made it possible to model extremely intricate scientific systems that require a large volume of data, complex calculations, and high precision. It is necessary to expose, and use the parallelism in HPC applications in order to process these volumes of data

quickly. Scaling these simulations across multiple nodes, each of which contain many cores can become difficult quickly, even for experts in the HPC domain.

Effectively using the necessary tools and programming models such as CUDA [102], OpenMP [32], and MPI [48], presents developers with additional challenges. This is especially true when these are used in conjunction with highly optimized for parallel performance math and modeling libraries, e.g., BLAS [63], LAPACK [9], and FFTW [41]. The steps to compile and marry together different versions and implementations of these libraries with the best thread and rank count can create an prohibitively large search space for optimal parameters.

Performance analysis and visualization of high-performance computing (HPC) codes, while complex, can be one of the most useful tools for improving HPC application efficiency. Moving the analysis and visualization earlier in a workflow can save scientists enormous amounts of time and money when they are optimizing and running their experiments on expensive and in-demand, HPC resources. This chapter surveys current performance analysis and visualization capabilities and challenges, and analyzes the current work and existing opportunities of taking a more *in situ* approach. This approach would boost efficiency and enable analysis and visualizations that might otherwise require prohibitively large amounts of data, thus incurring too much I/O overhead. We look at current work in this area, as well as the neighboring fields of scientific analysis and visualization, and information visualization.

2.2 Background and Definitions

In order to achieve optimal wall clock time for an application, it is important to monitor and analyze the performance, which adds yet another layer of data complexity. For an HPC application, this performance is a crucial consideration. Without sufficient performance of the application, both time and money are wasted on expensive resources. However, it can be difficult to understand the performance of such applications and systems. There are many aspects to this, including: understanding what the current application performance is, figuring out a realistic performance benchmark to achieve, the optimal parameters, and diagnosing reasons for poor performance so as to improve it.

One such way to help communicate these concepts and metrics is via data analysis and visualizations of performance data gathered from many of the different measurement applications. There are many tools that profile, measure, and can describe the performance of an application or of

specific hardware. However, performance data in HPC is notoriously difficult to manage and make sense of because it is complex, and high in volume. Despite these challenges, performance analysis and visualization has come a long way in the last decade, including many new tools and novel graphs that help scientists on this path.

K.E. Isaacs et al., provided a taxonomy of performance visualization organized into four *contexts*: hardware, software, tasks, and application [61]. Hardware covers physical structures of the hardware and their performance. The software category encompasses the source code, or application that is being run. Tasks are similar to software but with the source code context removed, still actions that are happening, but without any relation to how the code was written. The application context contains things like linear algebra calculations or the computational aspects of the program. Isaacs points out that some visualizations are characterized by more than one of these contexts. While this taxonomy was written for performance visualization specifically, we believe it applies also more generally to how to understand application performance, thus, also analysis.

The contexts provided by Isaacs et al. help us to understand what kinds of performance visualizations (and analysis) already exist, what they do well, and where there are opportunities for improvement. As a general example, some of these visualizations, especially of the hardware nature, rely on reducing the dimensionality of the data in order to present something easily digested by the reader. As problem spaces get larger, with increasingly complex code on larger clusters with more cores per node, reducing the dimensionality proves to be a bigger challenge [61]. These premises provide a good baseline understanding for discussing new methods for analyzing performance data and generating hardware-specific understanding and visualizations.

Analysis and visualizations are critical in presenting performance data in an understandable format. Yet knowing what data to present, and how is a challenge in and of itself. As the volume of performance data grows, there is an increasing need to be able to do more with less data. As one example, if a full simulation takes days to run, it can be unreasonable to do many runs to tune parameters, i.e., different libraries, number of ranks, threads. The ability to know if we have better performance earlier in the simulation would speed up productivity immensely. Another example is when a simulation is run on hundreds of nodes, with hundreds of processes and threads, and per rank, per thread metrics are gathered. This can actually create I/O or data storage problems,

especially if analysis is done *post hoc* (after completion). Thus being able to conduct *in situ* (during execution) performance analysis becomes necessary.

In this chapter, we contribute discussion on the following recent advances in the field of performance analysis and visualization:

1. Successful performance analysis and visualization concepts, common challenges, and how they have been managed.
2. An analysis of many of the current tools and their "readiness" for *in situ* or online observation.
3. Current *in situ* and *post hoc* performance prediction and problem diagnosis techniques and what opportunities have arisen there.
4. A look into the fields of scientific visualization and information visualization for inspiration.

Further research opportunities are also discussed in terms of the following challenges that are related to implementing *in situ* performance analysis and visualization in table 1.

Table 1. Four of the universal challenges associated with implementing in situ performance analysis and visualization

Challenge	Description
(A) Superfluous Data	A direct effect of the high-dimensionality problem, managing large amounts of data and analyzing and visualizing only items of interest
(B) Incomplete Data	Because a program has not completed, not all the possible data will have been collected, could lead to incorrect conclusions
(C) Distributed Data	The parallel nature of applications means that performance data could be distributed across resources and require synchronization
(D) Limited Resources	Collecting performance metrics already increases overhead, adding analysis and visualization generation can exacerbate this issue

The following sections provide some background on how performance data of HPC codes is gathered and structured for use. The structure of the data and it's many possible dimensions is one of the largest contributing factors to the challenge of analysis and visualization.

2.2.1 Dimensions. The high-dimensional nature of performance data is one of the challenges to creating meaningful 2d, 3d or even 4d (showing a change over time) analysis or visualizations. When thinking about the environment that HPC codes run in, we can quickly see

why. First, there can be the code logging events and gathering timing data for specific functions to complete. Then this code could be multi-threaded or multi-processed, with numerous threads or processes executing simultaneously. It could be a hybrid application with multiple processes and multiple threads per process. This could then be running across many compute nodes. Parallel-enabling programs such as MPI and CUDA have their own overhead and sometimes functions that should be tracked for timing data. Then there are the hardware metrics that can be tracked, such as the memory bandwidth during an application run. The data can quickly become unwieldy, for a program running on multiple compute nodes, for multiple days, with multiple ranks and threads.

2.2.2 Profiles. A large subset of performance analysis of HPC codes is done with the use of profile data. Profiles are essentially metadata about the code running that gets written out during runtime (dynamic). This metadata usually includes information like the function name, current line in the code, and performance information such as thread ID that can be used to determine how long certain sections of code take to run, among other things. These profiles are multi-dimensional data files, often containing information per thread, or node, etc. They can be tricky to parse into a format that cooperates with non HPC specific analysis and visualization tools.

Profile data can be gathered either through instrumentation or sampling. Instrumentation of the code is a more involved method that typically requires annotating and recompiling the code — the tool used to instrument will insert code that writes out performance data periodically. This is the most comprehensive method, but it can often be time-consuming to implement and may not be necessary if sample data is sufficient. Sampling is when the tool interrupts the HPC application periodically to query for the metadata mentioned above. Sampling usually requires less set up but offers the programmer less control over when and how the application is monitored. For a large application that runs for a significant period of time, sampling can often be sufficient as it averages out to being quite accurate over many timesteps.

2.2.3 Trace Data. The second category of performance data that is widely used in analysis and visualization tools is trace data. Traces can include the same data as a profile, but contain more detail, most notably with the addition of a temporal dimension. It can be thought of as a detailed log of events happening while the codes run. Trace data can be generated at many different levels including: the application, the compute node, the rank or process, and the thread. While this level of detail can be extremely useful, it can also quickly become unwieldy.

Because of this complexity and potential volume of trace data as well as the wide array of tools that use it. A standardization project has been in place for some time. The current standard format for trace data is give by Open Trace Format 2 (OTF2) [38]. This built on the predecessor Open Trace Format (OTF) [71, 85] and the EPILOG format [147]. The main improvements were adding more flexibility between different use cases and additional scalability.

2.2.4 Important Performance Metrics. One of the most common metrics that is used when analyzing performance of an application is the wall clock time, often also referred to as execution time, or time to completion. This is simply the full length of time that it takes for the application to run from start to finish. While this measurement is an excellent indicator of the performance of the system, and is often the main target for decreasing, it lacks the nuance that may be required to gain additional performance. To identify these further opportunities, function or kernel specific data, or hardware specific data is more helpful. For example, knowing how long a program spent executing one specific loop can give insight into how it could be optimized.

Hardware performance counters are registers that keep track of the count of certain events (e.g., number of cache misses). These events imply how efficiently code is executed on the hardware, and can be gathered by accessing API's like PAPI [26, 134]. Efforts to standardize this access across tools have been taken on with projects such as Score-P [72]. Examples of tools used to gather hardware counter information include: TAU [121], HPCToolkit [1], Caliper [23], Survey [132, 101], and Likwid [115]. The following sections provide some background on how performance data of HPC codes is gathered and structured for use. The structure of the data and it's many possible dimensions is one of the largest contributing factors to the challenge of analysis and visualization.

2.2.5 Performance Observation v Monitoring v Analysis. An important distinction we want to make in this dissertation are the differences between how we use the terms online observation, monitoring, and analysis in relation to performance data. Performance observation, as discussed in Chapter I involves enabling the availability of performance data online. Online performance observation thus enables performance monitoring, which is the act of viewing the performance data that is currently available. Finally, online analysis is the next step, where calculations, conclusions, or visualizations are drawn from the available data during runtime.

Tool	Trace		Profile		Arch.	CPU		Portable
	Thread	Rank	Inst.	Sampling	Roofline	Loop	Function	
Intel Advisor					X	X	X	
Intel VTune	X	X	X	X				
NSight Systems	X	X		X	X			
TAU/Paraprof	X	X	X	X			X	X
Perfetto	X	X	X	X		X	X	X
Chrome Tracing	X	X	X	X				X
HPCToolkit	X	X		X		X	X	X
Ravel	X	X					X	X
ERT					X			X
TAU/Vampir	X	X	X	X			X	X
Hatchet			X	X			X	X
CallFlow			X	X			X	X
Grafana	X	X	X					X

Table 2. Analysis and visualization capabilities of some HPC performance tools

2.3 Offline Performance Analysis and Visualization Tools

This section encompasses the current work and effectiveness within the field of performance analysis and visualization in general, without online capabilities. First, we discuss what is currently being measured, analyzed, and visualized, and at what granularity. Second, what makes these tools effective, including examples of some such successful ones, and a detailed description of TAU and APEX. Following this is a deep dive into the cache-aware roofline model, one of the most successful performance analysis concepts that has been adopted into many tools.

2.3.1 Current Performance Analysis and Visualization. Current capabilities for performance analysis and visualization span across many scopes, including, the actual code performance, single-node hardware performance (FLOP rates and memory bandwidth), distributed memory system performance (multiple node), and the relationship between some of these. Table 2 categorizes the different capabilities of some of the existing tools. From an application or software perspective, we can measure something as high level as the differences between wall clock times, or as granular as how long a loop takes to execute. Other concepts that are often measured and visualized are a full call graph of the application, and time spent in each function over the course of an application run, denoted in the “Profile" column of table 2 (by either instrumentation, or sampling). Table 2 also specifies in the tools that do trace analysis and visualization, which is simultaneously the most useful data due to its detail, but the least user-friendly due to its volume. This is discussed in much further detail within the next section, section 2.3.2.

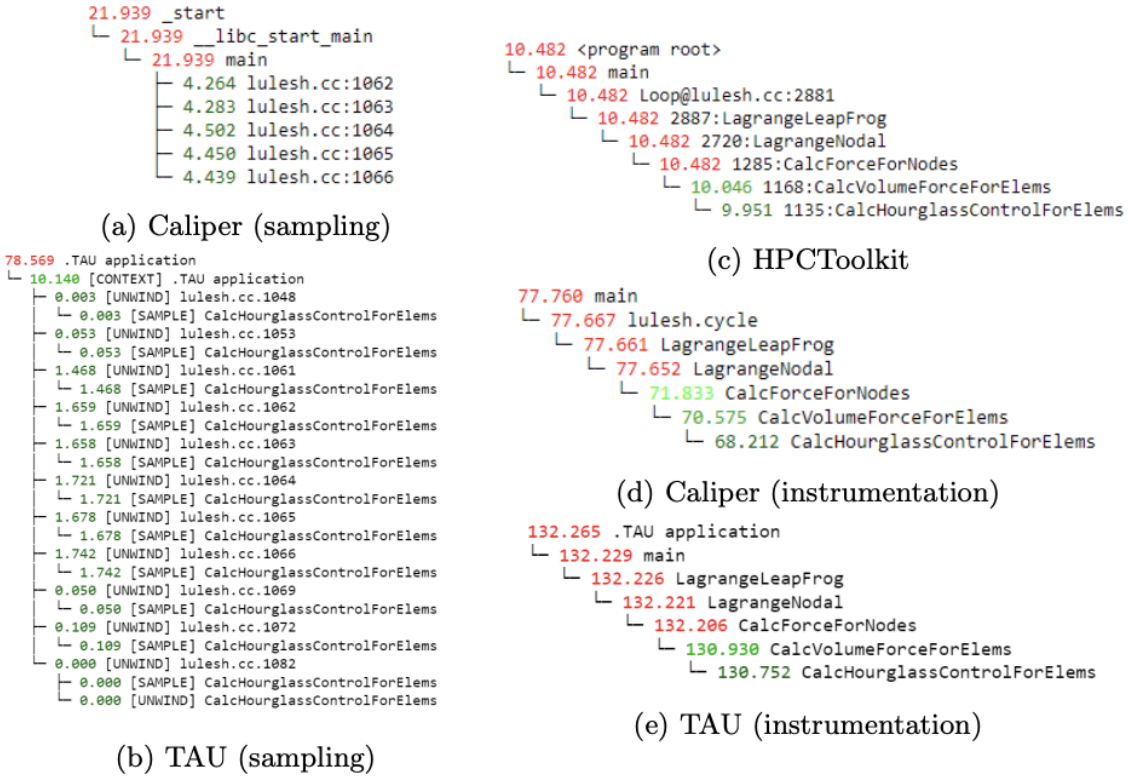

Figure 3. An example of four different call graphs generated by the Hatchet tool for data gathered for the same code, CalcHourglassControlForElems, by three heavily used analysis tools, Caliper, HPCToolkit, and TAU [28]. There are notable differences between the clarity of the call graphs with data obtained through instrumentation.

Call graphs and calling context trees are the two most common formats for structured profile data acquired through instrumentation or sampling. Call graphs provide a more static view of any interrelated functions (that call each other), disregarding how the application currently got to the specific function sampled, see Figure 3. Calling context trees show a more dynamic view by using the stack trace to expose to the programmer what functions called each other to get to the current state in this run of the application. Both are extremely important ways to structure the data as the call graphs are more straightforward but the calling context tree provides more detail and potentially important context about the application.

Another concept that researchers have been measuring, analyzing, and visualizing is memory bandwidth, as the majority of applications are memory bound [113]. Two concepts to understand here are the hardware capacity (maximum), and how well the application is utilizing the full bandwidth available. MemAxes is a tool developed to target specifically the memory domain of performance data [44]. Their approach takes into account source code, data structures, and hardware. They create interactive visualizations to communicate potential bottlenecks or issues with this complex relationship. Their sunburst-like visualizations are quite unique, which can be helpful if it can communicate well, yet require the consumer to adapt to a new visualization type.

Hardware performance analysis is often encompassed by measuring the peak performance of the compute node or cluster. The cache-aware roofline model, which is discussed in detail in section 2.3.3 calculates both the theoretical maximum memory bandwidth of the hardware system (per cache level), and the theoretical maximum FLOP rates. It also allows for incorporating a calculation of the actual bandwidth and FLOP rates (arithmetic intensity) achieved by the software for a full application, function, or loop. Another method of hardware performance analysis and visualization that is common to see is the stacked bar chart displaying the Top-Down metrics in conjunction with Top-Down Analysis. This method allows users to visualize at a high level and dive into more detail on hotspots [154]. This method was developed at Intel and has been incorporated into Intel VTune [130].

2.3.2 Effective Performance Analysis and Visualization. One of the major challenges facing the measurement of software trace data today is that they do not scale well with the amount of data collected.For the complex nature of performance data, the ability to flatten the data into something digestible by a human is a necessity. Many tools employ approaches to this technique,

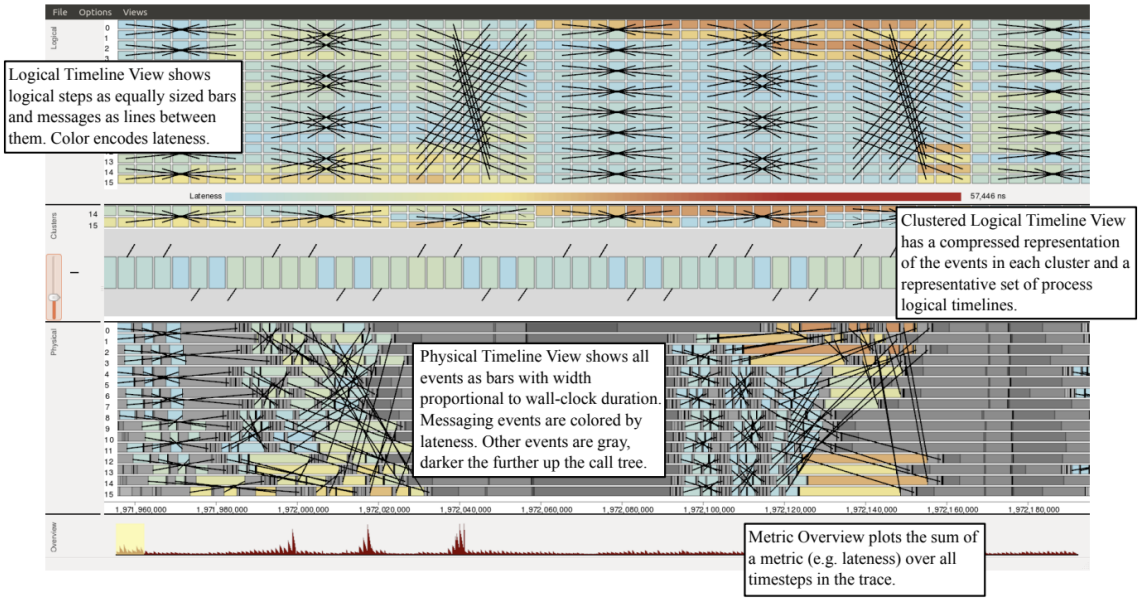

Figure 4. An example of the trace visualizations generated by Ravel, each row represents a thread and the color represents a procedure. [60]. By nature, trace data can get unwieldy quickly, and Ravel was built specifically to tackle the challenge A (scalability) in table 1 could be a particular challenge when visualizing trace data like this.

including but not limited to HPCToolkit [1], TAU [120], and Grafana [129]. To address this issue, *Ravel*, structures the typical trace data, into that of *logical time* [60] based on the happened-before relation introduced by Lamport [76]. *Logical time* in this context means looking at relationships between events, and the order in which they occur, rather than when events occur on a physical timeline. It takes the code structure into account, as well as making grouping of events easier. More details about the specific implementation, and topics like *lateness* can be read in section 3 of their paper [60]. Through preserving important physical timing data and different applications of clustering data and color coding, they create visualizations that are more easily understood, even at a higher processor count.

HPCToolkit is a popular and relatively widely used across domains and platforms. HPCToolkit gathers performance metrics with only a few percent overhead [1]. The HPCToolkit workflow includes measurement, analysis, correlation, and presentation.

Google has also created a robust trace analysis and visualization tool called Perfetto. Perfetto can analyze and visualize data from a number of different sources. It runs completely inside of the web browser and has interactive zoom and select capabilities. This allows the user to control their

view and dive deeper into areas of interest. This can even be done offline once the trace file is opened inside the browser window [131].

2.3.2.1 TAU and APEX. The TAU project began in the early 1990s with the goal of creating a performance instrumentation, measurement, and analysis framework that could produce robust, portable, and scalable performance tools for use in all parallel programs and systems over several technology generations. Today, the TAU Performance System® [120] is a ubiquitous performance tool suite for shared-memory, message passing, and task-based parallel applications written in multiple programming languages that can scale to the largest parallel machines available. It is installed on many HPC systems around the world and is used on a daily basis for performance analysis and tuning of applications across multiple domains.

The TAU Performance System [120] consists of two toolkits: the tuning and analysis utilities (*TAU*) and the autonomic performance environment for exascale (*APEX*). The TAU model of performance measurement is based on a "worker" (first-person) perspective. Essentially, each thread of execution in a program will make performance measurements with respect to its operation. A measurement could occur as a results of an instrumentation probe the thread executes or an event-based sample interrupt that occurs on that thread. All performance data (e.g., time, HW counters) are stored within the thread context and retained during execution. All threads output their performance information when the program terminates. Many HPC performance tools are like TAU, including HPCToolkit [1], Score-P [73], Scalasca [148], and Caliper [23].

In contrast, APEX [56] is based on a "task" (third-person) perspective, with event-based and sample-based measurements. APEX uses an event API and event listeners to observe when a task is created, started, yielded or stopped, and updating timers for measurement. (Note, this is with respect to what constitutes a task, not necessarily its thread of execution.) Dependencies between tasks are also tracked, using globally unique identifiers (GUID). APEX periodically and on-demand interrogates (samples) OS, hardware, or runtime states (e.g., CPU utilization, resident set size, or memory "high water mark"). This also occurs in TAU, but in a different manner. APEX measurement includes background buffer processing to record GPU kernel execution and memory transfers to and from GPUs. Available runtime counters (e.g., idle rate, queue lengths) are also captured on-demand or on a periodic basis.

Both TAU and APEX can produce profiles and traces. With this in mind, the TAU profile data model provides for a type of analysis that can look at individual thread operation and the performance of particular events across multiple threads. It is possible to compute statistics for specific events to get a sense of aggregated performance. APEX is particularly appropriate for task-based runtime environments. Existing and emerging programming models present technical challenges that first-person measurement systems had not considered: untied task execution and migration, runtime thread control and execution, state sampling, and runtime performance tuning. APEX can address these issues while being lightweight enough to be present in an application for continuous performance introspection and adaptation.

Several programming systems and communication libraries implement a performance interface that allows tools to observe events and associated data associated with those components (e.g., OMPT [103] for OpenMP and PMPI [40] for MPI). Some adopt a plugin design that enables tool connection at runtime. However, user code instrumentation generally lacks support for tool interfaces. The PerfStubs library [22] is a thin, stubbed-out, “adapter" interface for instrumenting library or application code. The PerfStubs library itself does not do any measurement, it merely provides access to an API that performance tools can implement.

We have mentioned the wealth of robust performance measurement and analysis tools that have been developed for HPC systems and applications. These include HPCToolkit [1], Score-P [73], Scalasca [148], Extrae/Paraver [106], Caliper [23], Timemory [83], and others, as well as machine-specifc vendor offerings. For the most part, these tools were designed for offline performance analysis and tuning, with a focus on first-person performance measurement of tied task functions on a per-thread OS thread basis. In addition to capturing time and hardware counter data, some of the tools also support heterogeneous systems and are able to measure GPU performance. TAU provides a comprehensive set of performance measurement and analysis capabilities that covers practically all HPC environments and parallel computing models.

In contrast, there are fewer performance tools that address existing/emerging programming models and runtime systems where there exists untied task execution and migration, runtime thread control and execution, third-person observation, and runtime performance tuning. What also sets APEX apart from runtime-specific solutions is that it has been refactored from its HPX-centric design [64] to a more general purpose asynchronous multi-tasking runtime profiling library. TAU

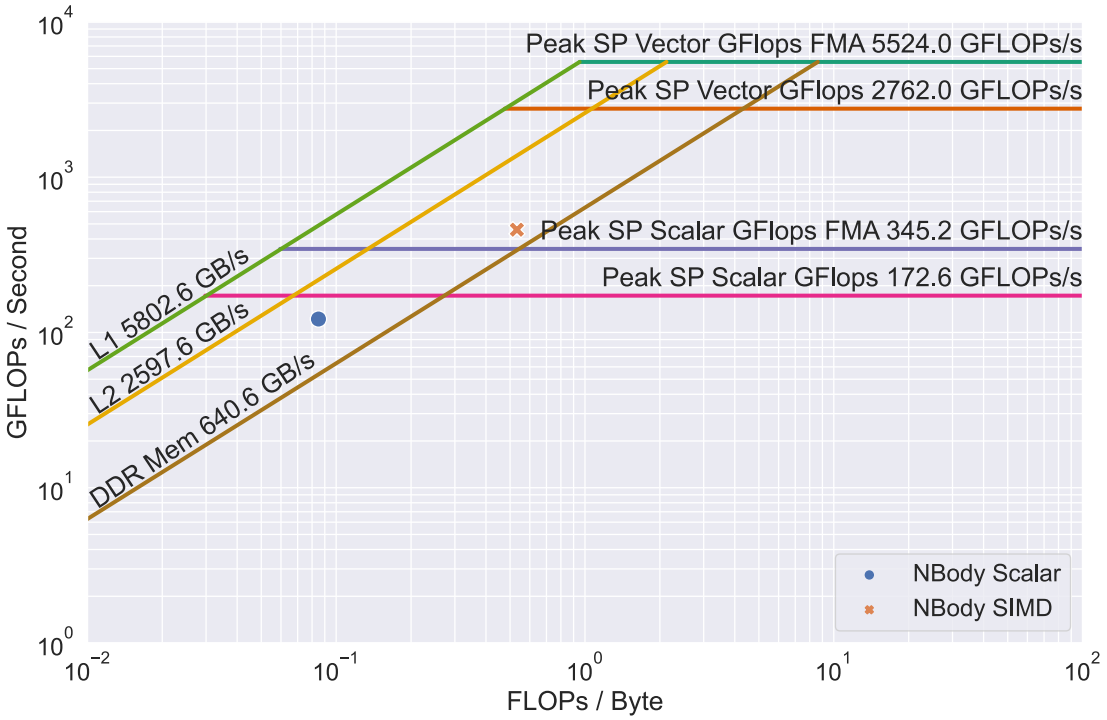

Figure 5. An example roofline plot displaying multiple cache levels (diagonal lines), multiple peak flop rates (horizontal lines), and some example application data points (dots) from [47]. Once an architectural roofline is created (lines), application points from functions or kernels could be plotted as soon data becomes available, which could integrate well into an *in situ* visualization workflow.

and APEX form a powerful combination (as the TAU Performance System) for HPC application performance analysis and engineering that is not replicated in other performance toolkits.

2.3.3 Rooflines: A Performance Analysis Success Story. The Roofline model is one of the most successful analysis and visualization concepts of hardware performance to come out of the literature in recent history. It has taken off like not many other concepts have, being incorporated into professional tools and used in the majority of papers discussing hardware performance. It's success is owed in part to the simplicity of the visualization, with a straightforward shape that is easily understood. Part of it's success is due to it's relatively portable and flexible nature, though some challenges still exist to be comparable across different architectures [46, 47]. The Cache-aware Roofline model (CARM) incorporates multiple cache levels [57]. Rooflines can also include multiple peak flop rates (roofs), for example, when looking at differences between scalar and vectorized peaks and/or enabling Fused Multiply Add (FMA). A Roofline model also lends itself well to customization,

allowing additional plotting of measured application performance or kernel performance in regards to the theoretical peaks.

The original roofline model was outlined by Williams et al. out of Berkeley Labs in [146]. First emerging in 2009, it was instrumental in assisting scientists moving to parallel programming from serial programming. The basic concept behind roofline analysis is that it calculates and plots GFLOPs per second on the y-axis, and Arithmetic Intensity on the x-axis. Arithmetic intensity is defined as the total number of FLOPs computed, divided by the total number of bytes transferred between the processor and the specified memory level. Plotted on a log-log scale, this creates the visual of a sloped roof, connected to a flat roof, indicating the peak values achievable on this particular hardware. The ability to evaluate how a kernel is performing based on the limits of the hardware was a key for scientists porting their code to new hardware, or improving their code on existing hardware. It was now easier than ever to answer questions about performance and work towards greater optimization of codes.

Ilic et al. made some extremely useful updates to the original Roofline model when they introduced the Cache-aware Roofline model (CARM). They take into account the different cache levels, not just the DRAM, and thus the different bandwidth measurements for each of these cache levels. The bandwidth to the L1 cache is higher than the bandwidth to DRAM and thus there is now opportunity for understanding performance within the two bounds. The result is an improvement on the older model, and has been widely adopted, often replacing the original Roofline model.

Intel Advisor is one of the tools that has incorporated the CARM into it's suite of offerings [89, 90], see table 2 in the "Architecture (Roofline)" column. Expanding and building on the models proposed in [58] which focus on understanding the memory access can affect the energy efficiency. Intel Advisor's capabilities extend to plotting the kernels on the graphical CARM, and whether those kernels are vectorized. Intel Advsior has made it more straightforward to profile an application and generate a graphical CARM bringing the concept to the forefront. It is easier, especially for domain scientists who do not have the time or expertise to dig into the specifics of the machine and application, to create these results and visualizations and understand the performance of their code.

The Roofline Model tools have been extended to support GPU architectures as well, using the same Roofline concept [153, 152, 143]. This is another example of how flexible and portable the roofline concept is, that without significant changes to the approach and visualization, it can be

extended to different architectures. The extension to GPUs from CPUs allows for far more robust comparisons of application performance. With the increasing use of GPUs to accelerate code it is crucial to be able to characterize application performance on these architectures.

A recent extension includes the Instruction Roofline Model, which deviates from the traditional floating point metric approach, and instead focuses on representing integer operations. The Instruction Roofline Model incorporates the number of instructions issued, and can help to highlight bottlenecks, instead of just overall performance. A couple of Instruction Roofline Models have recently been proposed for both NVIDIA and AMD GPU architectures [34, 80]. In general the Instruction Roofline Model allows for deeper insight into the memory performance, enabling identification of issues with shared memory conflicts and memory access patterns.

2.4 Performance Observation, Monitoring, and Analysis

Concurrent with the significant research and development work in the performance tools community, there has been long-term interest in parallel performance monitoring. There are numerous systems that can help with gathering the data required to conduct performance analysis online, but many opportunities remain [108]. Wood also provides an excellent survey in the area of online monitoring [149]. Analysis and visualization of this data ranges in these systems from very basic text/table output to more comprehensive graphs. The term “monalytics" was used as a combination of the terms “monitoring" and “analysis", referencing the approach for detecting and managing system and applications behaviors in a data center [75].

2.4.1 Application Monitoring. One example of a performace monitoring tool is SosFlow [151]. SosFlow shows how the *Scalable Observation System* (SOS) can collect low-level data from instrumented software for use in analysis. SOSFlow supports complex scientific workflows running on clusters by implementing an integration with TAU (called TAUflow) which intermittently submits the regularly collected TAU data to SOSflow. This is a useful integration as it makes use of existing tools for collecting performance metrics and couples it with an in situ analysis framework. They evaluated the overhead for this extra processing in the general range of 1%-3% of the total walltime of the application. Visualization of this data was extended using Alpine, mapping the performance data to the geometry of simulation data [150].

Tools like Falcon [50], Autopilot [111], Periscope [43], ActiveHarmony [127], and WOWMON [159] utilized monitoring to provide online analysis and/or support for adapting and

steering the application. Chimbuko [69], utilizing TAU as a performance measurement system, implemented in situ trace analysis to detect performance anomalies and generate provenance for root cause analysis.

The Falcon system provides application-specific monitoring, information about the overhead of the current monitoring, as well as graphical monitoring views [51]. This is useful in steering the application towards better performance. The visualization capabilities are relatively basic and two-dimensional, and some custom work was done for their evaluation to be able to create useful visualizations in a short enough period to enable on-line analysis and steering. While this is a promising start, realistically, scientists cannot be expected to do custom visualization work for each application they wish to monitor and analyze online. Additionally, there is a recent python-based tool [68] that offers a framework for flexible performance data input for HPC applications and has its own python-based visualizations as output.

DIMVisual Hierarchical Collection Model (DIMVHCM) is perhaps the most visualization focused of recent monitoring tools. Two major goals were to visualize the behavior of large scale parallel programs as well as collect this data in an on-line manner. DIMVHCM consists of three different types of data collectors and a push mechanism, e.g. data sent when certain conditions are met [135]. The system includes DimVisual [117], which aggregates the data for the visualization component TRIVA [116]. However, in order to actually run the graphical interface *in situ* they were required to implement a workaround client that integrated with TRIVA based on timestamps of the data. The ultimate effect was essentially *in situ* graphical monitoring, but with perhaps too many required steps and workarounds for mass adoption. While TRIVA is capable of some interesting distributed memory visualization, it is not clear what was taken advantage of in the DIMVHCM case studies.

OSU INAM is a tool that provides online introspection into application performance through a visualization dashboard, but it focuses on monitoring Infiniband network traffic as it relates to the MPI communication between nodes [74, 126]. Some of these metrics overlap with what we can collect from our TAU plugin, i.e. MPI message size, but they include more network specific metrics whereas we prioritize hardware and workflow states in this work.

Grafana is an open-source and enterprise tool that offers a dashboard of customizable analysis and visualizations for a number of different data types, including trace data. This is often used as

a monitoring dashboard for users to understand the overall health and performance of their system. Users can build visualizations that make sense for their system, including bar charts for categorical data and heatmaps. As their interface is an API, it is already used in an *in situ* fashion, with live updates as changes happen on the system [129]. Another tool along the lines of Grafana is DataDog, but DataDog has emerged as a commercial cloud monitoring interface. Online monitoring for companies who run their technology on cloud infrastructure is becoming increasingly populat and DataDog provides customizable analysis and visualization dashboards for such companies.

2.4.2 Workflow Monitoring. As HPC workloads have evolved to include complex heterogeneous workflows [62, 20], the need for online monitoring has grown in importance. Prior work for HPC workflow monitoring has enabled user-based workflow steering [93], integration of machine-level data into workflow monitoring [124], integration of ML-based techniques into distributed workflows for minimizing resource wastage [14], and specialized workflow monitoring systems to detect execution anomalies [88].

Bader et al. [13] present an architectural blueprint for categorizing monitoring data from the HPC workflow and application stacks, closely resembling the concept of namespaces implemented by SOMA. SYMBIOMON [110] introduces a service-based monitoring infrastructure for coupled HPC applications. SOMA represents a generalized design and data model for service-based monitoring of heterogeneous HPC workflows. In particular, SOMA's implementation and architecture are specially geared towards *real-time online monitoring* to enable adaptive execution of the workflow.

2.4.3 System-Level Monitoring. Much of the monitoring of HPC applications has historically been limited to "system-level" monitoring — typically available to HPC system administrators to view job-level statistics and hardware usage over time. The focus here is on the behavior and health of the entire cluster, and not of a specific application performance and/or how to improve it. Examples of such monitoring services include LDMS [2], Ganglia [91], Nagios [67], and XDMoD [124]. ZeroSum [55] is special in this regard, designed to operate in between the application and system layer and focused on optimizing the environment configuration of the HPC application. Typically, such services operate in the background (deployed as daemons on the compute nodes) and lie outside the scope of the HPC application — both in terms of online, real-time access to the monitoring data and the configuration of the monitoring service itself.

Recent work that has made use of the LDMS monitoring data by employing active learning-based frameworks to diagnose performance anomalies during runtime include [6, 7], and they focus on reducing the amount of labeled data required for accurate diagnostics. We envision that work in this vein could be a future integration with our SOMA framework, as a consumer and analyzer of the performance metrics in order to improve online decision-making, especially at scale.

Ganglia has interesting visualization capabilities though, specifically, it integrates with RRDtool (Round Robin Database), a circular database, [33] to visualize the time series data that is collected. The final output is web-based and separated from the performance data, which allows for customization of the visualizations without accidentally manipulating the collected data. While this is interesting and necessary for understanding the full context of application performance, it is not the granularity of information needed for on-line tuning of specific scientific codes.

TACC Stats is similarly focused on a full data center, and can offer insights such as when an application has idle nodes. [39] Some plotting functionality is included with some optional scripts/workflows that are designed to be run on a predetermined intermittent basis, not based on any conditional fulfillment. These plots are useful, but not able to be customized, two-dimensional, or interactive.

2.5 Neighboring Fields

2.5.1 *In Situ* Scientific Analysis and Visualization. The analysis and visualization of scientific simulation data has become increasingly challenging area as the volume of data grows. Scientists in this field face similar issues to those in performance data, with large amounts of complex data that needs to be managed and presented in a way that is digestible by the end user. Thus, the demand for *in situ* functionality grows, either so that a user or program can draw conclusions or even sometimes “steer" the application based on these results. One such example of a response to handling this data is with VisIt [30, 3, 145]. VisIt's focus is delivering scientific visualization capabilities for large datasets that have been generated on parallel clusters.

Numerous *in situ* scientific analysis visualization frameworks have emerged in recent years. Paraview focuses on generating interactive and exploratory scientific visualizations for large scale datasets [10, 4]. Paraview Catalyst has been incorporated into *in situ* scientific work flows as the visualization component in [5]. Alpine is an *in situ* scientific visualization infrastructure built upon

the Strawman prototype [86, 77]. The ALPINE API uses VTK-h, Flow, and Ascent to generate in situ data analysis and visualizations.

One key difference between the two fields that may be to the advantage of performance visualization is the fact that it may be more plausible to ignore, or lose performance data. Many scientific applications must be paused in order for the scientific visualizations to be generated, because if the simulation were allowed to continue on then important discoveries could be overlooked or missed completely. However, because of the nature of performance issues, it is much less likely that any such insight is only able to be made at a single point in the application. This opens the door to the possibility of reducing the overhead seen by some of these *in situ* scientific visualizations, and allowing the simulation to continue while performance metrics are analyzed and produce results.

2.5.2 Exploratory Information Visualization. Exploratory information visualization or visual analysis is the ability for a user to interact with their data and visualizations in order to guide what is being analyzed and visualized and uncover new insights. A user may start with a general hypothesis, and refine the visualization until they have a solution, or uncovered a problem. Or a user may have an "open ended" approach, without a specific goal in mind. Either way, exploratory visual analysis is a crucial component of analysis and visualization research, yet because of these differing goals, can be difficult to implement well [16]. The domain of the data, and goals of the user may be key in producing an effective tool here.

The increase of big data collected by businesses has given rise to tools like Tableau which can provides insights into their data through exploratory visual analysis [125, 92, 133]. While business data is not the exact same as HPC data, there can be multi-dimensional and temporal qualities, i.e., number of units of each item sold, across different types of stores, in different regions, within certain time periods [92]. Tableau's main approach is to enable exploratory visual analysis via clickable dashboards that can change what data is viewed to answer different questions, and remove the need for programming analysis written in Python or R, see Figure 6 [125]. Their VizQL solution enables the majority of their interactive visualization functionality, allowing the user to learn with feedback from the visualization, and vice versa [133]. Applying similar technology at this scale in the performance data domain would be groundbreaking.

Other research projects looking at visual analytics of high-dimensional data can provide other perspectives. For example, mapping multi-variate data to a concept that people already understand,

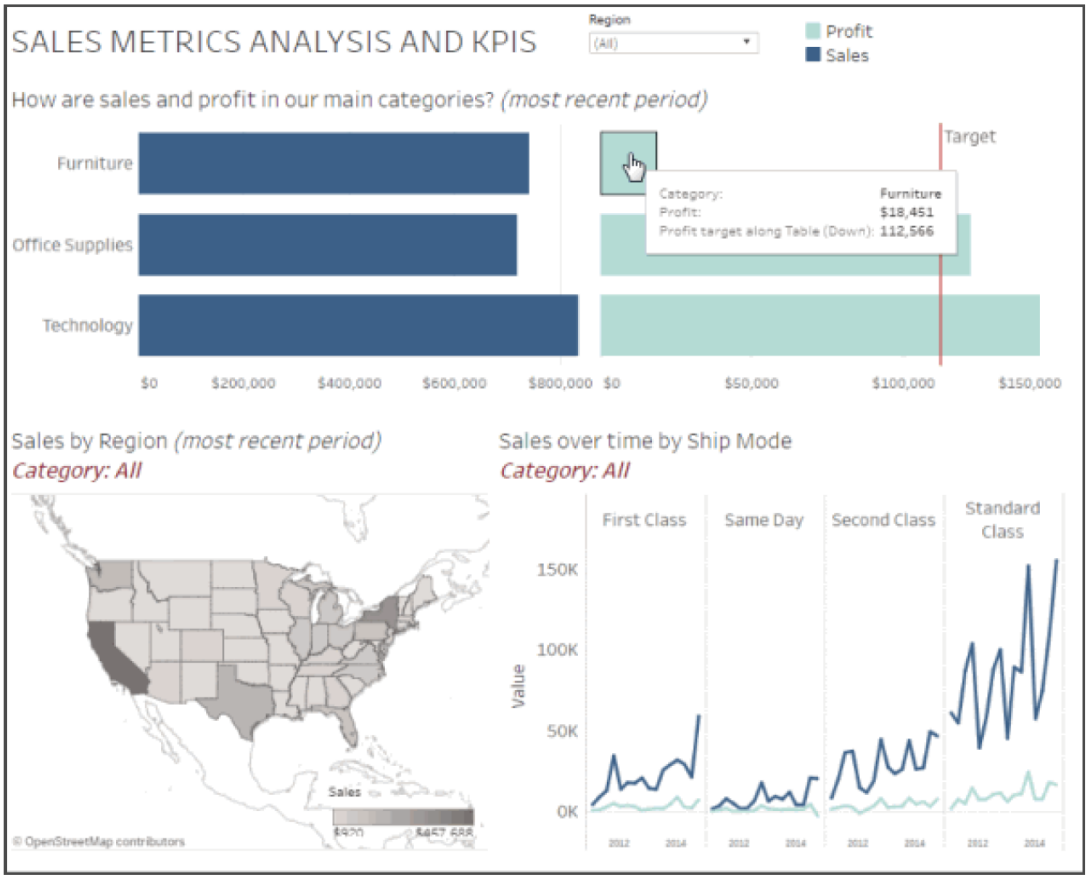

Figure 6. An example of an interactive, clickable dashboard in Tableau, that enables a user to change the data they are viewing [92]. The data behind these visualizations could be considered *post hoc*, as it is gathered from a previously completed time period, yet is made interactive for exploratory purposes.

in this case, interactive route planning [160]. This creates a visualization that the user can understand and interact with more readily from data that might otherwise be unmanageable. Another project explores the relationship between the data analyst and a machine learning model of the data, allowing the analyst to interact and refine the model. At the same time the model learns to identify relationships and patterns in the data, which may be difficult for a human to identify. This project supports “both model-driven data exploration, as well as data-driven model evolution" [42].

2.6 Requirements for Effective Performance Observation

With the rise of exascale systems and large, data-centric workflows, the need to observe and analyze high performance computing (HPC) applications during their execution is becoming increasingly important. While there is growing interest in enabling greater “observability" of HPC applications, they are typically not designed with online monitoring in mind. Most tools are tightly integrated with the application’s implementation, reducing their flexibility, extendibility, and portability. When new functionality, such as runtime performance monitoring or in situ data analytics is required, approaches are often severely constrained by how the HPC application is implemented and executed. Therefore, the observability challenge lies in being able to access and analyze interesting events with low overhead while seamlessly integrating such capabilities into existing and new applications. Furthermore, the rise of scalable heterogeneous HPC systems is increasing execution dynamics and the need for improved observational awareness. In particular, the ability to track, analyze, and interrogate interesting events and phenomena about the application and system is important to support the next generation of exascale solutions. We explore how SOMA’s approach to collecting and aggregating both application-specific diagnostic data and performance data addresses the following identified needs.

2.6.1 Configurable. The diversity in the architecture of HPC clusters has been very apparent over the years as evidenced by lists such as the TOP500 [36]. Differences span from physical layout of chips, number of cores per CPU, memory bandwidth, number of cache levels, GPU accelerators, as just a small fraction of ways they can vary. More recently, with the rise of larger artificial intelligence (AI) models, hardware has continued to evolve beyond general purpose computing, and instead to better support the specific needs of AI. In order to create an effective framework that can run on many or all of these we need to answer the question “How can we run on heterogeneous HPC ecosystems?" In other words, what are the requirements for

configurability that must be taken into account. Previous monitoring work has focused on the architecture design and high-performance implementation, specifically to access TAU performance data. TAUoverSupermon [100], TAUoverMRNet [99], TAUmon [78], and SOS [151] explored different approaches and technologies for scalable application-level monitoring. Our recent research on SYMBIOMON [110] instead chose to build upon the Mochi high-performance microservices framework [114] thereby adopting a existing development model with well-defined interfaces, available components, and active users. The modular, microservice-based architecture of Mochi enables the configurability of SOMA [108]. Seer [49], SERVIZ [109], Colza [37], and other projects have taken this route to build in situ analysis, visualization, and autotuning.

2.6.1.1 Portability. The crucial difficulty facing all consumers of performance analysis and visualizations is the ability to use tools or techniques across heterogeneous architectures. A scientist will find themselves faced with the dilemma of running their codes on a new architecture anytime they gain access to upgraded equipment, or explore other options for acceleration. It is very useful to understand performance across different CPU architectures, or analyze their application's performance with the addition of GPU acceleration. Vendors will often build analysis and visualization tools to support their own architecture. Open-source tools like TAU, that use standard formats such as OTF2 attempt to bridge this gap, yet many vendor-specific tools remain that do not work across architectures. Considering the differences in the physical hardware and network components (the interconnect between compute nodes), on modern HPC systems we must address the challenge of building a portable SOMA framework. Our approach to addressing this is discussed further in Chapter III, but involves making use of underlying microservice technologies (Mochi [114]) that provide some measure of portability for us already.

2.6.1.2 Other Configurations. Harvesting unused computing cycles has been explored in the context of inline visualization and analytics. The TINS [35] package leverages work-stealing strategies to execute analytics tasks when there are no available simulation tasks scheduled. GoldRush [161] and Landrush [45] employ smart co-scheduling of analytics routines alongside MPI-OpenMP and GPU simulation tasks. They combine monitoring data with a scheduler to identify regions of idle time on the processor that can be used to run these routines demonstrating significant cost savings without perturbing the execution of the simulation. Our approach is not nearly as sophisticated as these research results, but nevertheless attempts to take advantage of a situation

with free CPU resources via either a straightforward MPI-based strategy, or as part of a scheduled workflow. This is highlighted further in Chapters III and V.

2.6.2 Flexible. In addition to running across heterogeneous HPC ecosystems, SOMA needs to support heterogeneous scientific applications and performance measurement and analysis tools. The question “How can we support different input and output?" must be answered. The SOMA research makes strides in this direction with the additional incorporation of data models which are necessary for the semantic communication of application and performance data between microservices. Conduit [52] is an established project born out of the scientific visualization community for the purpose of describing and sharing data in situ.

As evidenced by the plethora of different performance tools that have been introduced in this chapter, it would be a weakness to prescribe the use of a specific measurement tool or output visualization in order to use SOMA. An application developer may not have the time or knowledge to re-instrument their codes to use a new tool, or they may be limited to using less invasive sampling methods for their codes. They may be more interested in memory bandwidth bottlenecks as opposed to network bottlenecks, or be looking for suggestions to optimize specific kernels. Many of the tools that support these measurements already exist - SOMA’s approach is to make use of these existing tools and offer support for plugging into any of them. Similarly, if there are specific backends (analysis or visualization tools) that an application development team already prefers, SOMA should be flexible enough to support an integration with those. So as to provide *flexible* support for data interpretation to any input and output SOMA implements a canonical data model, which is detailed further in Chapter IV. One of SOMA’s greatest strengths is that it does not prescribe a “One tool fits all" solution for the diversity of HPC applications and performance tools.

2.6.3 Performant. Our interest in HPC observability is motivated by the problem of large-scale performance monitoring and analysis. Specifically, given robust technology for heterogeneous performance measurement (e.g., *TAU Performance System* ® [84], HPCToolkit [1], and CALIPER [23]), how can real-time access to performance data and its in situ processing (e.g., to identify runtime performance issues and possibly feedback actionable results) be realized, with minimal impact on the application and efficiency? While building performance monitoring and analysis technology that can seamlessly integrate with an HPC application is a challenge, the

objectives are not unique intrinsically, as they are shared with other domains such as simulation data analytics and visualization.

The challenges of wrangling performance data have been introduced in this chapter, a summary of these challenges was given in Table 1. From a performance perspective - moving analysis online helps reduce the volume of data, but we are still generating and sending additional data, possibly distributed across many compute nodes. Some overhead is unavoidable, but we must consider "How do we minimize the overhead?" by first measuring it, and then making improvements. Due to the diversity of HPC applications and ecosystems, what helps in one situation may not help in another depending on where the bottlenecks exist. In Chapter V we explore different approaches to reducing the observation overhead of the SOMA framework for a variety of different applications or workflows.

2.7 Summary

Chapter II provided the background and introduction required for understanding and motivating the rest of this dissertation. Section 2.2 began by offering some definitions of terms in the performance measurement domain. Section 2.3 provided a comprehensive outlook on current performance analysis tools and their offline capabilities. Section 2.4 described current advances in the area of online performance observation, monitoring, and analytics. Section 2.5 offered a glimpse into two of the neighboring fields and how we might draw some inspiration from them. Ultimately, Section 2.6 presented a comprehensive view of how all the previous sections led us to identification of the requirements for SOMA.

CHAPTER III
A MICROSERVICE APPROACH

This chapter contains previously published and unpublished material with co-authorship. Sections 3.2 and 3.3 contain work from a paper initially published at the Cray User Group conference in 2023. An extension of this paper was published in a special edition journal of Concurrency and Computation: Practice and Experience in June 2024. The work was a collaboration between the University of Oregon, NVIDIA Corporation, University of Helsinki, Aalto University, and Academia Sinica. Co-authors include Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Boyana Norris, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, and Dr. Allen D. Malony. I was the first author for both publications, conducted all SOMA-related experiments and completed the majority of the SOMA-related writing. Some co-authors helped with writing and figures, all helped with suggestions, and proof-reading.

Section 3.4.2 contains example code from the above project, and also contains example code from an unpublished research collaboration between University of Oregon and Lawrence Livermore National Laboratory. I implemented the integration between SOMA and Caliper, conducted all experiments, created all figures, and wrote all sections with guidance from Dr. David Böhme.

Section 3.4 and section 3.5 contain material from an unpublished paper (under review at a 2024 conference). This work was a collaboration between University of Oregon, NVIDIA Corporation, Brookhaven National Laboratory, and Rutgers University. I was the first author, conducted all experiments and completed the majority of the writing. Co-authors include Dr. Mikhail Titov, Dr. Srinivasan Ramesh, Dr. Ozgur Kilic, Dr. Matteo Turilli, Dr. Shantenu Jha, and Dr. Allen D. Malony. The background sections on RADICAL-Pilot, Section 3.5.2 and Section 3.5.3.1, were written primarily by the RADICAL-Pilot co-authors, especially Dr. Matteo Turilli. All co-authors helped with suggestions, and proof-reading.

3.1 Introduction

The background laid out in Chapter II drove many of our decisions for how to design our service-based observability, monitoring and analytics (SOMA) framework. In order to answer the proposed research questions and meet the identified requirements of *configurable*, *flexible*, and *performant* we had to make very intentional design choices. This chapter focuses on how we addressed

the need for *configurability* by choosing a microservice architecture, and designing a useful API. With this architecture design, we answer the research question “How can we run on heterogeneous ecosystems?" The organization of this chapter is described here for convenience. Section 3.2 describes the underlying technologies used to build SOMA and how the software architecture is structured. Section 3.3 details the SOMA application programming interface (API). Section 3.4 describes how to integrate SOMA with applications and performance tools in order to monitor data, and shows examples of how a SOMA client can be implemented. Finally, Section 3.5 depicts how SOMA was integrated with RADICAL-Pilot to monitor heterogeneous workflows, as workflows are becoming increasingly popular in HPC environments.

3.2 Technology and Architecture

SOMA is service-based software, this means that it has well-defined functionality specified by an API, that can be deployed in modular pieces. Additionally, as SOMA executables provide single-task oriented functionality — that of making specific performance data available online – it is considered to provide microservice capabilities. One advantage over a more monolithic system is that only the specific functionality that is needed can be activated and used for any given scenario. This is especially helpful when running on bleeding-edge HPC ecosystems where deploying complex codes can more quickly become problematic.

3.2.1 Microservices. Being able to create monitoring processes alongside the HPC application and place them within the resource allocation (even on additional resources) is a first step. The question then becomes what code is being run on the monitoring processes and how do they interact with the application. We can think of a performance monitor as a coupled data service to an application for the purposes of capturing and processing performance information. HPC data services have emerged as an essential component of coupled HPC workflow architectures. Mochi [114] is a software stack for developing data services built by composing individual microservices through the remote procedure call (RPC) as the communication mechanism. A *client* instance is the *origin* of the RPC and the *service provider* instance is the *target*. By providing a set of microservice building blocks, necessary tools, and a development environment, the Mochi framework enables the rapid development of customized functionality.

Figure 7 depicts three microservices (A, B, and C) interacting through RPC calls to generate different *call paths* through the network. These microservices can be located on the same process, on

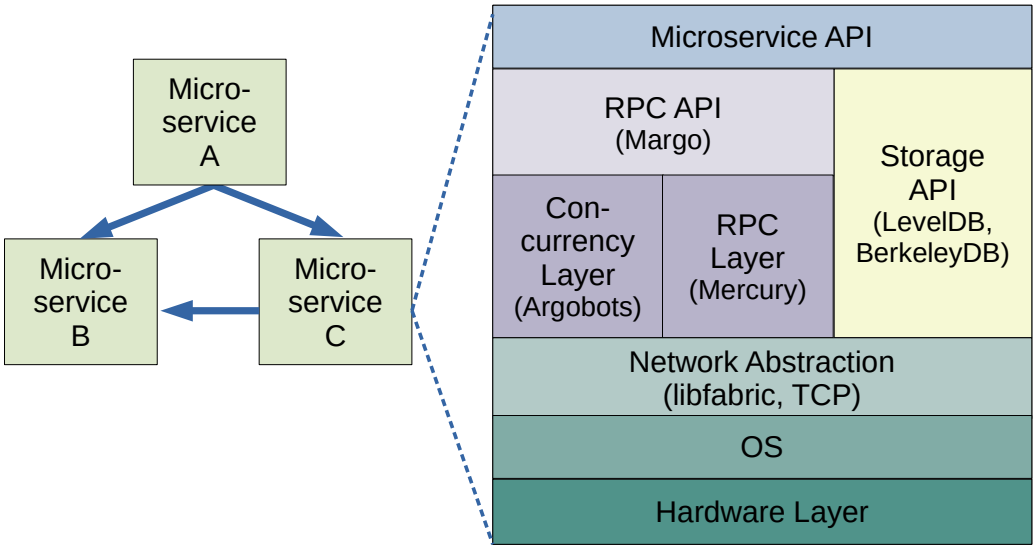

Figure 7. Mochi microservice stack and example microservice.

different processes within the same computing node, or on completely different nodes depending on how they are configured. The Mochi microservice software stack enabling this interaction consists of five core components: Mercury RPC library, Argobots, Margo, Thallium, and Scalable Service Groups. The first four are highlighted here:

Mercury A high-performance RPC library that can utilize remote data memory access (RDMA) capabilities to transfer large remote procedure call (RPC) arguments efficiently [123].

Argobots A lightweight user-level threading library that enables the development of highly concurrent software components [119].

Margo The Margo C library provides a convenient abstraction that hides the complexities of programming the callback-driven Mercury library [114].

Thallium A header-only, C++ interface to Margo and is provided as a convenient wrapper to ease programming with Mochi [114].

These technologies offer underlying structure to the SOMA microservices that is portable across heterogeneous hardware. This creates a huge advantage over building the entire software stack from scratch, which would require expertise across the many different systems, instead of a focus on collecting and making performance data available. With only some configuration changes

Supercomputer	LUMI-G	Mahti	Summit	Quartz
Organization	CSC	CSC	ORNL	LLNL
CPU Architecture	AMD EPYC 7653	AMD Rome 7H12	IBM Power9	Intel Xeon E5-2695 v4
GPU Architecture	AMD MI250X	NVIDIA A100	NVIDIA V100	None
Network	* HPE Slingshot 11	Mellanox HDR Infiniband	Mellanox EDR 100G Infiniband	Cornelis Networks Omni-Path

Table 3. Systems where we have successfully run SOMA. This demonstrates configurability and portability of the framework. (*There was an issue with the HPE virtual network interface integration with Mercury at the time we ran these experiments so we resorted to the TCP interconnect, however, a fix from HPE is on the way.)

during compilation and runtime, the SOMA software can run on many different HPC systems, that have different network interconnects, different compute nodes, etc. Table 3 demonstrates four of the clusters where SOMA has successfully been deployed. The noted issue with the HPE Slingshot 11 network actually highlights a benefit of relying on Mercury, where the Mercury development team and user base was able to interface with HPE to request a fix, a feat that would be much more difficult in a one-off scenario. Additionally, by building upon existing state-of-the-art HPC software frameworks, SOMA can take direct and seamless advantage of the improvements made to these frameworks, while cleanly separating functionality and performance. This design choice arguably makes SOMA a more maintainable monitoring service than other *ad hoc* implementations.

3.3 SOMA API

Now that SOMA has a robust microservice architecture, we are left to define the application programming interface (API) — the core functionality. Table 4 depicts the entirety of the SOMA Collector API. The core API revolves around the idea of a *monitoring namespace*, building on the earlier SYMBIOMON implementation [110]. The creation of the namespace requires the user to supply a `string` argument representing the namespace name, following which an empty Conduit::Node is created inside the collector client's memory. More on the Conduit [52] technology will be described in Chapter IV, but it can be considered the data building block structure for SOMA. Following creation, the namespace can be updated by providing a `key:value` pair, wherein the `key` represents the hierarchy level of the numeric data (e.g., "TAU/MPI/MPI_Allreduce"), and the value is the numeric data to be stored. Code examples are provided in section 3.4.2. Note that the top level in the hierarchy is always the namespace name — this name is automatically prefixed to the `key` argument and is not required to be supplied by the calling code. If the key exists, the

Table 4. SOMA API Description

`soma_create_namespace` Creates a SOMA namespace and returns a handle to the namespace
`soma_update_namespace` Updates the SOMA namespace with hierarchical data in a key:value pair
`soma_publish` Publishes a raw Conduit::Node to the SOMA collector service instance, blocking call, void return
`soma_publish_async` Publishes a raw Conduit::Node to the SOMA collector service instance, non-blocking call, returns a response
`soma_publish_namespace` Publishes the Conduit::Node underlying the namespace
`soma_commit_namespace` Commits a namespace – akin to closing a file, blocking call
`soma_commit_namespace_async` Commits a namespace – non-blocking call returns a response
`soma_set_publish_frequency` Sets the monitoring frequency associated with a namespace
`soma_analyze` Instructs the collector service to analyze current Conduit::Node data and then write results to a file
`soma_write` Instructs the collector service to write Conduit::Node data to a file

value is either updated or appended to an existing list depending on the `operation_type` passed to the `soma_update_namespace` API call. If the key does not exist, a new Conduit::Leaf object is created.

A namespace that is updated is left in an *uncommitted* or *open* state until the client explicitly invokes `soma_commit_namespace`. Committing a namespace decrements a *frequency counter* associated with the namespace. When this frequency counter reaches zero, `soma_publish_namespace` is triggered internally, resulting in an RPC call to the collector service instance carrying the payload of the namespace — a Conduit::Node object representing the data being monitored. The frequency counter is set using the `soma_set_publish_frequency` API. The association of a monitoring frequency with a namespace in SOMA is an improvement over SYMBIOMON — the latter only allowed monitoring frequencies to be set on a per-metric basis, resulting in a flurry of RPCs in the system and a tendency for the monitoring system to be more network-latency-sensitive than necessary. SOMA also exposes `soma_publish`, an API to publish a raw Conduit::Node object directly to the collector instance. For both `soma_publish` and `soma_commit_namespace`, an asynchronous (non-blocking) version is also

available, `soma_publish_async` and `soma_commit_namespace_async`. Under the hood, these utilize the Mochi Thallium asynchronous RPC call and return a Thallium response which can be waited on to ensure completion. The `soma_analyze` function requests that the service conduct some online analysis of the current Conduit::Node data and write the results to a file. In the future the `soma_analyze` functionality could be moved into a consumer client or analyzer service with more robust features. In summary, the SOMA API encompasses all of SYMBIOMON's features, while simultaneously being simpler, offering better performance, and being more generally applicable for scientific and performance data monitoring alike.

3.4 Application Monitoring

Now that the architecture and API have been laid out, we describe how the SOMA software is used in the context of application monitoring. This includes how the client and server are intialized and interact as well as how a client can use the API in Section 3.4.1. A couple of examples of clients are provided as well in Section 3.4.2.

3.4.1 Implementation. The SOMA client is a stub that is linked against an application requiring SOMA monitoring capabilities. The SOMA service instance typically resides on a separate process and is contacted by means of an RPC call. The client-server abstraction allows SOMA to be configured in a variety of ways without modifying the client stub code. Figure 2 depicted the SOMA stack in the context of the Astaroth application. Figures 9 and 10 show a scenario where each node is executing a SOMA performance data service instance and a SOMA application data service instance *alongside* the application, i.e., SOMA service instances share the computing node resources with the application. SOMA can also be configured to run on a different node or nodes than the application without *any* changes to the application or client stub code, this is depicted in Figures 8 and 10. The only difference between these two scenarios would be the datapath for the RPC calls — the former would involve shared-memory copies on the node (supported by most operating systems) while the latter would traverse the system network links. As mentioned, Mochi microservices automatically support a variety of different communication "plugins" via Mercury — a few of which are `shared-memory`, `TCP`, and `verbs`. This plugin model allows us to switch between communication protocols through a simple configuration change, thereby circumventing the need for any code changes to the application client stub or SOMA service instance. Here we enlist the steps necessary for an application to establish a connection to SOMA and begin the monitoring workflow:

– **Generation of Server Addresses**: The SOMA MPI program is launched first, following which the division of the processes into the configured number of server instances takes place. Each process instantiates its SOMA service provider and makes its unique RPC address public through a file. SOMA implements *server instances* to support the scenarios depicted in Figures 9 and 8. One instance can monitor scientific application data from the application, while another instance can monitor performance data. Once the RPC addresses are written out to an *address file*, the service is now ready to accept client requests.

– **Service Discovery**: The application and performance measurement library connect to SOMA by reading in the RPC address through the *address file* and creating a client object to manage the connection. This logic is housed inside the initialization routines of the client software. If there are N SOMA service instance ranks and M application (or performance tool) client ranks such that M>N, our current implementation assigns the N SOMA ranks in a round-robin fashion to the M client ranks. Other client-server mapping strategies are also possible.

– **RPC Invocation**: Most SOMA API operations depicted in Table 4 are *local* operations, i.e., they execute directly inside the client stub memory. However, the `soma_publish` API results in a Mochi RPC call. When this RPC is invoked, the Conduit::Node underlying the SOMA namespace is serialized to a string representation using native Conduit routines. The resulting string representation is passed to Mercury, which serializes the string on the client, manages the data transfer through the chosen communication plugin, and de-serializes the data back into a string representation on the collector service instance. Serializing the Conduit::Node object to a string representation can be expensive if the Conduit::Node is large. In the future, we plan to explore binary representations as a way to reduce the RPC payload size, thereby improving performance.

– **RPC Execution**: The collector service instances can be effectively modeled as workers executing RPCs from their local work queue. When an RPC for `soma_publish` is executed on the server, the string representation is converted back into a Conduit::Node and stored inside an in-memory queue. This queue is emptied upon receipt of a `soma_write` call on the collector service instance.

```
1       case PeriodicAction::PublishToSOMA: {
2           log_from_root_proc_wth_sim_progress(pid,
3               "Periodic action: publishing data to SOMA\n");
4           conduit::Node appl_data_node =
5               query_local_diagnostics(pid, info, i, simulation_time);
6           soma_channel.soma_publish(appl_data_node);
7           break;
8       }
```

Listing 3.1 Snippet example of how Astaroth publishes diagnostics to SOMA

3.4.2 Examples. We provide some example code from two different SOMA clients to better illustrate a couple of different scenarios. Example 1 is from our application monitoring integration with the Astrophysics code Astaroth [128]. This example is the most straightforward use-case, but does not make use of some of the fine-grained control gained from creating a namespace and committing to the namespace. A code snippet from the Astaroth application is shown in Listing 3.1. We do not show the code that is required to initialize the SOMA client and connect to a server instance. However, once our client stub is intialized within the application that we are monitoring, we can use our client handle (the "soma_channel" object) in the Listing to make API calls. This is done within Astaroth at regular intervals, which is a configurable number of application timesteps (line 1). Astaroth collects the metrics that are of interest, structures it into a Conduit::Node (lines 4-5) and publishes the Conduit::Node to the SOMA server (line 6). More information about the Conduit::Node data structure for Astaroth can be found in Section 4.4.

Example 2 is a bit more complex, and makes use of the namespace and more controlled commit structure described in the API section. The code example is taken from an integration with the Caliper performance measurement library [23] where we can make the metrics measured by Caliper available online via a SOMA plugin. Details about how this integration works and the Conduit::Node data structure for Caliper will be discussed in Section 4.3.2. In Listing 3.2 we see the general structure of how the plugin within Caliper works. Again, initialization code is not shown, but that must be done before this code can execute. Lines 1-2 is the function signature of the function that is called when a "snapshot" of performance data is taken by Caliper. Lines 4-20 show the restructuring of

```
void write_soma_record(std::ostream& os, int mpi_rank,
    RegionProfile& profile, Caliper* c, SnapshotView rec)
{
    std::map<std::string, double> region_times;
    std::tie(region_times, std::ignore, total_time) =
        profile.exclusive_region_times();
    std::string timestamp = std::to_string(unix_timestamp);
    std::string time_rank_key = timestamp + "/" +
    std::to_string(mpi_rank);
    // Iterate through metrics and add to Conduit::node
    for (const auto &p : region_times) {
        soma_collector.soma_update_namespace(ns_handle,
            time_rank_key, p.first, p.second, soma::OVERWRITE); }
    // Get other metrics as part of the record
    if (!rec.empty()) {
        for (const Entry& e : rec) {
                std::string metric = c->get_attribute(
                    e.attribute()).name_c_str();
                std::string value = e.value().to_string();
                soma_collector.soma_update_namespace(ns_handle,
                    time_rank_key, metric, value, soma::OVERWRITE);
    }}}
    auto response = soma_collector.soma_commit_namespace_async(
        ns_handle);
    if (response) {
        requests.push_back(*std::move(response));
}}
```

Listing 3.2 Code example of how the Caliper plugin publishes performance data to SOMA

the Caliper performance metrics into the Conduit::Node data structure and updating the namespace (i.e., appending or overwriting new data to the Conduit::Node), which is a local function call. On lines 23-24 we "commit" the namespace, which will trigger the first remote procedure call to publish the data to the SOMA server asynchronously. How often the remote publishes actually execute (publishing frequency) is based on a chosen configuration. Finally, in lines 25-26, the asynchronous responses are stored in a data structure so that they can be checked for successful completion upon shutdown.

3.5 Workflow Monitoring

HPC workflows require the capture of data from different sources across the workflow and application software stacks to enable observability. Not only are their storage needs different, but data from these sources may need to be monitored appropriately, directly impacting their computational needs within the SOMA service. In response, SOMA takes a leaf out of SERVIZ [109], a workflow-ready visualization service, to enable the partitioning of monitoring service resources for a single namespace into one or more independent "instances", each of which is responsible for monitoring data from one or more source. While a largely similar approach to application monitoring, some special considerations were required and Section 3.5.1 describes how we configure and run SOMA for workflow monitoring. Section 3.5.2 gives details on the RADICAL-Pilot pilot system for managing heterogeneous HPC workflows. Then, Section 3.5.3 describes how SOMA and RADICAL-Pilot are integrated in order to successfully monitor heterogeneous HPC workflows.

3.5.1 SOMA Configurations for Workflows. SOMA's client stub can run within the address space of the component being instrumented (application or middleware) and require no additional computational resources to execute, i.e., the application's main thread is used to drive the progress of the RPC calls. Alternatively, the client stub can also be implemented within a separate, standalone executable that does run on additional resources. This is beneficial when collecting metrics that do not require application instrumentation, such as hardware metrics. Both types of SOMA clients are demonstrated in this dissertation.

Typically, the SOMA service executes on a set of dedicated resources *outside* the application or workflow component being monitored. This clean separation between the client and service libraries allows significant flexibility in determining where SOMA's service instances execute while being completely transparent to the calling client. Previous work [157, 158] has explored the benefits

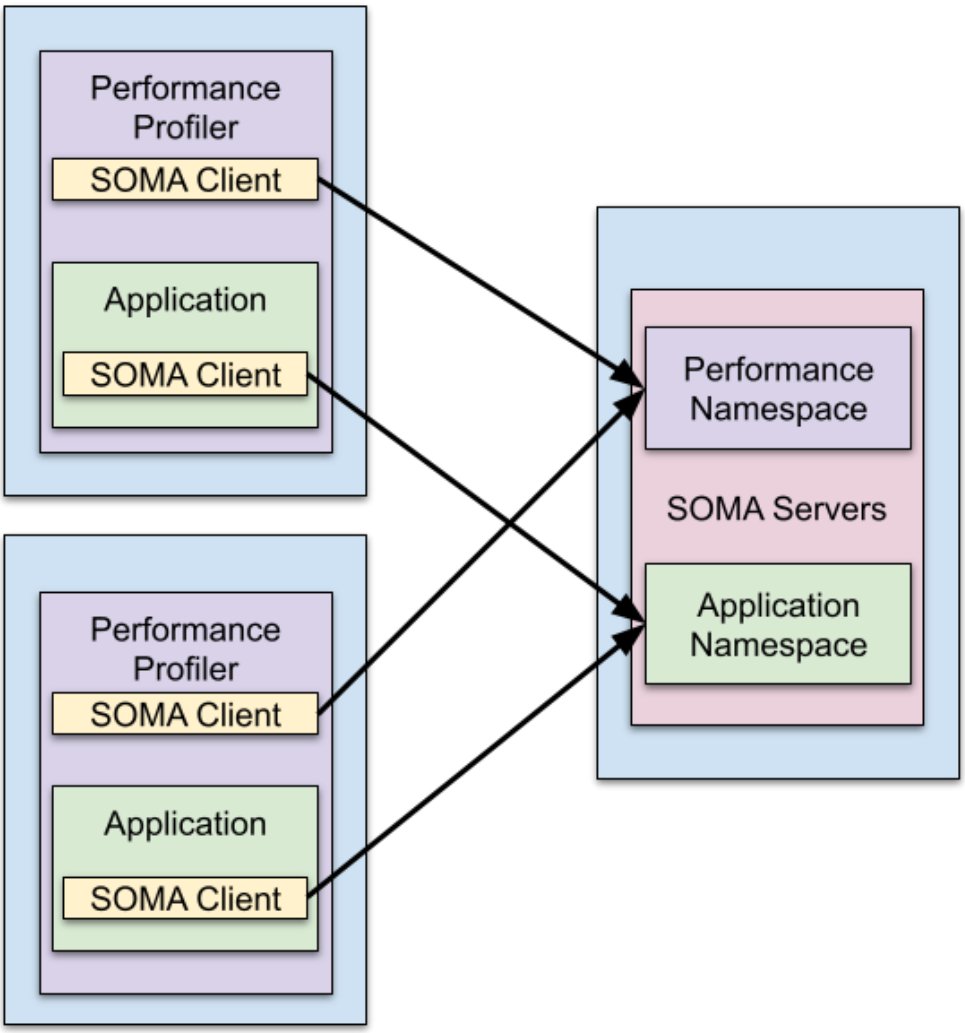

Figure 8. One typical layout of the client and service structure of SOMA, with client stubs in the address space of the application and performance profiler and services on separate, reserved resources. See the legend in Figure 10.

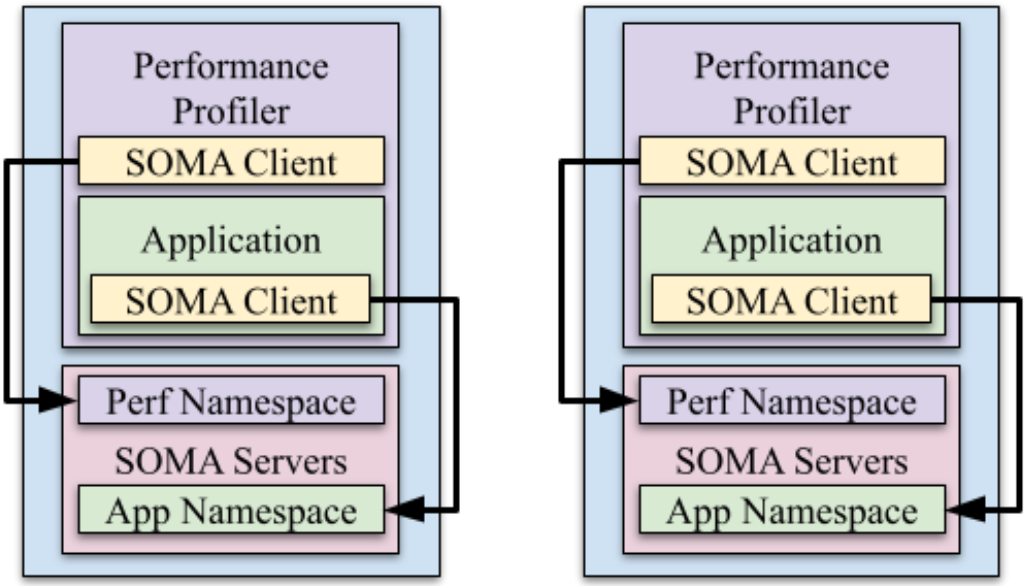

Figure 9. Another typical layout of the client and service structure of SOMA, with client stubs in the address space of the application and performance profiler and services shared, local resources. See the legend in Figure 10.

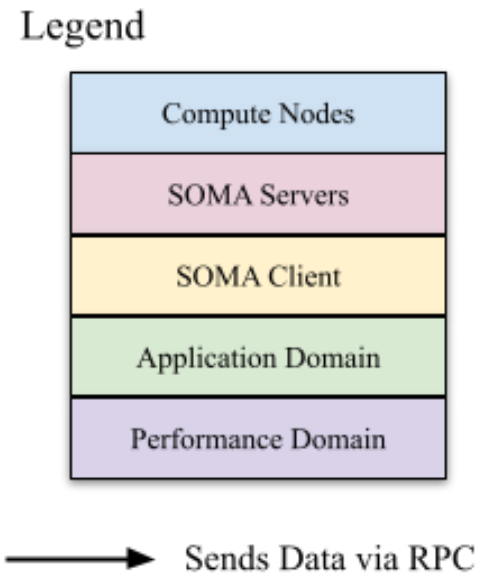

Figure 10. A legend describing the components of the figures above.

of running SOMA's service instances on the unused cores within the application's compute node. However, for HPC workflows, we choose to dedicate a part of the workflow's total computing resources to running the SOMA service instances.

3.5.2 RADICAL-Pilot. RADICAL-Pilot [96] is a Python implementation of the pilot paradigm and architectural pattern [141, 82]. Pilot systems enable users to submit jobs to HPC platforms and then use those resources to execute the application's tasks. Those tasks are directly scheduled via the pilot-system without queueing in the platform's batch system. In that way, it is possible to achieve high-throughput task execution on HPC, avoiding the limits imposed by a centralized, multi-tenant batch queue [141]

Distinctively, RADICAL-Pilot supports executing heterogeneous executable or function tasks on HPC resources. Both types of tasks can be single/multi-core/GPU/node and MPI/OpenMP. Executable tasks are programs that run as self-contained entities, while function tasks are functions or methods written in a specific programming language. Currently, RADICAL-Pilot utilizes a dedicated subsystem called RAPTOR [95] to execute Python functions at a very large scale. Uniquely, RADICAL-Pilot supports the concurrent execution of heterogeneous executable and Python function tasks on up to 193,000 cores and 27,600 GPUs [98].

RADICAL-Pilot implements two abstractions: Pilot and Task. Pilots are placeholders for computing resources, where resources are represented independently of architecture and platform details. Tasks are units of work specified by a program's executable or a language-specific function/method, alongside resource and execution environment requirements. Fig. 11 depicts RADICAL-Pilot's architecture with two subsystems (white boxes), each with several components (purple and yellow boxes). Purple components manage pilots and tasks, while yellow components enable communication and coordination. Subsystems can execute locally or remotely, communicating over TCP/IP and enabling multiple deployment scenarios.

Numbers in Fig. 11 show the resource acquisition and task execution processes. PilotManager uses PSI/J [53] to queue a pilot as a job on an HPC platform's batch system (Fig. 11 ①–②). Once scheduled, the job bootstraps RADICAL-Pilot's Agent and the Agent's Updater notifies RADICAL-Pilot's Client that tasks can be executed (Fig. 11 ③). Upon notification, the client's TaskManager queues all the available tasks onto the client's Scheduler and, after staging files when required, tasks are queued to the Agent's Scheduler (Fig. 11 ④–⑥). The Agent's scheduler assigns tasks

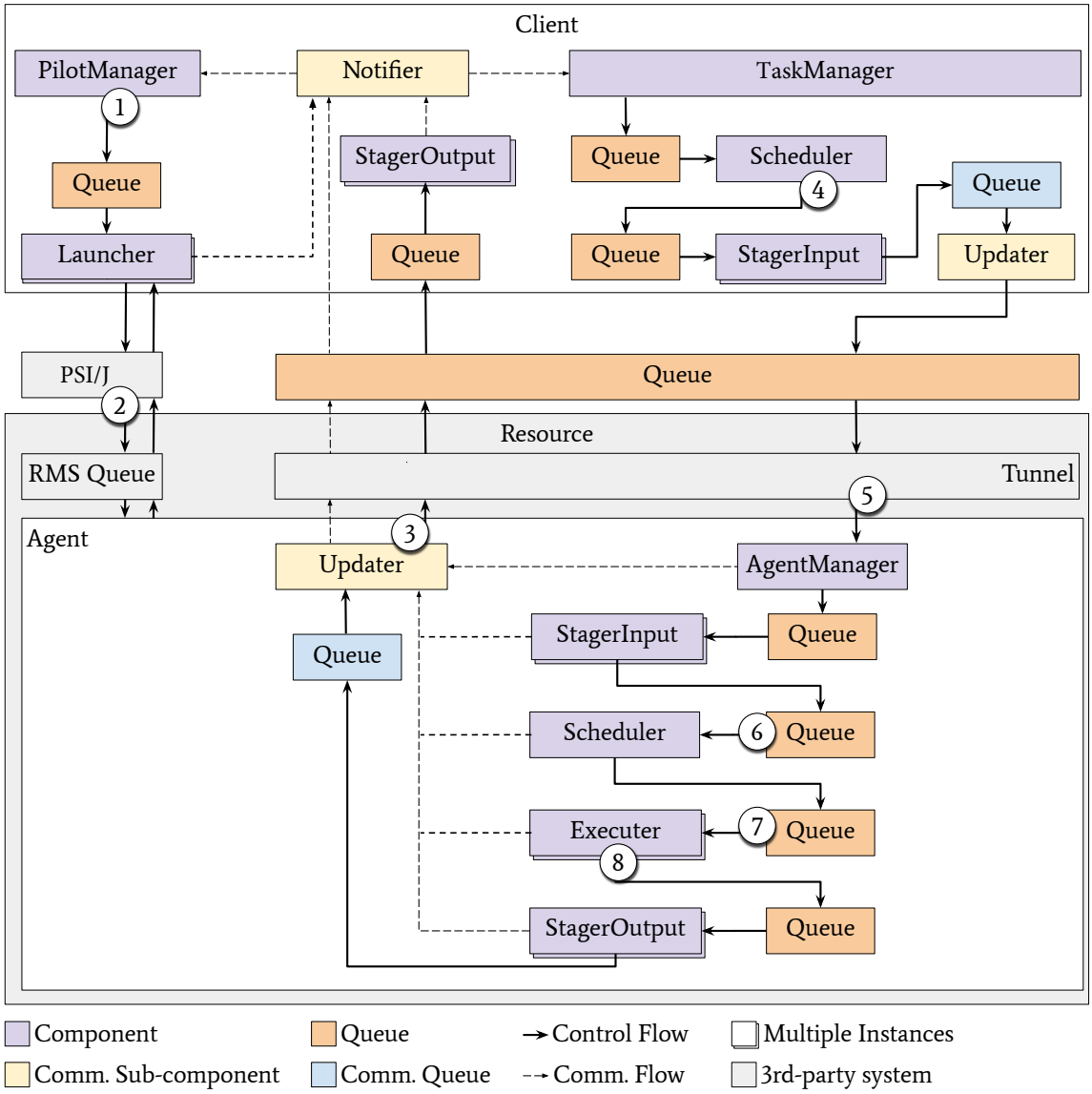

Figure 11. RADICAL-Pilot architecture has two main components: Client and Agent. Depending on the deployment scenarios, the Client may execute on an HPC platform's compute node or remotely, e.g., on the user's workstation. Agent always executes on a compute node. The Client and Agent subcomponents manage resource acquisition and task execution on those resources. Numbers indicate the RADICAL-Pilot's execution process.

to suitable portions of the available pilot's resources and then queues those tasks to an Executor (Fig. 11 ⑦). The Agent's Executor places each task on the assigned resources, sets up their execution environment, and then launches each task for execution (Fig. 11 ⑧).

RADICAL-Pilot is designed and implemented as a building block [138]. In that way, RADICAL-Pilot can be more easily integrated with software tools independently developed by third-party engineering teams. Integration can utilize RADICAL-Pilot's public or private application programming interfaces (APIs). For example, RADICAL-Pilot has been successfully integrated with Parsl [8], Swift [139] and PanDA [94] via its public API, but also with PMIx [140, 137], Flux [107], Hadoop and Spark [81] via its internal API. This paper uses RADICAL-Pilot's private API to integrate it with SOMA.

3.5.3 Workflow Monitoring With SOMA and RADICAL-Pilot. RADICAL-Pilot introduced the concept of services in its latest API implementation and SOMA is treated as a first-class citizen within RADICAL-Pilot. A first-class citizen means that a SOMA task is able to be scheduled and run as any other application task would. This helps support SOMA client binaries that run outside of the application namespaces for collecting different metrics. This section enlists the special considerations required to integrate RADICAL-Pilot and SOMA and the methodologies used to capture various types of monitoring data from across the workflow and application software stacks.

3.5.3.1 Service Support Inside RADICAL-Pilot. Integrating SOMA and RADICAL-Pilot required two main capabilities: (1) scheduling and launching SOMA components on dedicated and shared resources and (2) enabling data exchange between SOMA and RADICAL-Pilot. SOMA can use a set of dedicated resources to run its service instances. The SOMA service is treated as a service task within RADICAL-Pilot. While the service task can specify its resource requirements like any other regular RADICAL-Pilot application task, the SOMA service task needs to be scheduled *before* any application tasks. Recall that this stems from the need for the SOMA service instances to make their remote procedure call addresses publicly known within the workflow for clients to connect. RADICAL-Pilot enables such capability by scheduling the service tasks immediately after bootstrapping its Agent component but before any other task. Service tasks communicate their state to RADICAL-Pilot for the consumers of those services to know where, when, and whether they are available. RADICAL-Pilot's Agent components (see Fig. 11) exchange data via queues

implemented with ZeroMQ [54]. Each component gets its inputs via a queue and pushes its output to another component's queue. That enables RADICAL-Pilot to integrate third-party components via well-defined interfaces and a unified communication and coordination infrastructure.

Service tasks are also special concerning their scope. While regular application tasks execute and go out of scope, thereby releasing their computing resources, service tasks are long-running, i.e., execute for the entire workflow duration. Once the workflow is completed, service tasks are shut down through an appropriate control command from RADICAL-Pilot. Fig. 12 depicts the timeline of events and the RADICAL-Pilot-SOMA interaction model during workflow execution.

Initially, RADICAL-Pilot is scheduled via the HPC platform's batch system as a pilot job [141] (Fig. 12 ①). That allows us to execute RADICAL-Pilot Client that, in turn, executes RADICAL-Pilot Agent on one or more compute nodes (Fig. 12, solid arrows). In this way, we avoid consuming resources on the cluster's login node, in accordance to the HPC platform usage policies. Once the RADICAL-Pilot Agent bootstraps (Fig. 12 ②), it first schedules and launches the SOMA service (Fig. 12 ③), then it schedules the RADICAL-Pilot monitoring task, one for the entire workflow, co-located with the service (Fig. 12 ④). Next, RADICAL-Pilot schedules the hardware monitoring tasks, one on each available compute node (Fig. 12 ⑤). Both monitoring tasks run a SOMA client communicating with the SOMA server via RADICAL-PilotC (Fig. 12, dotted arrows). Finally, once the monitoring infrastructure bootstrap is completed, RADICAL-Pilot proceeds to schedule the task of the workflow application Fig. 12 ⑥. Note that each application task can also run a SOMA client to enable the SOMA service to receive asynchronous application information. SOMA Clients can be launched and stopped via RADICAL-Pilot's task pre/post execution capabilities and/or the task executable can be wrapped in a script that launches the SOMA client.

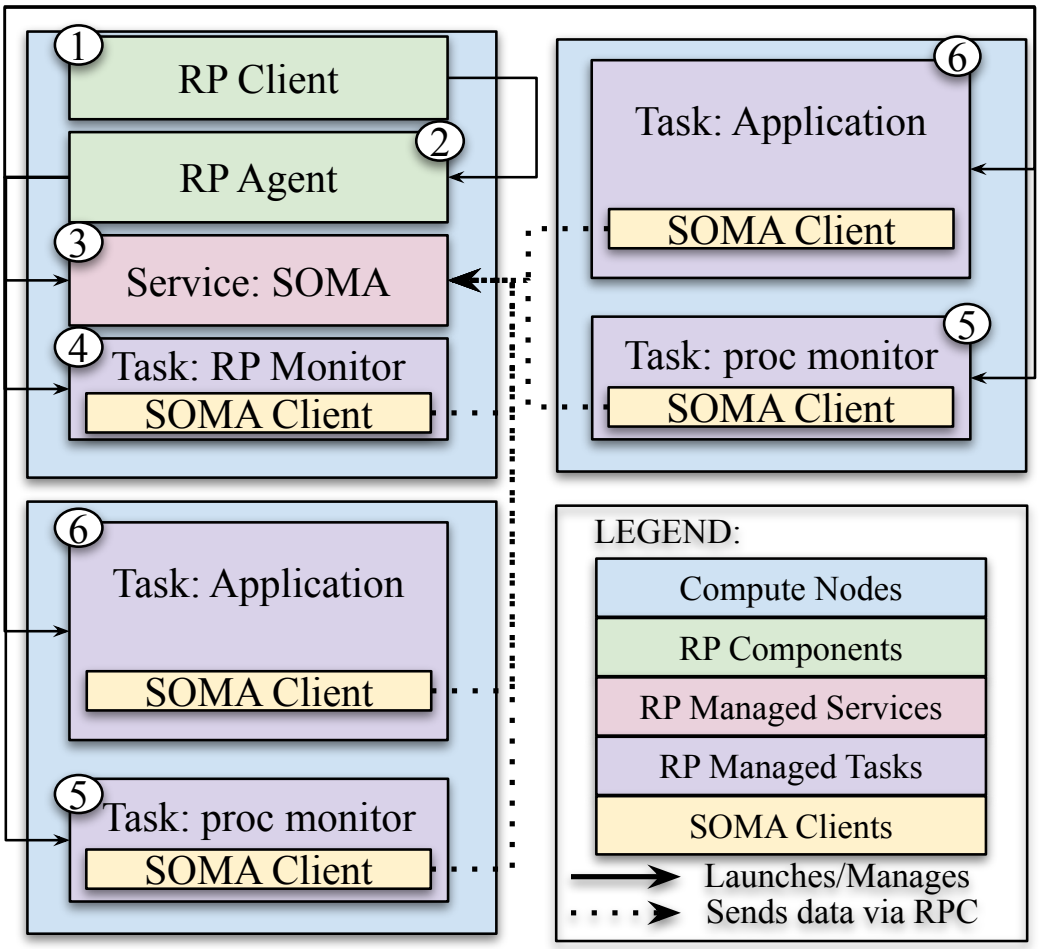

Figure 12. Example layout of how RADICAL-Pilot and SOMA components can be arranged and interact on the compute nodes of a pilot job. RADICAL-Pilot Client and Agent schedule and manage services and tasks on the available resources. A SOMA server is launched as a service, and each monitoring and application task has a SOMA client per rank. Monitoring tasks collect local node hardware and workflow profile data and send them to the SOMA server via a SOMA client.

3.6 Summary

Chapter III presented the technology and architecture used to build SOMA. Utilizing microservices allows for a modular framework which was configured and built to run on many different HPC systems. The Mochi software framework [114] is introduced as well as how SOMA is able to build upon it for better *configurability* across different use cases. Chapter III describes the SOMA API in detail and how our implementation enables fine-grained control over how and when we send data by remote procedure call. We then describe how the application monitoring and workflow monitoring integrations were built and run. An introduction to the RADICAL-Pilot workflow management system is provided as well. Chapter III answered the research question, "How can we run on heterogeneous HPC ecosystems?" by describing our approach and demonstrating the success across numerous heterogeneous systems, and in multiple use cases. Thus bringing us closer to our goal of answering "How can we support online performance observation for HPC applications?"

CHAPTER IV

A CANONICAL DATA MODEL

This chapter contains previously published and unpublished material with co-authorship. Sections 4.3.1 and 4.4 are from a paper initially published at the Cray User Group conference in 2023. An extension of this paper was published in a special edition journal of Concurrency and Computation: Practice and Experience in June 2024. The work was a collaboration between the University of Oregon, NVIDIA Corporation, University of Helsinki, Aalto University, and Academia Sinica. Co-authors include Oskar Lappi, Dr. Srinivasan Ramesh, Dr. Miikka Väisälä, Dr. Kevin Huck, Touko Puro, Dr. Boyana Norris, Dr. Maarit Korpi-Lagg, Dr. Keijo Heljanko, and Dr. Allen D. Malony. I was the first author for both publications, conducted all experiments and completed most of the writing. The Astaroth team (Lappi, Puro, Korpi-Lagg, Heljanko) wrote the majority of the subsection on Astaroth background 4.4.1, as well as the Astaroth scientific analysis in Subsection 4.4.2.

Section 4.3.2 represents unpublished research work from a collaboration between University of Oregon and Lawrence Livermore National Laboratory. I implemented the integration between SOMA and Caliper, conducted all experiments, created all figures, and wrote all sections with guidance from Dr. David Böhme.

Section 4.5 and section 4.6 contain material from an unpublished paper (under review at a 2024 conference). This work was a collaboration between University of Oregon, NVIDIA Corporation, Brookhaven National Laboratory, and Rutgers University. I was the first author for all papers, conducted all experiments and completed the majority of the writing. Co-authors include Dr. Mikhail Titov, Dr. Srinivasan Ramesh, Dr. Ozgur Kilic, Dr. Matteo Turilli, Dr. Shantenu Jha, and Dr. Allen D. Malony. Some co-authors helped with writing, all helped with suggestions, and proof-reading.

4.1 Introduction

In order to make SOMA *flexible* enough to support integrations with the input and output tools described in Chapter II we had to solve the problem of mismatched data representations. Data representation and coupling between scientific code bases is a key challenge to building a vibrant ecosystem of HPC simulation tools. It requires agreeing on or adapting between data representations.

This is also true for HPC application monitoring. It is not enough to set up SOMA services using Mochi on processes and execute RADICAL-Pilot operations, since what data is sent and how it is represented matters. Thus, SOMA utilizes the Conduit [52] data language to build hierarchical data models within each namespace, or for certain data sources. With this canonical data model approach, we demonstrate how we address the requirement of creating a flexible framework that can support the monitoring from different data sources, at different times. We answer the research question "How do we support different input and output?"

The structure of this chapter is listed here for convenience. Section 4.2 provides background on the Conduit technology used for the SOMA data models. Section 4.3 outlines the data models designed for performance measurement libraries TAU and Caliper. Section 4.4 discusses how we chose the data model for Astaroth, as a use case for application diagnostic data. It also offers some results based on collecting data from both the TAU plugin and Astaroth instrumentation and how said data can be interpreted and used. Section 4.5 details the data model for monitoring specific hardware metrics, specifically those gathered from each compute node running the simulation or workflow tasks. Section 4.6 describes the data model chosen for our integration with RADICAL-Pilot for monitoring workflows. We then provide some results from initial experiments enabling online data collection from the OpenFOAM workflow using the RADICAL-Pilot integration, hardware monitoring, and TAU as data sources.

4.2 Conduit

Conduit [52] is an open-source project from Lawrence Livermore National Laboratory (LLNL) designed to simplify data description and sharing across HPC simulation tools. It provides an intuitive API for in-memory data description that enables human-friendly hierarchical data organization. There are commonly shared conventions for exchanging complex data and modular interfaces (in C++, C, Python, and Fortran) for use across software libraries and simulation applications. Conduit provides easy-to-use I/O interfaces for moving and storing data, including support for moving complex data with MPI (serialization). At the heart of Conduit is a hierarchical variant type called a *Node* (henceforth referred to as a Conduit::Node so as to easily differentiate from a compute node). A Conduit::Node can be used to capture and represent arbitrarily nested numeric data. The Conduit data model was chosen for this ability to capture arbitrary hierarchical data. Further, Conduit also provides convenient interfaces to serialize Conduit::Nodes — we rely on

this capability to store and transport monitoring data within the SOMA environment. Conduit is also gaining support in the scientific computing community as a data model to exchange scientific data within components in a workflow. SOMA, like SERVIZ and SYMBIOMON, is assembled out of robust, high-performance Mochi services. The use of a well-supported API and data model in Conduit is in line with our strategy of building a high-performance monitoring service using "off-the-shelf" components. This strategy promotes a high degree of code reuse, resulting in a monitoring service that is (1) easier to maintain and (2) whose functionality is easier to extend compared to *ad-hoc* implementations.

4.3 Performance Profiles

For performance profile data models, there are two different performance libraries SOMA is integrated with. First the TAU data model is described in Subsection 4.3.1, and then Caliper in Subsection 4.3.2. Both have some similarities in structure, namely, keys for identifying where and when the data came from, followed by the performance metrics measured by the library. Once the data is structured appropriately SOMA now has a canonical, understandable data representation available online.

4.3.1 TAU. TAU was introduced in depth in Section 2.3.2.1, and the integration between SOMA and performance tools is detailed in Section 3.4.1. Thus, in this section we focus on describing the data model. We apply the Conduit techniques to create a shared performance data representation that becomes the basis for data sharing across a SOMA environment. Namely, creating Conduit::Node data structures to represent the hierarchical data within a TAU profile. Listing 4.1 is an example of a TAU profile data capture in a Conduit representation for the LULESH application. In this example, the profile represents the data for MPI rank 0, running on node b01n45, which form the second and third level of the Conduit::Node hierarchy, the first being the TAU namespace. The timestamp on the fourth level indicates when the profile data was captured. Lower levels in the hierarchy represent the event classes, while the Conduit::Leaf nodes hold the actual interval timer or counter data.

4.3.2 Caliper. Caliper is a performance measurement and analysis library from Lawrence Livermore National Laboratory [23]. Caliper supports both application instrumentation and sampling for collection of performance metrics. It has numerous "services" which can be enabled to customize which metrics are measured. These options include but are not limited to: hardware

```
{ TAU: /* Top-level Namepace tag */
    b01n45: /* IDENTIFIER: Compute Node */
        Rank 0: /* IDENTIFIER: MPI Rank */
            1708566761.261629: /* IDENTIFIER: timestamp */
                /* TAU Metrics */
                MPI_Routines:
                    MPI_Barrier()_Calls:  fp64
                    MPI_Barrier()_Inclusive:  fp64
                    MPI_Barrier()_Exclusive:  fp64
                    MPI_Irecv()_Calls:  fp64
                    MPI_Irecv()_Inclusive:  fp64
                    MPI_Irecv()_Exclusive:  fp64
                    MPI_Isend()_Calls:  fp64
                    MPI_Isend()_Inclusive:  fp64
                    MPI_Isend()_Exclusive:  fp64
                    MPI_Waitall()_Calls:  fp64
                    MPI_Waitall()_Inclusive:  fp64
                    MPI_Waitall()_Exclusive:  fp64
                    MPI_Allreduce()_Calls:  fp64
                    MPI_Allreduce()_Inclusive:  fp64
                    MPI_Allreduce()_Exclusive:  fp64
                TAU:
                    Tau_plugin_mochi_dump_Calls:  fp64
                    Tau_plugin_mochi_dump_Inclusive:  fp64
                MPI_Counters:
                    Message size all-reduce_Mean:  fp64
                    Message size all-reduce_Min:  fp64
                    Message size all-reduce_Max:  fp64 }
```

Listing 4.1 Conduit data model of the TAU performance data.

performance counters, memory allocations and frees, MPI metrics, and external integrations such as with NVIDIA Nsight. The typical use case is to produce profiles, but traces are also supported.

Caliper's architecture is designed to create "snapshots" of performance data, to be taken at configurable times, i.e., each iteration, or annotated regions of code. These snapshots contain all of the metrics that Caliper has been configured to measure since the last snapshot. Typical offline use for Caliper is that these snapshots are aggregated and written to a text report at the end of the application execution.

We built an integration between SOMA and the Caliper performance analysis library. For an application to use the integration, it would just need to be run with a SOMA service-enabled caliper version profiling it. In other words, the application need not be aware of the connection to SOMA. Another benefit is that any application that is already annotated to use Caliper could make use of this. It works by implementation of a callback for when caliper takes a snapshot of performance data. When a snapshot is taken we make a call to our plugin that restructures the data into our Conduit::Node data model, seen in Listing 4.2, then publishes the Conduit::Node data to the SOMA service. The effect is that all the Caliper profile data that was originally processed *post-hoc* is now available online via SOMA.

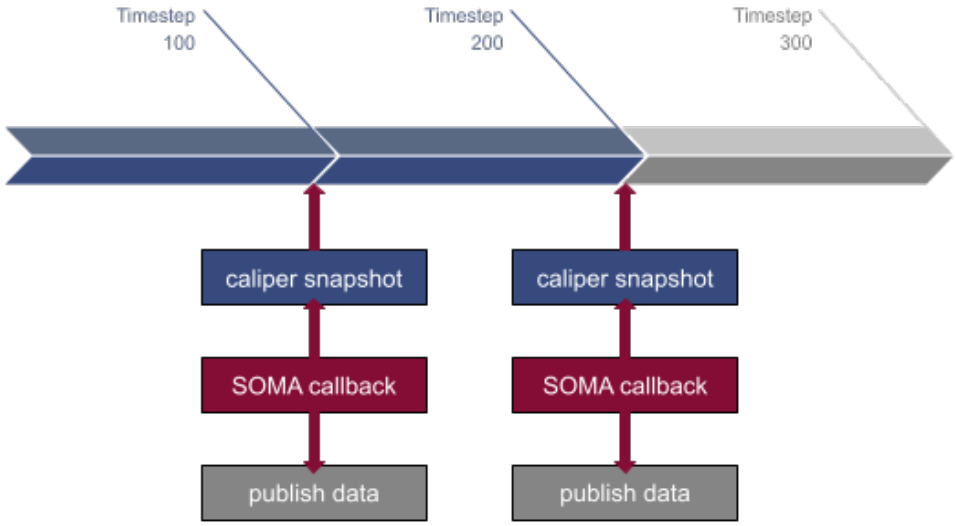

Figure 13. SOMA enables the access of Caliper data online by intercepting Caliper's snapshot function. Every time Caliper samples data, the SOMA client can restructure the data and publish it to the server.

```
{
CALIPER: /* SOMA top-level namespace */
  1708566761.261629: /* IDENTIFIER: timestamp */
    Rank 5: /* IDENTIFIER: MPI rank */
      ApplyMaterialPropertiesForElems: fp64
      CalcEnergyForElems: fp64
      CalcFBHourglassForceForElems: fp64
      CalcForceForNodes: fp64
      lulesh.cycle: fp64
      main: fp64
      loop.iterations: int64
      loop.start_iteration: int64
      memstat.vmsize: fp64
      memstat.vmrss: fp64
      memstat.data: fp64
}
```

Listing 4.2 Conduit data model for Caliper.

4.4 Application Diagnostics

SOMA is by no means limited to capturing performance data from traditional sources such as performance measurement libraries. In addition, it is of interest for the application to report custom information to be monitored. Monitoring of application execution state can be insightful to identify anomalous behavior and other artifacts. In this case, the data is application-specific, and the knowledge of how to best represent it lies with the application developer. Such custom information can include (but is not limited to) the *scientific rate-of-progress* or *figure-of-merit* self-reported by the application. For example, a molecular dynamics code might want to capture the atom-timesteps per second as the figure of merit. At the same time, a DL model is likely interested in monitoring the training loss. The timely availability of such information to the workflow system is the first step towards enabling online adaptivity. In Subsection 4.4.1, we demonstrate an application diagnostic data model for the Astaroth application

4.4.1 Astaroth. Astaroth is a multi-GPU library designed for high-order stencil computations on modern HPC systems [105]. Recently, it has been applied to build a simulation framework for magnetized astrophysical plasmas in the magnetohydrodynamics (MHD) regime, for details of the physics and first production runs, see [142]. The framework solves the standard set of partial differential equations for MHD, namely the continuity, angular momentum, entropy, and induction equations, under conditions that usually occur in astrophysical plasmas. In such plasmas, the densities and temperatures usually range several orders of magnitude, in which case non-conservative formulation of the equations is numerically advantageous over flux conserving schemes. For example, a formulation in terms of logarithmic density, albeit non-conservative, can be numerically more accurate and faster to compute. In this case, however, there is no guarantee that the conserved quantities are accurate to the machine precision, but rather conserved up to the discretisation error of the scheme. Therefore, constant monitoring of the conserved quantities is necessary, and simulations that do not adequately conserve them should be disregarded. Magnetic fields are implemented in terms of the magnetic vector potential to ensure that the field remains divergence-free.

A full-fledged multi-node implementation was taken into production during the LUMI-G pilot phase to study a setup intended for investigating the solar fluctuation dynamo, the physics and highest-resolution CPU simulations so far are described in [144]. Astaroth allows for scenarios of unprecedented resolution, but this comes with the cost of several hundred terabytes per system state. The analysis and movement of such data has become a major bottleneck of performing large-scale computations. This motivates the idea to do data analysis in situ, thus reducing the amount of data that eventually needs to be stored.

Astaroth operates in a *single-program, multiple data* (SPMD) manner, with one MPI rank per GPU device. LUMI-G nodes house four AMD MI250x GPUs each, with each GPU consisting of two Graphics Compute Dies (GCDs). HIP considers each GCD a separate device, and so Astaroth maximally uses eight MPI ranks per node, one per GCD. During the LUMI pilot, Astaroth ran on 1024 nodes and 8192 GCDs. With Astaroth only using eight processes per node, many CPU cores remain idle during the computation. These CPU cores could be used for data analysis *in situ*.

The type of data analysis that is most valuable for large-scale Astaroth runs is determining the numerical health of the simulation. Simulations sometimes fail due to misconfiguration of numerical

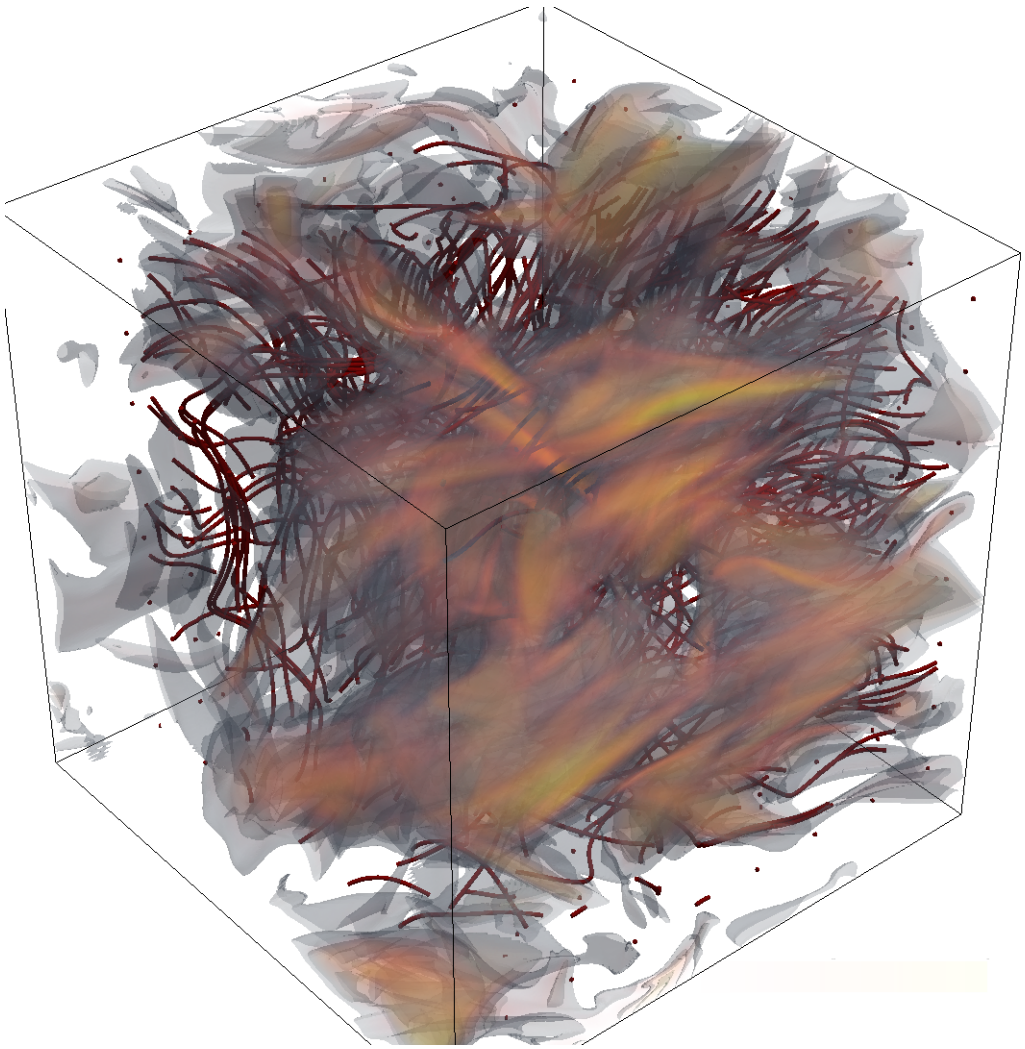

Figure 14. Visualization of magnetic field lines (dark red streamlines) and its intensity (volume-rendered colours) in a dynamo-active Astaroth simulation from [142].

parameters — the simulation becomes unstable or starts behaving nonphysically. For well-tested schemes this is less likely to occur, but during the development of Astaroth, new numerical methods are constantly being added. Without the experimential knowledge of how these new methods interact with Astaroth's existing system components, it is very difficult to create an intuitive understanding of the mechanisms that cause simulation failures. Being able to observe and analyze the numerical state of the simulation improves our ability to build an intuitive understanding of the numerical processes and how they should be tuned to avoid failures. With increasing number of nodes and processing elements, in addition, the probability of bit flips and node failures increases, in which

case monitoring the health of the system becomes of high importance, to avoid unnecessary crashes, corruption of data, and loss of computing time.

The most common symptom of a numerical failure in Astaroth is that the simulation becomes numerically unstable. Due to the non-conservative scheme, such an instability will manifest itself as a loss/gain in the quantity that is required to be conserved. Usually such events occur very localized in the simulation grid, where locally high gradients or extrema occur, and a numerical quantity will begin to grow uncontrollably. There are two possible actions once an instability starts developing: 1) the simulation is scrapped and reconfigured or reseeded to avoid the instability, 2) the simulation is recovered by reconfiguring the simulations parameters on the fly to counteract the instability. In both cases, instabilities should preferably be caught early. As the instability grows, it becomes harder and harder to recover any useful data, and if the simulation has to be scrapped, then one can iterate through simulation parameters more quickly if instabilities are caught early. Recovery is made even more difficult at high resolutions, where it is not feasible to keep more than two snapshots on disk. The snapshot frequency then determines the limit of how far back a simulation can be rewound. The period between snapshots forms a deadline within which Astaroth should detect instabilities.

In order to diagnose numerical instabilities, each Astaroth rank calculates certain reductions on the computational mesh that act as signals of a developing local instability. There are two kinds of data that we are interested in: 1) the numerical loss or gain of conserved quantities, 2) extreme values and their locations. The most important conserved quantity is the mass, on which we concentrate for this proof-of-concept study. For extrema, we record the minimum and maximum values of each field, and their corresponding location. Astaroth considers NaN values to be extremum and detects these as well. It is almost certainly too late to recover once the simulation has produced a NaN value, as they provide a clear indication that the simulation has failed. The mass is obtained as an integral over density, for this Astaroth uses a handcrafted reduction operation. The extrema and NaN locations are obtained using the thrust [19] library, which supports reductions on GPU buffers, both through CUDA and HIP.

Tracking extrema is useful not only for the detection of numerical instabilities, but also for identifying points or regions of interest within the simulation domain. By following the locations of extrema, we could carve out slices or volumes in the simulation domain for visualization and analysis, which would reduce the total amount of data needed to be stored on disk for post-processing.

If this was done in situ, the observability of high-resolution Astaroth simulations would improve considerably.

Astaroth logs these signals, the mass conservation and extrema, but a human needs to interpret them in order to take action. The end goal in monitoring the signals is for Astaroth to be able to configure a policy in Astaroth that, e.g., pauses a simulation when one of these signals indicate a certain likelihood of an instability, or adjusts the simulation parameters on the fly, and continues computations in a numerically safer regime. Methods for calculating the likelihood of instabilities will be easier to develop in a separate component outside of Astaroth. A step towards this goal is to stream the diagnostics from Astaroth through a channel to which analysis components can subscribe. As examples, the diagnostics stream could be fed into a machine learning model doing inference, or a signal processing component that detects spikes in the extrema or fluctuations in the mass conservation.

4.4.2 SOMA and Astaroth Integration. Chapter III detailed the SOMA API through which structured data can be published. In addition, Chapter III provided an example of how Astaroth can use its linked SOMA client library. That implementation is used to structure the diagnostic metrics — demonstrated in this section — and publish them to SOMA. We have set a publishing interval in Astaroth that determines how often Astaroth publishes diagnostics to SOMA. As a proof of concept, we write the data to a file and visualize this data later in this section. It is not difficult to imagine how SOMA could provide on-line analysis of the data through a plugin system, or forward the data to a completely separate analysis engine.

In this study we use a low magnetic Prandtl number setup, meaning that the molecular magnetic resistivity in the induction equation is much higher than the viscosity in the Navier-Stokes equation. This setup is similar to [144], mimicking small-scale dynamo action in the solar and stellar convection zones. We initiate a non-zero but very small in magnitude seed magnetic field in the domain, and the crucial physics question is whether the flow field can act as a dynamo and exponentially amplify its seed? In such a setup, the plasma flow is highly turbulent, and Reynolds numbers, measuring the vigor of fluid turbulence, are high, while the magnetic Reynolds number can only be kept slightly above the threshold for dynamo action due to numerical reasons. In this kind of a system, the dynamo will grow, but very slowly, and the stability properties of the system are mainly determined by the hydrodynamical part. Hence, in this paper, we concentrate on

```
{
            pid: uint32, /* IDENTIFIER: MPI rank */
       timestep: uint64, /* IDENTIFIER: Iteration */
simulation_time: fp64, /* IDENTIFIER: timestamp */
    local_mass: fp64,
        FIELD_1: {
         min: { value: fp32, location: [uint16] x 3 },
         max: { value: fp32, location: [uint16] x 3 },
         nan: { value: boolean, location: [uint16] x 3 }
        },
        FIELD_2: { min: ..., max: ..., nan: ... },
        ...
}
```

Listing 4.3 Conduit pseudoschema of the Astaroth diagnostics data model. The pid and timestep uniquely identify a node in a data stream. The simulation time and local mass provide important values for calculating instabilities.

monitoring the conservation of mass and extrema of the flow field, see the example Conduit::Node data model in Listing 4.3. In the study of [142], systems with magnetic Prandtl numbers of unity were investigated, see Fig. 14. In such a setup, the magnetic field can grow to a significant strength, and also participate in the dynamical evolution. In this case, monitoring also the magnetic field diagnostics becomes important. This is not the case in the setups presented in this paper, however.

SOMA collects the data of interest from Astaroth live during a simulation. At the moment, for this proof-of-concept, we simply write the application data stream to a file at the end of the simulation, which we postprocess using Astaroth's analysis tools. While the analysis in the proof-of-concept is not done live, there is no fundamental reason why these analysis tools cannot be attached to the data stream while the simulation is running. Live analysis would allow us to pause or stop the simulation or in the most ideal case to adjust the simulation parameters and carry on the integration with numerically safe parameters, if signs of numerical instability would be detected.

As discussed, an Astaroth simulation can fail due to numerical instabilities, which result from the simulation encountering an extreme state, for which the chosen simulation parameters do no longer guarantee numerical stability. Typical cases are the viscosities being too small or the time integration step being too high to resolve the plasma flow and its evolution. These are the two example cases investigated here. As the most typical sign of an approaching numerical instability is the loss/gain of mass, which should be a conserved quantity in a healthy simulation, we will use this quantity for our proof-of-concept monitoring cases.

In the case of a healthy simulation, the mass as a whole and the isotropic turbulent fluid in each rank remains constant. Mass diagnostics collected from different ranks from such a simulation is shown in the leftmost panel of Fig. 15, and the evolution of density over time in each rank in the leftmost panel of Fig. 16, both showing healthy statistics with constant mass and nearly constant density extrema amongst all ranks.

Next we demonstrate a case, where the integration over time, which is in this case performed with 3rd order Runge-Kutta scheme, is done too inaccurately. The simulation does use the Courant-Friedrichs-Levy (CFL) condition for calculating the maximum allowed time step length during each iteration, but in addition different physical setups require a safety prefactor around 0.1 ... 0.9 for stability. The required value of the prefactor is not known a priori for different types of physical setups, and hence it often happens that the user gives too high a value for it in the simulation parameter setup. Mass diagnostics collected from such a simulation are shown in Fig. 15 middle panel. We see that in such a case the mass is systematically lost. The mass loss is global, but happens isotropically in each rank, giving a clear imprint of the simulation parameters being globally wrong instead of indicating numerical instability developing due to an extreme condition localized to some rank/simulated region. Also the middle panel of Fig. 16 consistently agrees with this picture, the density extrema showing linearly decreasing trend with the same slope.

Next we will study a case, where the viscosity is set to a value that turns out to be too small to guarantee numerical stability. In this case, some parts of the flow are well resolved, while more extreme conditions happen only as local fluctuations in the turbulent fluid. These extreme conditions cause a numerical instability on one individual rank, in this case rank 14, the instability grows rapidly, and causes the entire simulation to crash. In the mass distribution plot Fig. 15 rightmost panel the local rapid increase of mass in rank 14 is clearly visible.

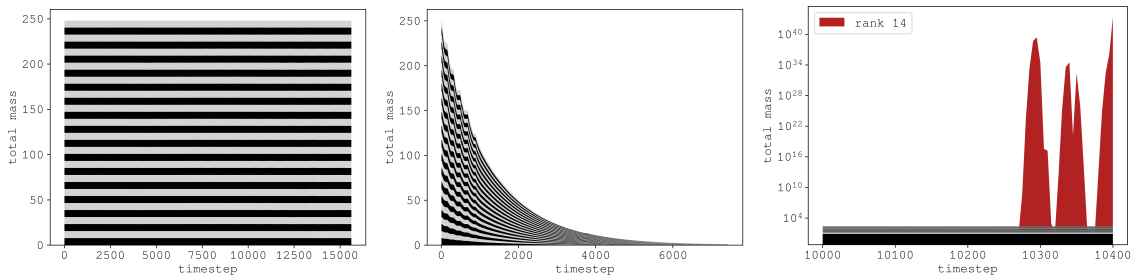

Figure 15. Time series of the mass distribution from different types of simulation setups: healthy (left), too large time step prefactor (middle) and too small viscosity (right). Each shaded region in the stack plot is the local mass in a rank over the runtime of the simulation. In a healthy simulation the total mass in the system remains constant to the accuracy of the spatial discretization scheme. Note: in the rightmost series, for a better visualization of the instability, the y-axis is on a log scale and the x-axis starts from 10000 .

Even more information can be retrieved by plotting the density evolution in each rank, Fig. 16 rightmost panel. From there one can see that the explosion of mass is clearly preceeded by abnormal behavior of the minimum density on Rank 14. It starts diminishing abnormally fast in comparison to the other ranks already 1500 timesteps before the actual crash happens. This gives us a clear hint of the nature of the numerical instability. In the location of the extreme conditions in the turbulent flow some mass is actually lost, pressure in the location is decreased, and that causes a rapid inflow of plasma towards that location. This results in the break-down of the numerical scheme and the seen mass explosion.

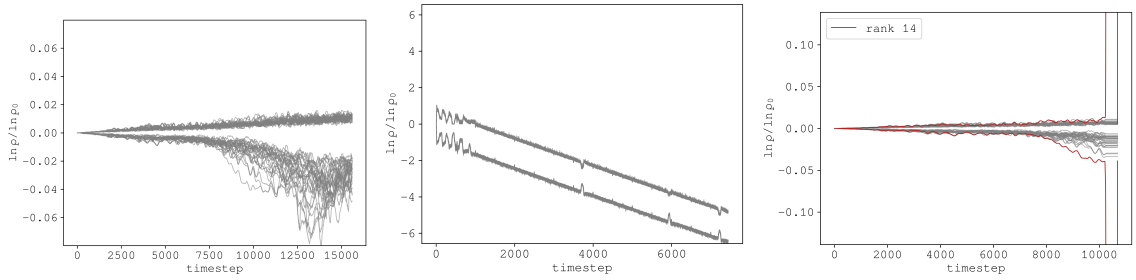

Figure 16. Time series of the density range from different types of simulation setups: healthy (left), too large a time step prefactor (middle) and too small viscosity (right). Each gray curve is a minimum or maximum of one rank's density field over the runtime of the simulation.

4.5 Hardware Metrics

For measuring hardware metrics on each compute node, we created a data model for certain CPU utilization metrics. Basic information about the state of the hardware, gathered periodically by reading `/proc/` is captured by standalone SOMA client tasks, which can be scheduled on reserved

```
{
 PROC: /* Top-level SOMA namespace */
    cn4302: /* Hostname tag */
        3824813742052238: /* Timestamp */
            Uptime: 49902
            Num Processes: 3
            Available RAM: 8422
            stat:
                cpu: 10749 865 685 9293 999 745
                cpu0: 4698 591 262 8953 612 449
                ...
}
```

Listing 4.4 Conduit::Node of the hardware data model

cores on each compute node within the workflow. RADICAL-Pilot launches these client tasks before application tasks are scheduled and they run for the duration of the workflow. Granted, such information can be captured directly by the application task in an alternative design. However, there is the issue of task scope. Application tasks come and go, but the hardware information is valid throughout the workflow execution. Therefore, we captured this information through long-running special SOMA client tasks. See Listing 4.4 for an example of the Conduit data model implemented to represent some `/proc/` data.

4.6 Heterogeneous Workflow States

HPC clusters have continued to grow in size, scale, and complexity, offering degrees of scale-out parallelism that few scientific applications can take full advantage of using traditional scaling strategies. In the past decade or so, the type and mix of scientific application workloads requiring HPC resources have undergone a paradigm shift, moving away from the traditional single program multiple data (SPMD) model to include a set of heterogeneous tasks working towards a common scientific goal [20]. The recent advances in machine learning (ML) and artificial intelligence (AI) technologies have served to motivate their integration into traditional scientific

applications, accelerating this move away from traditional message-passing-based SPMD execution models. Heterogeneous workflows, touted to be the new HPC "application" [20], have emerged as a promising approach to deploying scientific applications on HPC platforms. Heterogeneous workflows is an umbrella term, covering the broad spectrum of multi-task execution, from ensemble computing, wherein several instances of the same (parallel) task are launched on a set of HPC resources to workflows where several instances of different types of tasks asynchronously execute with varying degrees of concurrency.

The breadth of HPC workflows spans a variety of scientific domains, including, for example, molecular dynamics, ML applications, material science, climate scienceand drug discovery. Today, workflows executing on HPC platforms orchestrate 100s to 10000s of individual tasks [62]. The tasks can be heterogeneous both in terms of the application they execute and the computing resources to which they are mapped. As HPC workflow scale continues to grow, managing workflow resources efficiently while simultaneously optimizing for the scientific output or performance of the workflow assumes vital importance. We argue that the ability of the workflow system to dynamically adapt task execution based on the information about the tasks that have been completed or the state of the hardware resources would represent an important step towards enabling optimal workflow resource management.

Consider, for example, the simplest case of a workload consisting of a set of MPI tasks scheduled to execute on a set of HPC resources. Most workflow systems place the onus of deciding on the resource requirements of the individual tasks on the user. The user supplies this information *a priori* to the workflow system, typically during the task creation step. However, if the task does not scale well, assigning too many or too little resources to each individual task may lead to poor use of the assigned HPC workflow resources. The optimal strategy might well be to run more or less tasks, each at a smaller or larger scale. In another scenario, involving the concurrent coupling of machine learning (ML) tasks to traditional ensemble simulations[79], the allocation of computing resources to the two task types is not always apparent, and a misconfigured allocation can accrue vast performance penalties. In both these scenarios, enabling adaptive decision-making within the workflow system would require the *timely* availability of all the *necessary* performance information.

Several software challenges need to be addressed before the promise of adaptive workflows is realized in full effect. This research work focuses on the challenges involved in enabling robust

performance observability of the workflow execution. First, there is the challenge of understanding what data to collect and make available online to the workflow such that observability is enabled. The second challenge pertains to choosing the appropriate data model for monitoring and analysis. Third, different types of data may require different instrumentation strategies and sampling frequencies that must serve the goal of observability while not incurring significant overheads during their measurement and transport. Fourth, there is the related challenge of exporting, storing, and making available the monitoring data online in a timely fashion to enable adaptive decision-making. Lastly, there is the question of how to assign resources to the monitoring system, given that it must run as a part of the workflow. This includes choosing the suitable interaction model between the workflow and the monitoring system.

This section focuses on the use of SOMA, (Service-based Observability, Monitoring, and Analysis) [157, 158] in a new approach — for HPC workflows. SOMA's ability to monitor and enable adaptive workflow execution is demonstrated through integration with RADICAL-Pilot [98], an HPC pilot-enabled [141] runtime system, integrated with multiple workflow systems including RADICAL-EnTK [15] and Parsl [8, 11], and that can be deployed on exascale platforms [12]. Current projections suggest that scientific workflows for HPC can comprise between 10^6–10^9 tasks [62] At this scale, performance monitoring is (1) likely to be treated as a first-class citizen of the workflow and (2) likely to require a non-negligible amount of computing resources [62].

In summary, the key contributions of this work are:

1. The design and implementation of SOMA for service-based workflow monitoring with RADICAL-Pilot as an exemplar use-case

2. Experiments demonstrating that SOMA can enable holistic observability of workflow performance

3. Experiments demonstrating the costs and benefits of enabling workflow observability through a service-based monitoring architecture.

This section discusses the data model necessary to monitor heterogeneous HPC workflows manged by the RADICAL-Pilot pilot system. Background on the two main tools, RADICAL-Pilot and SOMA, then we discuss the work required to integrate them. Section 3.5.1 discussed SOMA and the associated service components for workflow monitoring. Section 3.5.2 described the architecture

of RADICAL-Pilot and the ensemble environment. Section 3.5.3 presented the novel work on how SOMA is integrated into RADICAL-Pilot and the special considerations needed to support HPC workflows (as opposed to traditional MPI-based HPC applications). In Section 4.6.1 the workflow state data model is shown. Then in Sections 4.6.2, 4.6.3, and 4.6.4 the data model is successfully used in numerous workflow monitoring experiments.

4.6.1 Capturing Workflow State Data. SOMA's ability to split its service task resources between several instances and the concept of logical namespaces finds application in workflow monitoring. Monitoring data can now be divided into four namespaces — workflow, application, hardware, and performance. The total of N SOMA service processes within the service task is divided appropriately among how ever many instances are implemented for a given setup, each supporting the compute and storage needs of one namespace. For this work specifically, we required the hardware monitoring data model discussed in 4.5, but also a workflow data model. In addition, the performance namespace for TAU was improved on from previous work, and the application namespace is not used. The methodology for capturing data for the workflow namespace is as follows.

For the first of the two new namespace implementations we created a model for RADICAL-Pilot's workflow states. RADICAL-Pilot's components function as a state machine — the lifecycle of each component, including application tasks, proceeds through a set of predictable states through the execution. For example, a task proceeds through the NEW, SCHEDULED, EXECUTING, and DONE/FAILED states. The transitions between states are further broken down by timestamped events that indicate a new state, shown in Listing 4.5 are the events within the EXECUTING state. Likewise, the Pilot component and the RADICAL-Pilot Agents have appropriate state transitions. Snap-shotting these state transitions and collecting statistics on the total number of pending tasks, completed tasks, and so on can provide valuable insight into the overall performance of the workflow, to be used for subsequent analysis (online or offline). SOMA captures workflow-level information through a service client task launched on a single compute node within the workflow. This client task launched as a daemon, periodically reads the appropriate profile files generated by RADICAL-Pilot, summarizes basic statistics about the workflow from this data, and publishes the same to the SOMA service processes (using RPC) at a configurable monitoring frequency. This client task runs for the

```
{
 RADICAL-Pilot: /* Top-level SOMA namespace */
    task.000000: /* Task name tag */
        /* Timestamp:       "event" */
        1698435412.6060030: "launch_start"
        1698435412.6135450: "launch_pre"
        1698435412.6177090: "launch_submit"
        1698435412.9642950: "exec_start"
        1698435412.9681430: "exec_pre"
        1698435412.9717330: "rank_start"
        1698435427.9775150: "rank_stop"
        1698435427.9812500: "exec_post"
        1698435427.9850750: "exec_stop"
        1698435428.0497950: "launch_collect"
        1698435428.0540810: "launch_post"
        1698435428.0583980: "launch_stop"
    task.000001:
    ...
}
```

Listing 4.5 Conduit::Node of the RADICAL-Pilot workflow data model

duration of the workflow. See Listing 4.5 for an example of the Conduit data model implemented to represent this workflow data.

4.6.2 OpenFOAM Workflow. To demonstrate this case we run the existing real-scale workflow from the Exascale Additive Manufacturing (ExaAM) project [29], part of the Exascale Computing Project. The ExaAM project has developed a suite of exascale-ready computational tools to model the process-to-structure-to-properties (PSP) relationship for additive manufactured (AM) metal components. The target workflow contains simulations for the melt pool physics and uses AdditiveFOAM [31], an extension of OpenFOAM [136] for AM processes.

We created a corresponding workflow using RADICAL-Pilot [12] and will refer to it as the OpenFOAM workflow. We run this workflow on the Summit supercomputer, monitoring it with our SOMA service. Each compute node of Summit has 44 physical cores, two of which are reserved for the system, leaving 42 available to the user. We run four different task configurations within the OpenFOAM workflow, with either 20, 41, 82 or 164 MPI ranks per task, and with each rank using one physical core. We turn hardware multi-threading off, and reserve one core per node for the SOMA hardware monitoring client. Thus, each task utilizes between 0.5 compute node to 4 compute nodes, respectively. Initially, we run one instance of each task configuration across a workflow with 4 nodes, we refer to this as the "tuning" run, and then we run 20 instances of each across 10 nodes, which we refer to as the "overloaded" run, see Table 5. In both cases we allocate one extra node (for 5, and 11 total) that is reserved for running the RADICAL-Pilot Agent and SOMA service.

Experiment	Tuning	Overload
Number of Tasks	4	80
Number of Nodes	10	10
Number of MPI Ranks	20, 41, 82, 164	
Monitors	proc, rp, tau	
SOMA Ranks Per Namespace	1	
Configuration	exclusive	
System	SUMMIT	
Hardware	IBM POWER9	

Table 5. OpenFOAM Experiment Summary

4.6.2.1 Monitoring Setup. In these experiments we use the following three SOMA clients to observe the workflow and collect metrics. The first is workflow task information via our RADICAL-Pilot monitoring client for SOMA, launched with one client per workflow. At configurable intervals, the SOMA client gathers RADICAL-Pilot-managed task status information, calculates the time spent in each state, and sends it via remote procedure call to the SOMA service. This client setup is shown in Figure 12 by the pink square ③. We schedule the RADICAL-Pilot monitoring client onto the same node as the RADICAL-Pilot client and agent, sharing the node resources. This is an additional node that is not used for any simulation-related tasks, but only by RADICAL-Pilot and its associated SOMA monitoring client. The second source of metrics is from the hardware monitoring client shown in Figure 12 by the squares ④. At configurable intervals, that client reads memory and CPU usage data from `/proc/`. As mentioned, this monitoring client runs on one physical core on each node of the Summit supercomputer.

The third and final data source for these OpenFOAM experiments is the TAU [84] performance profiling plugin for SOMA, which samples the running application to gather performance data, e.g., time spent in each function or MPI metrics. In this case, the TAU plugin is executed without any need for instrumentation of the OpenFOAM code. The plugin uses the `tau_exec` functionality to sample the code and publishes the sampled performance profiles to the SOMA server. This isn't specifically denoted in Figure 12, but is similar to the squares ⑤, where a SOMA client is created within the same task space as the application. However, from OpenFOAM's perspective, it is just being profiled by TAU, and all SOMA functionality is encapsulated by the TAU plugin.

The three data sources: RADICAL-Pilot task information, hardware (proc), and TAU performance profiles, give us a rich understanding of the online performance of the OpenFOAM workflow. In sections 4.6.3, and 4.6.4 we discuss the OpenFOAM results at two levels of granularity, local to each task and global to the entire workflow.

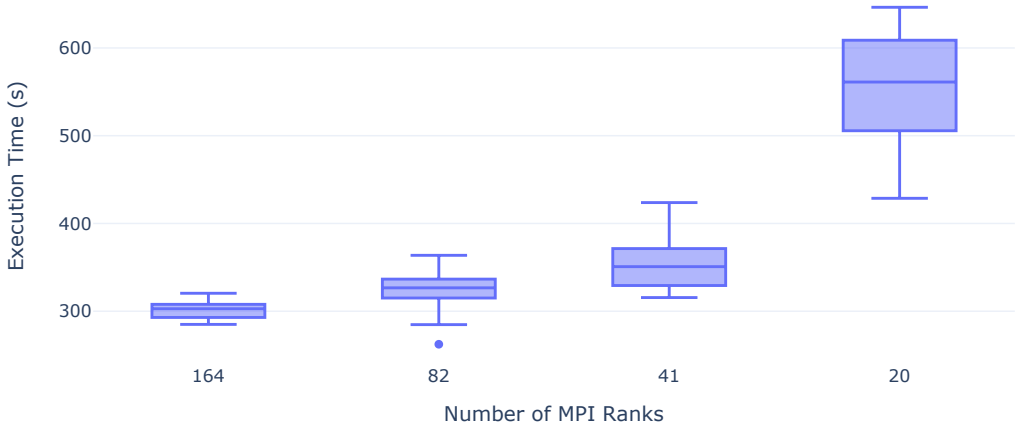

Figure 17. A scaling study of different OpenFOAM configurations. We vary the number of MPI ranks and run 20 instances of each configuration in one RADICAL-Pilot managed workflow.

4.6.3 OpenFOAM Task Scaling Analysis. Figure 17 shows the results of strong-scaling the OpenFOAM tasks from one to four nodes. Without any *a priori* knowledge of the application execution time we ran multiple configurations. Since we capture performance from 20 different instances of each configuration, we see a variation in the total execution time, but are able to calculate averages. An interesting observation that can be made here is that there is limited benefit to scaling

the OpenFOAM tasks beyond two nodes. This information can be used to inform RADICAL-Pilot which MPI task configurations to use, and is fundamental in enabling runtime adaptivity. RADICAL-Pilot could collect information about MPI task performance, and utilize that information to change the task description, adjusting the number of ranks of each type of task in the workflow. As shown by our experiments, that would allow to utilize the available resources better, thus reducing the total time to completion of the entire workflow. Additionally, from monitoring with the TAU SOMA integration we can also gather the related MPI metrics that correspond for further analysis. We zoom in on one instance of a task for clarity in Figure 18. We observe that a large portion of time for each rank is spent in `MPI_Recv()` and `MPI_Waitall()`. While this may not be as easy for on-the-fly adaption, it can be used for further simulation tuning.

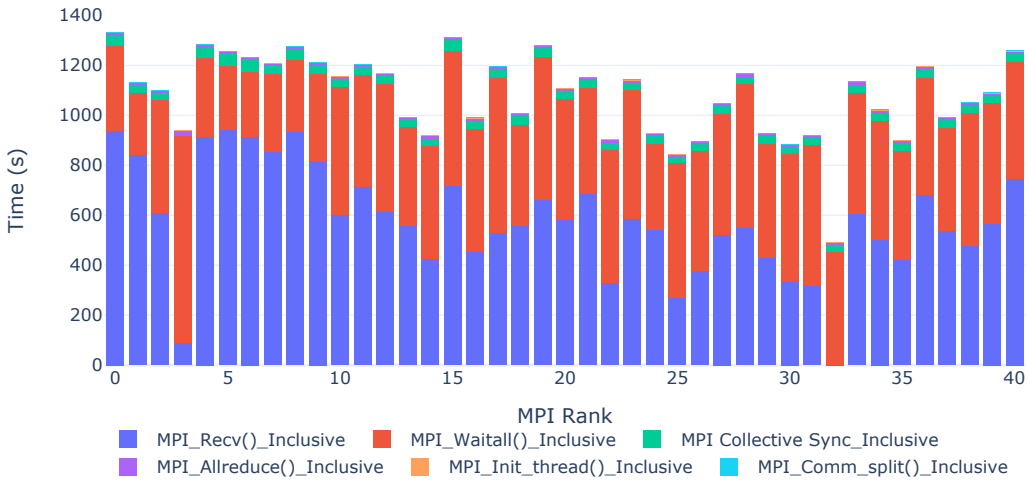

Figure 18. From the TAU SOMA plugin we have access to TAU profile information such as the time spent in MPI calls here to observe load balancing.

As with the number of ranks per task, RADICAL-Pilot could utilize the information about the physical location of the MPI ranks across nodes to make adaptive scheduling decisions. Figure 19 shows our results from comparing the execution times when the physical location of the MPI ranks can differ. In the case of 20, and 41 MPI ranks — where all ranks could fit on one compute node — RADICAL-Pilot has the option to schedule all of these on one compute node, or spread the ranks out across any available CPU cores of the allotted nodes. Figure 19 shows a distribution of 20 and 41 rank tasks running on different node distributions based on what was available during

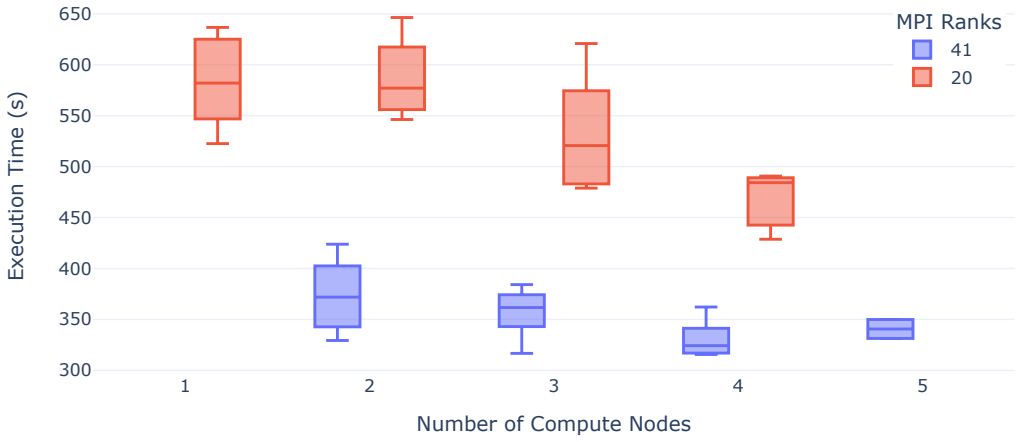

Figure 19. Comparison of the execution time of OpenFOAM tasks when configured with 20 or 41 ranks and scheduled onto a different number of compute nodes.

the overloaded run. Along the x-axis we see whether each task was run on a single node, or split across up to 5 nodes. For the 20 rank runs we actually see an execution time improvement as the ranks are spread across more nodes. This is possibly due to the fact that the smaller rank runs were typically scheduled later in the workflow, when resources are less utilized. This can be seen in Figure 20 where non-green sections indicate unused resources. However, in the 41 rank case our performance improvement is not quite as remarkable as we may be approaching an upper bound.

4.6.4 OpenFOAM Workflow Analysis. Observation of the workflow from a global perspective offers insights into areas where scheduling of future tasks could be improved. Results from our overloaded scenario — with twenty instances of each task configuration — are shown in Figure 20 and in Figure 21. In the CPU utilization graph (Figure 21), we see the cpu percent used on each node represented by a colored line, this was sampled by the SOMA hardware monitoring client from `/proc/` every 30 seconds. Each *rank_ start* event is denoted by a purple dot at the bottom of the figure, at the time when the SOMA RADICAL-Pilot monitor records these events. In Figure 20 we can see the resource utilization from the perspective of RADICAL-Pilot. It marks resources (cores in this case) as either available or unavailable, but does not look at percent utilized. The green color on the graph indicates a resource has a task running on it, and the purple sections show when it is scheduling a new task to be run on that resource. The resources are well used, but using this data

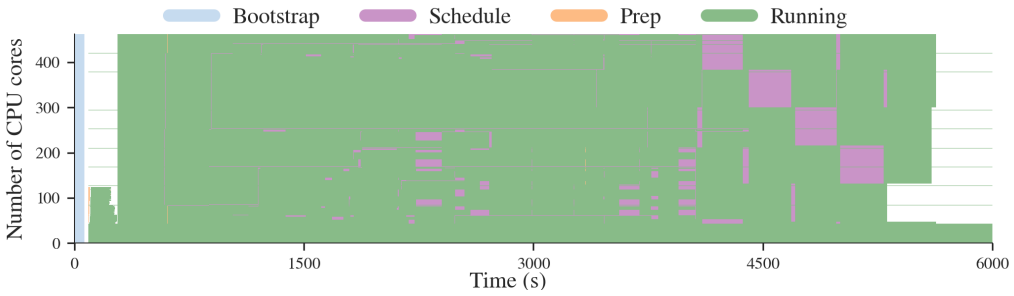

Figure 20. RADICAL-Pilot resource utilization for the OpenFOAM Overload workflow. The figure shows the overload workflow with 4 types of tasks, and 20 task instances for each type. The 4 types of tasks have instances of tasks with different configurations: tasks with 20, 41, 82 and 164 MPI ranks (80 tasks in total). Light blue indicates RADICAL-Pilot bootstrap time. Purple indicates RADICAL-Pilot task scheduling time. Green indicates task running time. The time spent by RADICAL-Pilot preparing the tasks (Prep) is negligible. White space indicates unused resources, a measure of RADICAL-Pilot scheduling optimization based on information provided by SOMA.

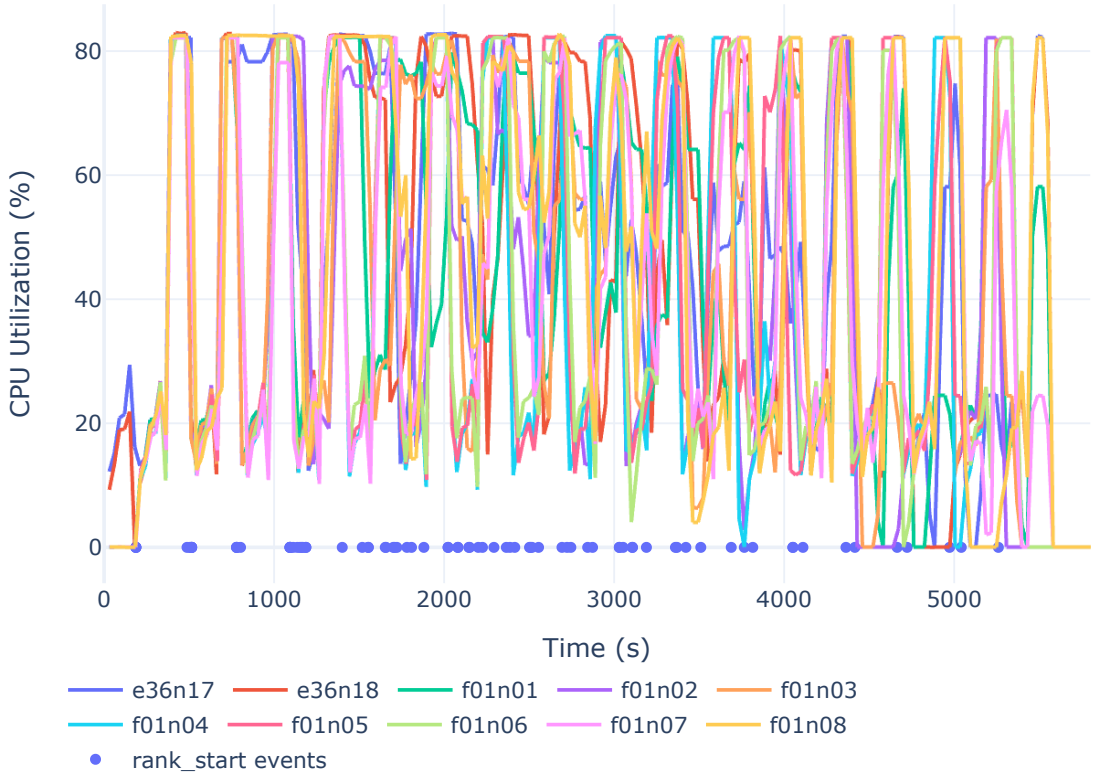

Figure 21. CPU utilization percentage during our “overloaded" run with 20 instances of each task configuration (of varying rank counts). The colored lines represent the CPU utilization on each compute node, sampled every 30 seconds by the SOMA hardware monitor. The purple dots at the bottom indicate when a rank has started, as observed by the SOMA RADICAL-Pilot monitor.

in combination with the CPU utilization could lead to even more informed decisions. Currently, RADICAL-Pilot schedules a task as soon as there are enough free resources, i.e., CPU cores and/or GPUs. Based on the online information about overall CPU (or GPU) utilization, RADICAL-Pilot could adapt its scheduling decisions, prioritizing the use of the free CPUs on a node with comparably lower overall CPU utilization. The benefits of this approach are better appreciated with a run with a lower overall resource utilization.

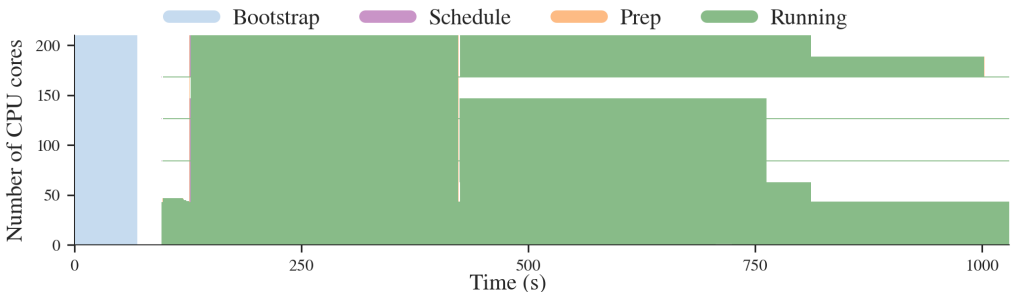

Figure 22. RADICAL-Pilot resource utilization for the OpenFOAM Tuning workflows. The figure shows the Tuning workflow with the same 4 types of task as for the top figure, but with 1 task instance for each type, for a total of 4 tasks. Light blue indicates RADICAL-Pilot bootstrap time. Purple indicates RADICAL-Pilot task scheduling time. Green indicates task running time. The time spent by RADICAL-Pilot preparing the tasks (Prep) is negligible. White space indicates unused resources, a measure of RADICAL-Pilot scheduling optimization based on information provided by SOMA.

Figures 22 and 23 provide a simplified version of the results above where we launched only one instance of each task type. With fewer nodes and tasks it is easier for a human to observe the specific behavior. We can see on the CPU utilization graph (Figure 23 that as a rank starts, there is a corresponding spike in CPU utilization. However, we can clearly see an imbalance in the utilization on each node in roughly the latter half of the runtime, which offers room for better scheduling decisions on a second go-around. As in Figure 20, in Figure 22 green represents resource utilization, and white space denotes free cores, or resources that could potentially be utilized. In this case, once the 164 rank task completed using all of the cores, the other tasks were scheduled to use the cores simultaneously.

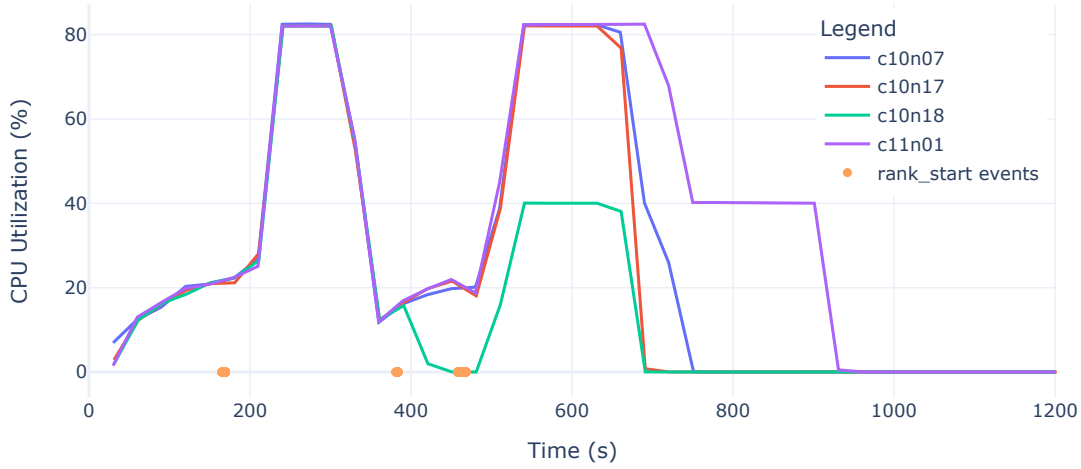

Figure 23. CPU utilization for an OpenFOAM Tuning workflow with 4 types of tasks, and 1 task instance for each type. The 4 types of tasks have 20, 41, 82 and 164 MPI ranks respectively, for a total of 4 tasks. Each colored line shows the CPU utilization on a different compute node, measured every 30 seconds by the SOMA hardware monitoring client. The orange dots indicate when the SOMA RADICAL-Pilot monitor observed from RADICAL-Pilot that a task is starting.

4.7 Summary

The key contributions in Chapter IV revolve around the successful implementations of Conduit [52] data models for representing performance data. The four paradigms that are supported so far are:

1. Performance profiles from integrations with existing performance measurement libraries. Both TAU and Caliper data models were demonstrated.

2. Application diagnostic data by application instrumentation. The usefulness of this data model was demonstrated with the Astaroth work.

3. Event-based workflow state data which was collected from RADCIAL-Pilot which enables task-level updates.

4. Hardware metrics from `/proc/` for monitoring compute node behavior.

Being able to represent all these different data sources with a canonical model makes SOMA more *flexible*. We demonstrated the implementation of combinations of different data collection sources with both the Astaroth application and the OpenFOAM workflow managed by RADICAL-Pilot. The research question "How do we support different input and output?" is thus satisfied by the successful approach and deployments of SOMA's canonical data model.

CHAPTER V
AN EXASCALE PERFORMER

This chapter contains previously published and unpublished material with co-authorship. Section 5.2 contains work from a paper initially published at the Cray User Group conference in 2023. An extension of this paper was published in a special edition journal of Concurrency and Computation: Practice and Experience in June 2024. The work was a collaboration between the University of Oregon, NVIDIA Corporation, University of Helsinki, Aalto University, and Academia Sinica. I was the first author, conducted all experiments, and completed the majority of the writing. Some co-authors helped with writing, all helped with suggestions, and proof-reading.

Section 5.3 contains material from an unpublished paper (under review at a 2024 conference). This work was a collaboration between University of Oregon, NVIDIA Corporation, Brookhaven National Laboratory, and Rutgers University. I was the first author, conducted all experiments and completed the majority of the writing. Co-authors include Dr. Mikhail Titov, Dr. Srinivasan Ramesh, Dr. Ozgur Kilic, Dr. Matteo Turilli, Dr. Shantenu Jha, and Dr. Allen D. Malony. Co-authors helped with suggestions, and proof-reading.

5.1 Introduction

Performing at scale with applications and workflows running in HPC environments is a crucial requirement for a performance monitoring framework such as SOMA. While some of the architectural decisions explained in Chapter II contribute to creating a more performant framework we also have to look for novel and creative ways to make improvements. The differences in application and workflow characteristics, while posing a challenge due to the heterogeneity, can also present opportunities for out-of-the-box solutions. We explore some of these ideas, such as sharing compute nodes in this chapter as well. We discuss the results of our overhead studies and how well they address the challenge of making SOMA a *performant* framework. This chapter answers the research question "How do we minimize the overhead?"

Section 5.2 provides a study of different configurations of SOMA and how it contributed to monitoring overhead differently in the cases of monitoring LULESH and Astaroth applications. Along with physical placement and ratio of servers, we also compare publishing rates and SOMA's blocking versus non-blocking API calls. Section 5.3 provides additional overhead analysis but with

the DeepDriveMD Mini-app heterogeneous workflow managed by RADICAL-Pilot. In this section we look into the perks of reserving or sharing extra nodes allocated for SOMA servers, we also discuss how the online analysis we can conduct during the workflow could lead to more online adaptations.

5.2 Monitoring Overhead

Our objective in this section was to measure the performance of the integration of the SOMA framework with Astaroth and TAU to enables online performance monitoring, application-specific diagnostics, and simulation state data analytics. SOMA's microservice architectures allows us significant flexibility concerning how and where to run SOMA in relation to the application. The most straightforward configuration for monitoring *in general* would be to launch a job with extra compute nodes and run the SOMA monitoring service there. However, in the case of Astaroth, since there were idle CPU cores we also implemented the scenario where the SOMA monitoring service ran on those idle CPUs.

The dominant computational model for scalable HPC applications, and for Astaroth, is Single Program, Multiple Data (SPMD). SPMD programs combine distributed-memory parallelism with some form of shared-memory parallelism. MPI is the leading library for message passing [122], OpenMP [32] is popular for multi-threading on multi-core CPUs and CUDA [102] or HIP [59] support accelerator programming. Most HPC performance tools have been developed to support SPMD applications. Measurement libraries are primarily implemented to execute with the application code and run on the same processes and computational resources. Performance measurements are made in the context of processes/threads and stored in the application memory, making them low-overhead and highly efficient. For the most part, measurements are node-local and performance data is collected and written at the end of the execution.

Unfortunately, the SPMD model and its implementation with MPI can constrain the development of new HPC tool functionality. Consider a simple case of wanting to compute performance statistics from performance data across all MPI processes while the application is executing. While MPI does offer support for both synchronous and asynchronous methods of computing these statistics, running MPI in threading mode is notably more difficult than its traditional synchronous counterpart, either because of the limited support for threaded MPI, or because it contorts the MPI programming, making it awkward to integrate more advanced analytics solutions and forcing the "shoe-horning" of arbitrary analytics functionality into an MPI program.

Further, there is another practical problem encountered on many leadership class clusters — some large HPC platforms disallow the running of multiple binaries (programs) on the nodes used by an application. In other cases, MPI environments do not allow the number of application ranks on a single node to exceed the number of available cores (known as *oversubscription*).

Therefore, running two different binaries on the same compute node required splitting the MPI ranks appropriately between the different applications. Figure 24 shows how we split the `MPI_COMM_WORLD` communicator between our two applications. In this process, SLURM passes certain ranks to each application, and it is the responsibility of the application to assign these ranks to it's own communicator group. This functionality works without modifications in the case where all ranks on a node are assigned to one application as well – in this scenario the application just copies all ranks to its new communicator group. All experiments were carried out on either the LUMI-G or MAHTI systems detailed in Table 6.

	LUMI-G	**MAHTI**
System	HPE Cray EX	Atos BullSequana XH2000
CPU	AMD EPYC 7653	AMD Rome 7H12
Total CPU Cores	64	128
Memory (GB)	512	256
# GPUs	8	4
GPU Arch	AMD MI250X	NVIDIA A100

Table 6. Architectures used for the Astaroth and LULESH simulations and experiments.

5.2.1 Special Case of Idle Cores. Heterogeneous HPC places an emphasis on the use of accelerated devices, primarily GPUs, for achieving performance gains. In some heterogeneous applications that take advantage of GPUs it can be the case that the CPUs (cores) on accelerated nodes are under-utilized during the execution. One motivation for our monitoring research is to take advantage of such situations by finding techniques to locate monitoring processes on the same nodes as the application. Suppose we consider a special case where an MPI application does not use all of the available cores on each node allocated to it. Instead, the cores remain idle for the entirety of the execution.

Without loss of generality, consider an MPI-based application configured to run with R total ranks on N nodes where each node has $r = R/N$ ranks per nodes (assume N evenly divides R). Let C be the number of CPU cores on a node and c be the number of cores not used by the application at all during execution. These unused CPU cores could be available to run monitoring processes.

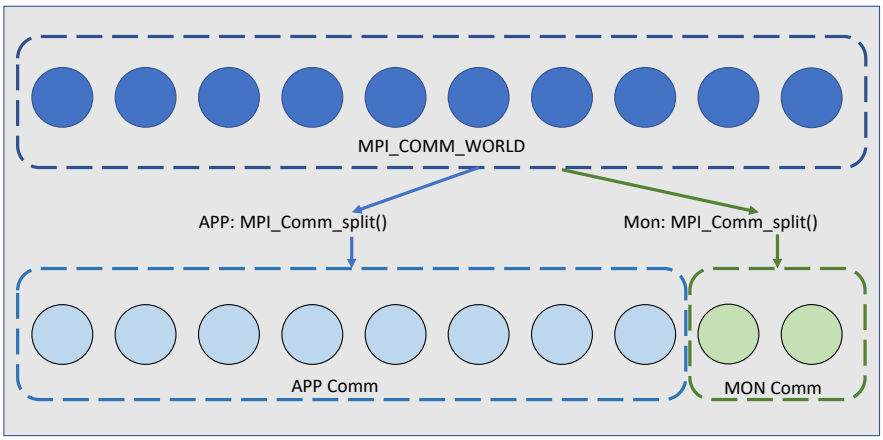

Figure 24. Example of how `MPI_Comm_split` works to run two applications on the same node.

Suppose M total monitoring processes are to run with $m = M/N$ monitoring processes on each node (assume N evenly divides M). The question is then how to create the monitoring processes and configure them to run with the application such that m are assigned to each node.

One way of doing this is with the *Multiple Program, Multiple Data* (MPMD) approach can be done entirely in MPI. The idea is to run the application with $R+M$ ranks and split `MPI_COMM_WORLD` into two communicators: one for the application (call it `APP`) and one for the monitor (call it `MON`). The application code must be modified to do the split and to use the `APP` communicator in place of `MPI_COMM_WORLD` throughout. When launching the computation, $r + m$ ranks must be allocated to each node. There are two potential downsides to this approach. First, it might be problematic to modify the application code, for several valid reasons. Second, it might be difficult to integrate the monitor code with the application code. This is necessary because a single binary that includes the application and monitor code is being run by the MPI processes. A simple example of this approach is shown in Figure 24.

A big advantage of the above approach is that it does not require anything special to be done by the job submission system. Another approach that does not involve changes to the application or monitor MPI communicators is to launch the application (with R ranks) and the monitor (with M ranks) as separate binaries at the same time on the same nodes where r application ranks and m monitor ranks are running on each node. This can be done using a script that is launched on N

nodes and then executes the monitor and application. Each would be able to use `MPI_COMM_WORLD` for MPI operations. Each would be its own binary. A different technique might be needed for the application to discover the monitor above. For this approach to be viable, the environment must allow two different binaries to run at the same time on the same resource set. Unfortunately, for some platforms, this is not the case.

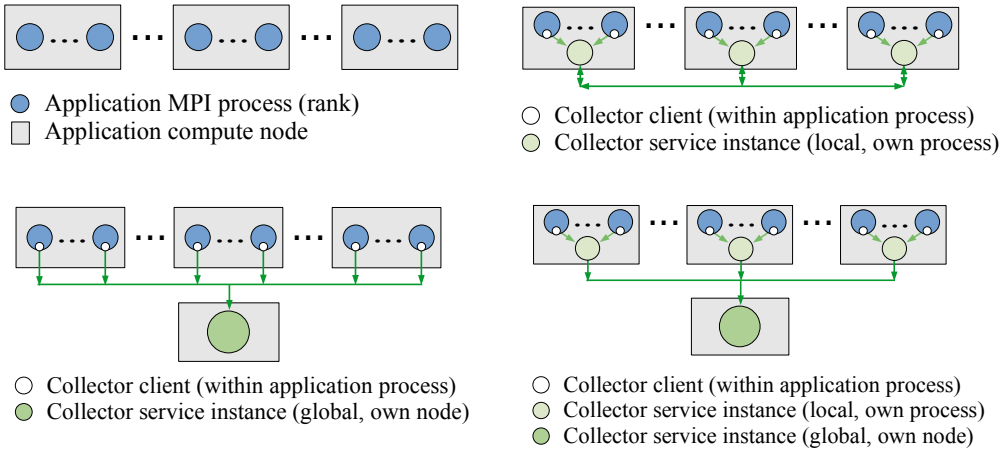

Figure 25. **(Top-Left)**: Standard MPI application. **(Bottom-Left)**: A SOMA *client* is embedded within each MPI rank (i.e., process). The *client* is called via a TAU plug-in. The application is launched with an additional node on which a SOMA *service* instance executes. Each *client* will interact with the *service*. **(Top-Right)**: The *client* is the same as Bottom-Left. However, the application is launched with another process on each application node where an *service* executes. Each *client* will interact with the local *service* instance. The *services* will interact with each other for distributed processing. **(Bottom-Right)**: The *client* and local *service* are the same as the Top-Right. An additional node is allocated for a global *service* instance. The *service* instances will interact with each other and with the global *service* instance.

5.2.1.1 LULESH. It was interesting to us to show the use of microservices in an application that could be configured similarly to Astaroth. For this purpose, we installed LULESH hydrodynamics proxy application [66, 65] on the CSC Mahti supercomputer and ran it successfully with SOMA in both the node-local and remote scenarios shown in Figure 25. In particular, we used an MPI-only version and allocated up to 64 ranks on each node, leaving 64 or more unused cores of the 128 cores available. We modified LULESH to launch with additional ranks per node and split the `MPI_COMM_WORLD` communicator to use certain ranks for LULESH and the rest for SOMA (see the top-right configuration in Figure 25).

These LULESH experiments were performed with a problem size of 45 (per domain) which is the equivalent of 5,832,000 elements. LULESH simulations must be run with a number of ranks

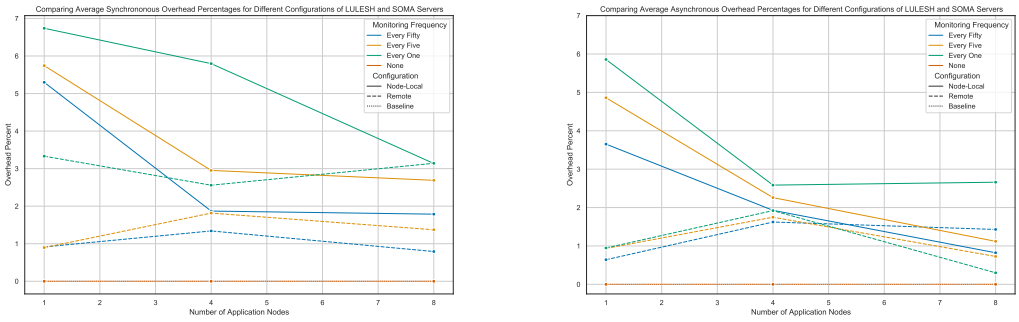

Figure 26. Synchronous overhead percent (Left) and asynchronous overhead percent (Right) based on the configuration of our monitoring service with LULESH on Mahti. The baseline is indicated by the orange dotted line. In the case of the remote configuration, we run the indicated number of application nodes and an extra node solely for the SOMA servers. When publish more frequently, and run the SOMA servers locally we see increased overhead. The dashed lines demonstrate our performance when publishing data in the remote node configuration, which tend to outperforming the node-local solid lines. Improvements are noted when using the non-blocking asynchronous calls over the blocking synchronous calls, especially in the four node case, when there are free cores.

that is also the cube of an integer. Because we used up to 64 ranks per node, while we increased the node count, we also had to calculate the ratio to keep the total ranks equal to the cube of an integer. We measure SOMA overhead by varying the monitoring frequency (how often both application and performance data is published to the servers), with every 50 application iterations, every 5 iterations, and every single iteration. Figure 26 demonstrates a comparison between using SOMA's blocking calls — `soma_publish`, `soma_commit_namespace` — and SOMA's non-blocking calls — `soma_publish_async`, `soma_commit_namespace_async`. For both of these experiments, overhead is calculated against a LULESH baseline, where it is run without any monitoring enabled. The network protocol used for all Mahti experiments was ofi+verbs. We see that in general, using a less frequent monitoring interval of every 50 iterations has the least overhead while monitoring every iteration generates the most overhead. The configuration of running the SOMA monitoring ranks on a remote node from the application does typically outperform the case where we run the SOMA monitoring ranks on the local node.

When only running on a single application node, we see a trend of greater overhead percentage than more nodes, but the total execution time remains within range. This is based on an average of 5 runs in each configuration. However, as we scale LULESH to run on an increasing number of compute nodes, we see the overhead begin to converge slightly in the synchronous case. Running the

SOMA servers on an extra allocated node likely utilizes completely free resources, even though there are free cores on the local node, perhaps allowing more free cores would yield better performance. Further tuning can be done when looking at message size, publishing frequency, and the ratio of SOMA server instances per rank to find the best performance in each scenario.

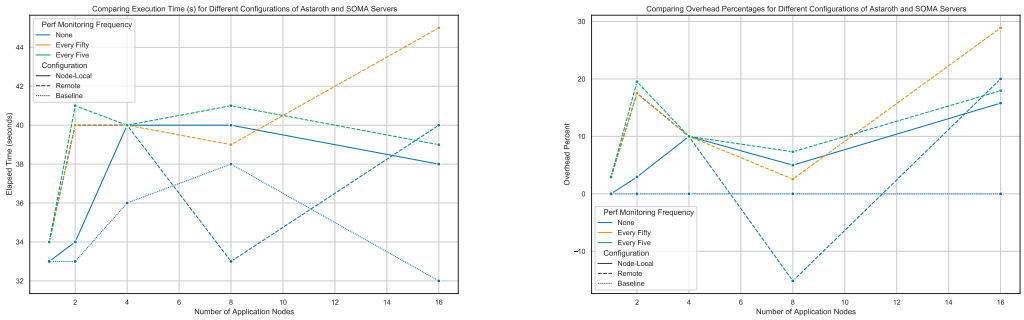

Figure 27. Execution time (Left) and percent overhead (Right) based on the configuration of our monitoring service for Astaroth on LUMI-G. In the case of the remote configuration, we run the indicated number of application nodes, and an extra node solely for the SOMA servers. We ran a baseline with no monitoring, application only monitoring, and performance + application monitoring. We see only a small increase in overhead when we combine performance monitoring with the application monitoring — when performance monitoring frequency is every 5 or 50 iterations.

5.2.1.2 Astaroth. We analyze the overhead of collecting application data via SOMA for Astaroth. The description and analysis of the data are in Section 4.4. For the purpose of measuring overhead, we conducted, first, baseline execution times for Astaroth without any monitoring. This baseline is pictured in Figure 27 with Monitoring Frequency = 0.

We scale Astaroth from one node up to 16 nodes on LUMI-G, which contains 8 GPUs per CPU node. In all scenarios we run 8 Astaroth ranks per node to utilize all of the GPUs available. The grid size of Astaroth must be adjusted for each scaled run, beginning with x, y, z dimensions of 256, 256, 256 for one rank and multiplying each dimension by the number of ranks starting with z, e.g., for 8 ranks x,y,z was 512, 512, 512, for 16 ranks it was 1024, 1024, 2048. Application data is structured as a Conduit node and configured to publish to the SOMA service instance every five application iterations. Performance data monitoring is conducted via the SOMA TAU plugin which converts TAU profiles to conduit nodes for the SOMA RPC calls. Publishing frequency is varied between every 5 iterations and every 50 iterations for the performance data. The communication protocol used for publishing SOMA data between nodes used was ofi+tcp — we are investigating other high-performance communication protocols for Mochi on LUMI-G.

We launch the Astaroth jobs in the same configurations as the baseline, but with SOMA ranks on either an extra ranks on an extra node, or extra ranks on each node already allocated. These ranks are reserved for the SOMA service instances, whether that be the application data or performance data service. On this additional node, we use the number of ranks equal to the number of Astaroth nodes, giving us an 8:1 ratio of Astaroth ranks to each type of SOMA service rank. This allows us to compare the results with the node-local version. For the node-local version, we run a SOMA service instance with an extra rank on each Astaroth node, also at an 8:1 process count ratio.

We can see in Figure 27 that for both configurations, there is a non-negligible amount of monitoring overhead. One potential reason for this is due to the use of the lower-performance ofi+tcp interconnect. Based on the significantly lower overheads we saw with the ofi+verbs network in our Mahti/LULESH experiments we anticipate significant performance improvement when we use the high-performance Cray network (ofi+cxi) on LUMI-G. On a positive note, we see very little increase in overhead when we conduct the performance data monitoring simultaneously with the application data monitoring, even at more frequent publication rates. Although these results may seem a bit underwhelming, we are early in our design, and there are many opportunities for optimization, beyond the network changes, that we believe will have a significant impact. For example, we are further scrutinizing the size of the data being sent, the serialization protocol used for the data, the frequency of publication, and the ratios of SOMA server instances per application rank, node count, and problem size. Analysis of the application telemetry data that we collect for Astaroth in these experiments with SOMA was described in Section 4.4.

5.3 Adaptive Feedback Potential

Chapter IV discussed the increase in heterogeneous machine learning workflows as an acceleration technique for domain science, and the importance of the ability to monitor and improve such a workflow with SOMA. While Section 5.2 focused on overhead analysis for an individual application, in this section we look at the monitoring overhead and also potential for adaptive feedback with a Deep Learning heterogenous workflow mini-app managed by RADICAL-Pilot. An advantage gained from the RADICAL-Pilot integration was that the splitting of MPI communicator groups that was required for the experiments in section 5.2 is no longer necessary. Since SOMA is treated as a first-class citizen, RADICAL-Pilot is able to create and schedule both SOMA tasks and multiple application tasks per compute node, each within their own MPI scopes as needed.

5.3.1 DeepDriveMD Mini-app Workflow. The DeepDriveMD workflow mini-app [70] models the computational patterns and behaviors of a Deep Learning oriented Moleculer Dynamics simulation workflow using DeepDriveMD (deep-learning driven molecular dynamics) [79]. The DeepDriveMD workflow mini-app consists of phases, each phase made up of four stages. The stages are (1) Simulation, (2) ML Training, (3) Model Selection, and (4) Agent (Inference), which must be run in order. The baseline workflow uses 12 simulation tasks and 1 task each for training, selection, and agent. We do not modify the number of tasks except for the training tasks as explicitly stated in Table 7. The simulation, training, and agent stages use both CPU cores and one GPU resource per task, whereas the selection stage runs only on the CPU. This is configured and managed by the RADICAL-EnTK (Ensemble Toolkit), which is a higher-level abstraction of RADICAL-Pilot functionality [12]. We use EnTK to schedule n number of phases in a row, within m number of concurrent pipelines, this can be done for any combination of $n \times m$, see Figure 28. We run three different experiment setups with the DeepDriveMD workflow mini-apps.

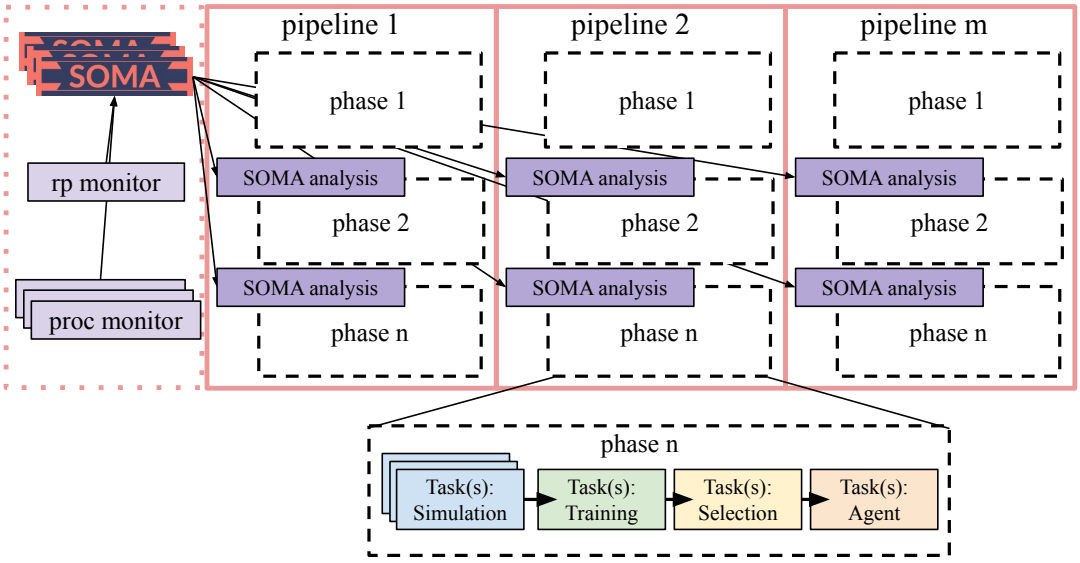

Figure 28. An illustration of how RADICAL-EnTK can launch SOMA monitoring tasks, one or more pipelines, and one or more phases of the DeepDriveMD mini-app workflow. Each phase is one full DDMD mini-app workflow comprised of multiple tasks, shown in the inset.

The first experiment is a tuning study, where we alter the number of cores allocated per task. We run $n = 6$ phases, and $m = 1$ pipelines. We can compare each phase of the workflow for

the effects the core assignment has on the runtime of each stage of the workflow. Details of the configurations for this experiment can be found in Table 7 under the “Tuning" column.

The second DeepDriveMD mini-app experiment involves conducting SOMA analysis to identify free resources during runtime. While EnTK cannot make adaptive changes during runtime, we can use SOMA to learn more about how to configure the workflow during each successive phase. We run $n = 4$ phases, and $m = 1$ pipelines. The setups that were run during the second experiment can be found in Table 7 under the “Adaptive" column. The number of training tasks listed in the Table are set *a priori*, as in the original workflow, but we conduct online SOMA analysis to be available between phases. Figure 28 depicts where the SOMA analysis fits within the flow of the adaptive experiments.

The third experiment set for the DeepDriveMD mini-app was scaling it up to more compute nodes to ensure SOMA monitoring could keep pace. In Scaling A and B we run $n = 1$ phases for $m = a$ pipelines of the workflow, where a is the number of application nodes. With one task for each of the training, selection, and agent stages and 12 tasks per simulation stage, this creates an over-subscription during the simulation stage, where each simulation stage (per pipeline) requires 12 GPUs, but there are only 6 available per node. In Scaling A we vary the ratio of SOMA server ranks to pipelines, and in Scaling B we keep the ratio steady, see Table 7.

Experiment	Tuning	Adaptive	Scaling A	Scaling B
Phases (n)	6	4	1	1
Pipelines (m)	1	1	64	64, 128, 256, 512
Application Nodes	2	2	64	64, 128, 256, 512
SOMA Nodes	1	1	1, 2, 4	4, 7, 13, 25
Num Simulation Tasks	12	12	12	12
Cores Per Simulation Task	1, 3 , 7	6	3	3
Num Training Tasks	1	1, 2, 4, 6	1	1
Cores Per Training Task	1, 3, 7	1	7	7
SOMA Ranks Per Namespace	2	2	16, 32, 64	64, 128, 256, 512
Monitoring Frequency (s)	60	60	60	60, 10
Configuration	shared	shared	shared, exclusive	shared, exclusive
Monitors	proc, RADICAL-Pilot			
System	SUMMIT			
Hardware	2 IBM Power 9 CPU + 6 NVIDIA Tesla V100 GPUs			

Table 7. DeepDriveMD Mini-app Experiment Summary

5.3.1.1 Monitoring Setup. In the DeepDriveMD Miniapp experiments, we implemented data collection from two sources. Similar to the OpenFOAM workflow in Section 4.6.2, we initialized

one RADICAL-Pilot monitoring client per workflow, scheduled onto the same node as the RADICAL-Pilot agent. Also similarly, we initialize one `/proc/` monitoring client per compute node for collection of CPU usage data, which takes a reading every minute, and calculates the current CPU utilization online. We did not use the TAU monitoring client in the DeepDriveMD Mini-app experiments. In most experiments, we sampled and published the performance data every 60 seconds. In Scaling B, to push the limits of SOMA, we increased the frequency to every 10 seconds. We also run baseline workflows with no SOMA nodes or monitoring for comparison.

5.3.2 DeepDriveMD Mini-app Analysis. For the DeepDriveMD mini-app experiments we use the collected data from the hardware (`/proc/`) and RADICAL-Pilot monitors. In section 5.3.2 we first discuss the results from the tuning and adaptive workflows. We then look at the scaling workflows and illustrate our discoveries about the overhead and effects of SOMA monitoring at scale.

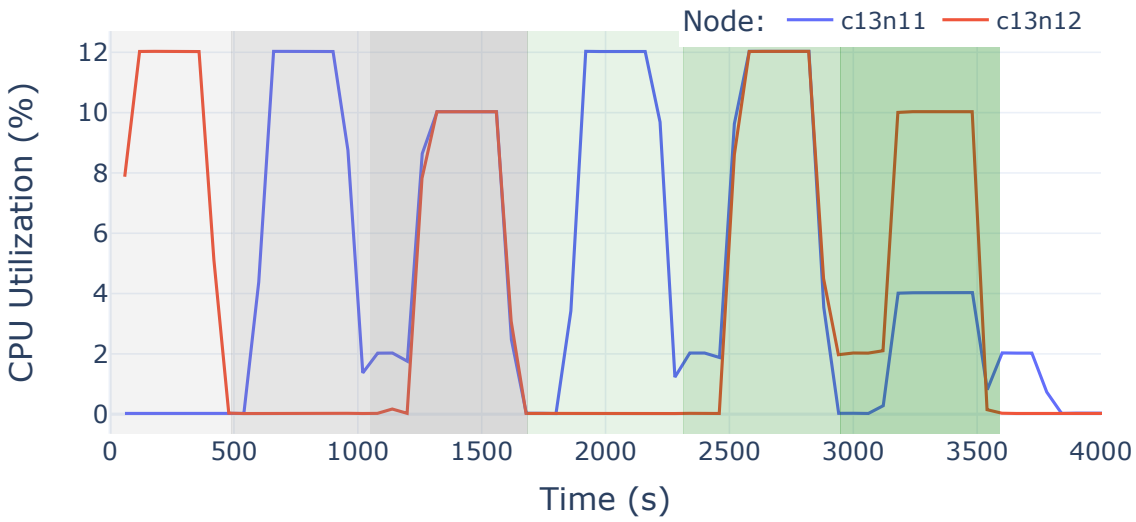

Figure 29. CPU utilization for the DeepDriveMD mini-app Tuning experiment shows low CPU utilization. The gray background shows phases where we assign 7 cores per training task, and green background we assign 3 cores per training task. The shading changes from light to medium to dark for 1, 3, and 7 cores per simulation task, respectively.

The tuning experiment results — where we have 6 phases of the workflow and change the number of cores per simulation and training task — are in Figure 29. We see that even when changing the number of cores that can be used per task, CPU utilization remains low. This is due to the fact that most of the work for the two longest stages, simulation and training, is done on the GPU. Therefore, learning that the effect of using fewer CPU cores per task was minimal, we next explored

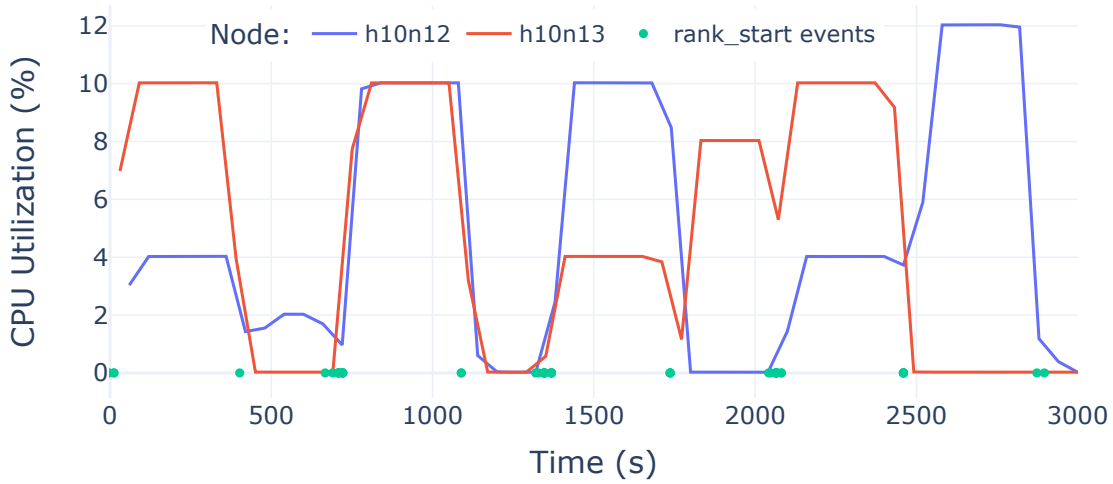

Figure 30. CPU utilization for the DDMD Adaptive Workflow.

parallelizing the training tasks in order to use more GPUs per node. Figure 30 shows the results of these other configuration changes. To parallelize the training tasks appropriately, we also resized the data and added additional `MPI_Reduce` calls to the tasks. While the full-scale DeepDriveMD workflow does not yet support such parallelization, we determined this a good use of the mini-app, to model potential changes without having to implement it in the full-scale workflow unless deemed worthwhile. SOMA can calculate the current CPU utilization online, but the integration to utilize that feedback with RADICAL-Pilot is still in progress.

Figure 31 depicts the pipeline execution times from the Scaling A experiment where we increased the number of pipelines to SOMA ranks from 1:1 to 8:1 to pinpoint any significant bottlenecks. Because we allocated extra nodes for SOMA and did not necessarily need to use every single core on those nodes for SOMA ranks we ran in two configurations. The *shared* configuration allowed RADICAL-Pilot to make use of any free cores on the SOMA nodes. The *exclusive* configuration reserved the nodes only for SOMA ranks and did not allow RADICAL-Pilot to schedule any application tasks on any available cores. Because of the oversubscription of simulation tasks in the workflow, each pipeline requires 12 GPUs during the simulation stage but there are only 6 per node. In the *shared* cases, RADICAL-Pilot was able to make use of the some of the free GPUs and cores on the SOMA nodes as SOMA runs on CPU-only and did not completely

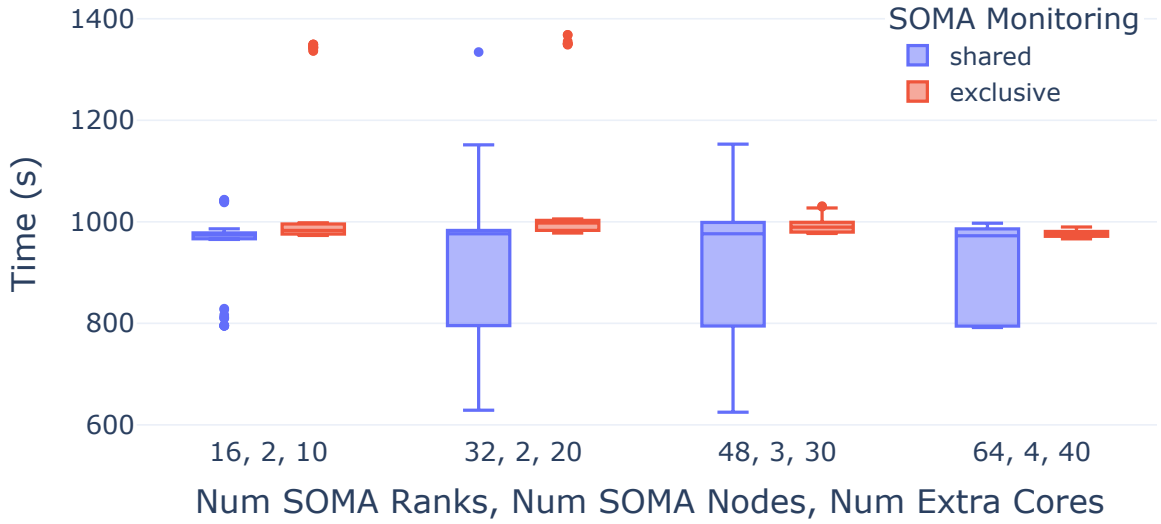

Figure 31. Runtimes for 64 pipelines of the DDMD Mini-app Scaling A experiment workflow. The oversubscription of GPUs causes more variability in scheduling and execution times in the shared configuration, but the ratio of SOMA ranks to pipelines does not have much effect.

fill the allocated SOMA nodes. This caused more variance in overall execution times but reduced the execution time for many of the pipelines.

Figure 32 demonstrates the distribution of pipeline execution times when we kept the ratio of SOMA ranks to pipelines at 1:1 and scaled up SOMA ranks and nodes, and application pipelines and nodes. Details of our configurations are given by the Scaling B column in table 7. We again make use of the flexible integration between RADICAL-Pilot and SOMA to run in both the *shared* and *exclusive* configurations described previously. We compare these with the baseline *none* configuration which was only m DeepDriveMDmini-app workflow pipelines running on m applications nodes with no SOMA monitoring or SOMA nodes. Again, we can actually see potential benefit from running with extra SOMA nodes in the *shared* configuration as RADICAL-Pilot's opportunistic scheduling system can make use of the available cores. The higher outliers in the *shared* configurations are due to the fact that RADICAL-Pilot is non-deterministic in scheduling and may make an inefficient placement during runtime that delays one or more pipelines.

Furthermore, when pushing SOMA to a higher monitoring frequency — every 10 seconds, up from every 60 seconds — we do start to see increased overhead costs. This is shown in Figure 32 under the *frequent-exclusive* label. In these results, the *frequent-exclusive* experiments are also exclusive in configuration, i.e., the extra cores on the SOMA nodes cannot be used by the application

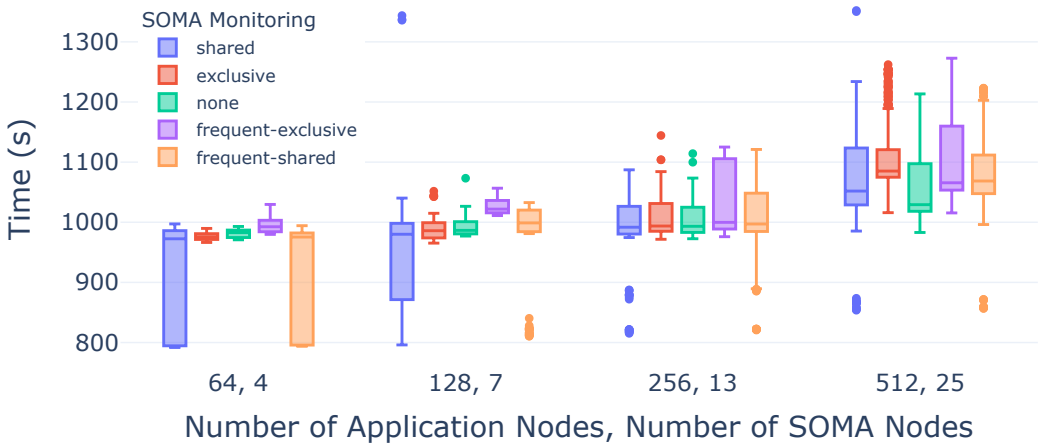

Figure 32. Runtimes for each pipeline of the DDMD Mini-app Scaling B experiment workflow. Shared indicates that free cores on SOMA allocated nodes were available to the application (RADICAL-Pilot can schedule application tasks on them), Exclusive: means that SOMA nodes were reserved for SOMA only. None indicates our baseline, where no SOMA monitoring was done and no SOMA nodes were allocated. (Note the x-axis — number of nodes — is a log scale).

tasks. This gives us a better direct comparison to the *exclusive* execution times because only the monitoring frequency changes. Some of this overhead can be mitigated when we run in the *frequent-shared* configuration. This allows for RADICAL-Pilot to utilize any extra cores during scheduling, thus increasing the variability, but reducing some execution times. When comparing the worst performance, *frequent-exclusive*, with the baseline we see approximately 1.4, 3.4, 3.2, and 4.6 percent runtime overhead for 64, 128, 256, and 512 nodes respectively. In the *shared* configuration, we actually see a reduction in runtime for 64, 128, and 256 nodes (6.5, 3.8, 1.1 percent, respectively) and an overhead cost of about 1.8 percent at 512 nodes.

5.4 Summary

Chapter V provided an overview of the performance of the SOMA monitoring framework. First, Section 5.2 presetend results on the monitoring overhead when running on remote or local compute nodes as well as comparing the blocking and non-blocking API calls. These experiments were completed with the Astaroth and Lulesh applications. Then Section 5.3 moved into how performance relates to adaptive feedback potential. While it is still important to measure the overhead in these scenarios, this step forward in enabling adaptive feedback gets closer to the ultimate goal of stopping or improving the simulation or workflow earlier, which will then realize performance gains. Figure 33 illustrates this end goal — a fully adaptive system with flexible inputs and outputs allowing for reconfiguration of the simulation. Both low overhead and the potential for adaptive feedback contribute to a more *performant* SOMA framework and answer the research question "How do we minimize the overhead?"

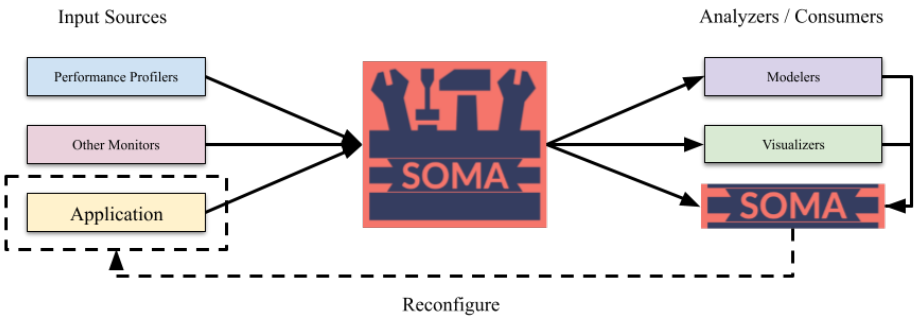

Figure 33. How SOMA adaptive feedback integration would work.

CHAPTER VI

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The challenge of online performance observation for HPC applications remains an important one to consider as simulations grow in size and complexity. This dissertation presented a solution for enabling performance observation for many different HPC applications across varying computer architectures. Chapter I provided an introduction to the research questions in the dissertation and an overview of the layout of each chapter of the dissertation. We introduced the definition for *online performance observation* and the requirements identified to support it in an HPC environment. This dissertation sought to answer the main research question (RQ) and subquestion (SQ):

RQ What are the requirements for an approach to online observation of simulation performance in an HPC environment?

SQ What implementation choices for a monitoring system can support the requirements identified by answering RQ?

In Chapter II we provided the background information necessary to both understand and properly motivate the importance of this dissertation work. Chapter II lays out the challenges associated with monitoring performance in an HPC environment. It explored the capabilities of current performance analysis, monitoring, and visualization tools. It also expands upon how we arrived at the three framework requirements listed in Chapter I, of (1) configurable, (2) flexible, and (3) performant.

In Chapter III we discussed the research direction we pursued in the design and development of a configurable microservices-based monitoring framework called SOMA and its deployment with HPC applications. We described how our design choices answer the question “How can we run on heterogeneous HPC ecosystems?" Based on the robust Mochi infrastructure, the SOMA monitoring system targets in situ performance and application observation, data collection, and analysis. We described the architecture and functionality of our SOMA prototype provide some examples for how it is implemented and used. The high degree of configurability possible with SOMA allows it to be

flexibly deployed to address in situ objectives. We demonstrated different SOMA configurations and where we have successfully run it to show the collection of data for diagnostics.

Chapter IV answered the question “How do we support different input and output?" The unique research contribution was the design of a flexible performance data model for the performance metrics to enable data representation and sharing, in this case, between SOMA client/server processes. An application data model was also created for in situ transmission and processing within SOMA. It was important to engage with the application team to define the application diagnostic data of interest. We also implemented data models for performance profile data, hardware metrics, and heterogeneous workflow state data. The Conduit [52] technology was used to implement all data models. We demonstrated collection and analysis of all of the data, with great future potential to optimize simulations and workflows.

Chapter V answered the question “How do we minimize the overhead?" We discussed the different techniques implemented to reduce the overhead of the SOMA monitoring service and make it performant. We compared blocking and non-blocking remote procedure calls when publishing data from client to server. We also compared the effects of different performance data publication rates on the time to solution for the application. Furthermore, we explored the different behavior and overhead when SOMA resources are local to the application or on reserved remote nodes. In the case of Astaroth, the opportunity to utilize idle CPU cores for SOMA operations was favorable. In the workflow experiments, tangible benefit was found when allowing RADICAL-Pilot to utilize idle cores on allocated SOMA nodes for the application tasks. All of these techniques showcase the configurable overhead reduction options based on the needs of the application team.

6.2 Future Work

For future work there are four main thrusts for building on the work done so far. The first is creating a more robust visualization integration. The second is using SOMA as a shared service, across independent tasks and workflows. The third is making use of machine learning models or performance modeling technology to fully automate the adaptive decision making. The fourth is extending the overhead analysis and finding specific recommended settings based on application or workflow characteristics. We detail these further in the next three subsections.

6.2.1 Robust Visualization Integration. Current advances in effective performance analysis and visualization have trended towards more exploratory and interactive approaches, as

Figure 34. Future Visualization Integration Possibilities for SOMA

evidenced by [21, 24, 118, 89], and many more. These aid the user in performing visual analytics, diagnosing performance issues and finding opportunities for speedup. These also lend themselves to interfacing with *in situ* performance data in a way that could make the best use of real-time decisions by a user. Addressing the complications, i.e., superfluous data, lack of data, distributed data, and limited resources, listed in Table 1 with a new perspective could warrant exciting results.

The integration between SOMA and Caliper [23] opens up potential for online analysis of metrics from sources that Caliper is already integrated with, such as Variorum, Raja [17] and Umpire. It would be beneficial to have the ability to "check in" on your application and see visualizations of these metrics in a user-friendly dashboard or jupyter notebook. This could be done through integrations with tools like Grafana [129], Hatchet [21] and Thicket [25] tools in conjunction with timeseries analysis and visualization functionality [155] that we recently introduced into Hatchet and Thicket, see Figure 34 .

This allows for analysis and visualization over time, before the simulation has completed. It discusses the straightforward approach of utilizing existing performance measurement tools, integrating with SOMA and it's canonical data model to bring the tools online, and visualizing that data in real time. The benefits of this project are to reduce data storage needs over the lifecycle of the application and also to be able to get a current "status" report of a long running application.

6.2.2 SOMA as a Shared Service. Another use case where SOMA would be useful is as a shared monitoring service across a cluster. It could gather application or workflow specific

performance data and provide it to a resource scheduling model such as fluxion [104]. One of the goals of the fluxion project is to implement elasticity of resources allocated to workflows. Because SOMA could provide real-time data from any of its sources to fluxion, an integration would enable application-specific or workflow-specific online decision making by the scheduling model.

6.2.3 Machine Learning for SOMA Adaptive Feedback. We would like to build on SOMA's capabilities for adaptive feedback purposes. It is technically possible to build an integration between SOMA and the application or workflow that allows feedback, and for example, Astaroth or RADICAL-Pilot could take advantage of this for runtime adaptation. Additionally, there is numerous work in the area of using machine learning (ML) for automated and fast empirical performance modeling [27, 112, 18, 156]. Many of these works could serve as an analysis model for SOMA to integrate with and use to provide feedback to the application. This could provide a generic approach for suggesting improvements to the application or workflow without requiring a built-in rule engine or user intervention.

6.2.4 SOMA Tuning Study. Now that SOMA functionality has been validated with both applications and workflows, we hope to find optimal configurations for minimal overhead, and test its performance in larger-scale scenarios. An extensive overhead and tuning analysis to find "recommended" settings or bottlenecks for SOMA based on application characteristics and additional factors such as Conduit::Node payload size would be extremely beneficial.

Another consideration in this area is the speed at which the application or workflow management system can consume any recommendations for adaptation. Considering how often an application can either be checkpointed or use additional processes to digest feedback is an open question. This solution becomes less of an issue when the workflow is managed with RADICAL-Pilot, which can use its own allocated resources to consume the feedback, and decide whether to interrupt any currently running tasks to make a change or not. There are also natural intervention points that make a lot of sense, such as right before a new task or phase begins executing.

REFERENCES CITED

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

[2] Anthony Agelastos, Benjamin A. Allan, Jim M. Brandt, Paul Cassella, Jeremy Enos, Joshi Fullop, Ann C. Gentile, Steve Monk, Nichamon Naksinehaboon, Jeff Ogden, Mahesh Rajan, Michael T. Showerman, Joel Stevenson, Narate Taerat, and Thomas W. Tucker. The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications. *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 154–165, 2014.

[3] Sean Ahern, Kathleen S. Bonnell, Eric Brugger, Hank Childs, Jeremy S. Meredith, and Brad Whitlock. Visit: a component based parallel visualization package. 2000.

[4] James P. Ahrens, Berk Geveci, and C. Charles Law. Paraview: An end-user tool for large-data visualization. In *The Visualization Handbook*, 2005.

[5] James P. Ahrens, Sébastien Jourdain, Patrick O'Leary, John M. Patchett, David Honegger Rogers, and Mark Roger Petersen. An image-based approach to extreme scale in situ visualization and analysis. *SC14: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 424–434, 2014.

[6] Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, and Ayse K. Coskun. Albadross: Active learning based anomaly diagnosis for production hpc systems. In *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 369–380, 2022.

[7] Burak Aksar, Efe Sencan, Benjamin Schwaller, Omar Aaziz, Vitus J. Leung, Jim Brandt, Brian Kulis, Manuel Egele, and Ayse K. Coskun. Runtime performance anomaly diagnosis in production hpc systems using active learning. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–14, 2024.

[8] Aymen Alsaadi, Logan Ward, Andre Merzky, Kyle Chard, Ian Foster, Shantenu Jha, and Matteo Turilli. Radical-pilot and parsl: Executing heterogeneous workflows on hpc platforms. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 27–34. IEEE, 2022.

[9] Edward Anderson. Lapack users' guide. 1987.

[10] Utkarsh Ayachit. The paraview guide: A parallel visualization application. 2015.

[11] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin Wozniak, Ian Foster, Mike Wilde, and Kyle Chard. Parsl: Pervasive parallel programming in python. In *28th ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2019.

[12] Jonathan Bader, Jim Belak, Matthew Bement, Matthew Berry, Robert Carson, Daniela Cassol, Stephen Chan, John Coleman, Kastan Day, Alejandro Duque, Kjiersten Fagnan, Jeff Froula, Shantenu Jha, Daniel S. Katz, Piotr Kica, Volodymyr Kindratenko, Edward Kirton, Ramani Kothadia, Daniel Laney, Fabian Lehmann, Ulf Leser, Sabina Lichołai, Maciej Malawski, Mario Melara, Elais Player, Matt Rolchigo, Setareh Sarrafan, Seung-Jin Sul, Abdullah Syed, Lauritz Thamsen, Mikhail Titov, Matteo Turilli, Silvina Caino-Lores, and Anirban Mandal. Novel approaches toward scalable composable workflows in hyper-heterogeneous computing environments. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 2097–2108. Association for Computing Machinery, 2023.

[13] Jonathan Bader, Joel Witzke, Soeren Becker, Ansgar Lößer, Fabian Lehmann, Leon Doehler, Anh Duc Vu, and Odej Kao. Towards advanced monitoring for scientific workflows. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 2709–2715. IEEE, 2022.

[14] Jonathan Bader, Nicolas Zunker, Soeren Becker, and Odej Kao. Leveraging reinforcement learning for task resource allocation in scientific workflows. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 3714–3719. IEEE, 2022.

[15] Vivekanandan Balasubramanian, Antons Treikalis, Ole Weidner, and Shantenu Jha. Ensemble toolkit: Scalable and flexible execution of ensembles of tasks. In *2016 45th International Conference on Parallel Processing (ICPP)*, volume 00, pages 458–463, Aug. 2016.

[16] Leilani Battle and Jeffrey Heer. Characterizing exploratory visual analysis: A literature review and evaluation of analytic provenance in tableau. *Computer Graphics Forum*, 38, 2019.

[17] David Beckingsale, Thomas R. W. Scogland, Jason Burmark, Richard D. Hornung, Holger E. Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, and Brian S. Ryujin. Raja: Portable performance for large-scale scientific applications. *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019.

[18] David A. Beckingsale, Olga Pearce, Ignacio Laguna, and Todd Gamblin. Apollo: Reusable models for fast, dynamic tuning of input-dependent code. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 307–316, 2017.

[19] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, chapter 26, pages 359–371. Morgan Kaufmann, Boston, 2012.

[20] Tal Ben-Nun, Todd Gamblin, Daisy S Hollman, Hari Krishnan, and Chris J Newburn. Workflows are the new applications: Challenges in performance, portability, and productivity. In *2020 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 57–69. IEEE, 2020.

[21] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: pruning the overgrowth in parallel profiles. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2019.

[22] David Boehme, Kevin Huck, Jonathan Madsen, and Josef Weidendorfer. The case for a common instrumentation interface for HPC codes. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 33–39, 2019.

[23] David Böhme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance introspection for hpc software stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016.

[24] Stephanie Brink, Ian Lumsden, Connor Scully-Allison, Katy Williams, Olga Pearce, Todd Gamblin, Michela Taufer, Katherine E. Isaacs, and Abhinav Bhatele. Usability and performance improvements in hatchet. *2020 IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 49–58, 2020.

[25] 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. Thicket: Seeing the performance experiment forest for the individual run trees. *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, 2023.

[26] Shirley Browne, Christine Deane, George Ho, and Philip Mucci. Papi: A portable interface to hardware performance counters. 1999.

[27] Alexandru Calotoiu, David Beckinsale, Christopher W. Earl, Torsten Hoefler, Ian Karlin, Martin Schulz, and Felix A. Wolf. Fast multi-parameter performance modeling. *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 172–181, 2016.

[28] Onur Cankur and Abhinav Bhatele. Comparative evaluation of call graph generation by profiling tools. In *ISC*, 2022.

[29] Robert Carson, Matt Rolchigo, John Coleman, Mikhail Titov, Jim Belak, and Matt Bement. Uncertainty quantification of metal additive manufacturing processing conditions through the use of exascale computing. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 380–383. Association for Computing Machinery, 2023.

[30] Hank Childs. Visit: An end-user tool for visualizing and analyzing very large data. 2011.

[31] John Coleman, Kellis Kincaid, Gerald L. Knapp, Benjamin Stump, and Alexander J. Plotkowski. ORNL/AdditiveFOAM: Release 1.0, June 2023.

[32] Leonardo Dagum and Ram Menon. OpenMP: An industry standard API for shared-memory programming. 1998.

[33] Sweta Dargad and Manmohan Singh. Rrdtool: A round robin database for network monitoring. *Journal of Computer Science*, 2017.

[34] Nan Ding and Samuel Williams. An instruction roofline model for gpus. *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–18, 2019.

[35] Estelle Dirand, Laurent Colombet, and Bruno Raffin. Tins: A task-based dynamic helper core strategy for in situ analytics. In *Asian Conference on Supercomputing Frontiers*, pages 159–178. Springer, 2018.

[36] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. Top500 supercomputer sites. *Supercomputer*, 13:89–111, 1997.

[37] Matthieu Dorier, Zhe Wang, Utkarsh Ayachit, Shane Snyder, Rob Ross, and Manish Parashar. Colza: Enabling elastic in situ visualization for high-performance computing simulations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 538–548. IEEE, 2022.

[38] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix A. Wolf. Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, 2011.

[39] Richard Todd Evans, William L. Barth, James C. Browne, Robert L. DeLeon, Thomas R. Furlani, Steven M. Gallo, Matthew D. Jones, and Abani K. Patra. Comprehensive resource use monitoring for hpc systems with tacc stats. *2014 First International Workshop on HPC User Support Tools*, pages 13–21, 2014.

[40] MPI Forum. Mpi specifications, 2023.

[41] Matteo Frigo and Steven G. Johnson. Fftw: an adaptive software architecture for the fft. *Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181)*, 3:1381–1384 vol.3, 1998.

[42] Supriya Garg, Julia Eunju Nam, I. V. Ramakrishnan, and Klaus Mueller. Model-driven visual analytics. *2008 IEEE Symposium on Visual Analytics Science and Technology*, pages 19–26, 2008.

[43] Michael Gerndt, Karl Fürlinger, and Edmond Kereku. Periscope: Advanced techniques for performance analysis. In *International Conference on Parallel Computing (ParCo)*, pages 15–26, September 2005.

[44] Alfredo Giménez, T. Gamblin, I. Jusufi, A. Bhatele, M. Schulz, P. Bremer, and B. Hamann. Memaxes: Visualization and analytics for characterizing complex memory performance behaviors. *IEEE Transactions on Visualization and Computer Graphics*, 24:2180–2193, 2018.

[45] Anshuman Goswami, Yuan Tian, Karsten Schwan, Fang Zheng, Jeffrey Young, Matthew Wolf, Greg Eisenhauer, and Scott Klasky. Landrush: Rethinking in-situ analysis for GPGPU workflows. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 32–41, 2016.

[46] Brian J. Gravelle, William David Nystrom, and Boyana Norris. Performance analysis with unified hardware counter metrics. *2022 IEEE/ACM International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 60–70, 2022.

[47] Brian J. Gravelle, William David Nystrom, Dewi Yokelson, and Boyana Norris. Enabling cache aware roofline analysis with portable hardware counter metrics. *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 75–81, 2021.

[48] William Gropp, Ewing L. Lusk, Nathan E. Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Comput.*, 22:789–828, 1996.

[49] Pascal Grosset, Jesus Pulido, and James Ahrens. Personalized in situ steering for analysis and visualization. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 1–6. 2020.

[50] W. Gu, G. Eisenhauer, E. Kraemer, K. Schwan, J. Stasko, J. Vetter, and N. Mallavarupu. Falcon: On-line monitoring and steering of large-scale parallel programs. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 422–429, 1995.

[51] Weiming Gu, Gary Bernard Eisenhauer, Eileen T. Kraemer, K. Schwan, John T. Stasko, Johannes Vetter, and Niru Mallavarupu. Falcon: on-line monitoring and steering of large-scale parallel programs. *Proceedings Frontiers '95. The Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 422–429, 1995.

[52] Cyrus Harrison, Matthew Larsen, Brian S. Ryujin, Adam J. Kunen, Arlie G. Capps, and Justin Privitera. Conduit: A successful strategy for describing and sharing data in situ. *2022 IEEE/ACM International Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*, pages 1–6, 2022.

[53] Mihael Hategan-Marandiuc, Andre Merzky, Nicholson Collier, Ketan Maheshwari, Jonathan Ozik, Matteo Turilli, Andreas Wilke, Justin M Wozniak, Kyle Chard, Ian Foster, et al. Psi/j: A portable interface for submitting, monitoring, and managing jobs. In *19th IEEE International Conference on eScience*. IEEE, 2023.

[54] Pieter Hintjens. Zeromq: Messaging for many applications. 2013.

[55] Kevin Huck and Allen Malony. Zerosum: User space monitoring of resource utilization and contention on heterogeneous hpc systems. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, SC-W '23, page 685–695, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Kevin A. Huck, Sameer Shende, Allen D. Malony, Hartmut Kaiser, Allan Porterfield, Robert J. Fowler, and Ron Brightwell. An early prototype of an autonomic performance environment for Exascale. In *International Workshop on Runtime and Operating Systems for Supercomputers*, 2013.

[57] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13:21–24, 2014.

[58] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Beyond the roofline: Cache-aware power and energy-efficiency modeling for multi-cores. *IEEE Transactions on Computers*, 66:52–58, 2017.

[59] Advanced Micro Devices Inc. HIP API documentation, 2022.

[60] Katherine E. Isaacs, P. Bremer, I. Jusufi, T. Gamblin, A. Bhatele, M. Schulz, and B. Hamann. Combing the communication hairball: Visualizing parallel execution traces using logical time. *IEEE Transactions on Visualization and Computer Graphics*, 20:2349–2358, 2014.

[61] Katherine E. Isaacs, Alfredo Giménez, Ilir Jusufi, Todd Gamblin, Abhinav Bhatele, Martin Schulz, Bernd Hamann, and Peer-Timo Bremer. State of the art of performance visualization. In *EuroVis*, 2014.

[62] Shantenu Jha and Allen D. Malony. Dynamic and adaptive monitoring and analysis for many-task ensemble computing. *2021 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 637–641, 2021.

[63] Bo Kågström, Per Ling, and Charles Van Loan. Gemm-based level 3 blas: high-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Softw.*, 24:268–302, 1998.

[64] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11. ACM, 2014.

[65] Ian Karlin. LULESH programming model and performance ports overview. 2012.

[66] Ian Karlin, Jeff Keasler, and J. Robert Neely. LULESH 2.0 updates and changes. 2013.

[67] Gregory Katsaros, Roland Kübert, and Georgina Gallizo. Building a service-oriented monitoring framework with rest and nagios. *2011 IEEE International Conference on Services Computing*, pages 426–431, 2011.

[68] Maximilian Keiff, Frederic Voigt, Anna Fuchs, Michael Kuhn, Jannek Squar, and Thomas Ludwig. Automated performance analysis tools framework for hpc programs. *Procedia Computer Science*, 207:1067–1076, 2022. Knowledge-Based and Intelligent Information and Engineering Systems: Proceedings of the 26th International Conference KES2022.

[69] Christopher Kelly, Sungsoo Ha, Kevin Huck, Hubertus Van Dam, Line Pouchard, Gyorgy Matyasfalvi, Li Tang, Nicholas D'Imperio, Wei Xu, Shinjae Yoo, et al. Chimbuko: A workflow-level scalable performance trace analysis tool. In *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, pages 15–19. 2020.

[70] Ozgur Ozan Kilic, Tianle Wang, Matteo Turilli, Mikhail Titov, Andre Merzky, Line Pouchard, and Shantenu Jha. Workflow mini-apps: Portable, scalable, tunable & faithful representations of scientific workflows, 2024.

[71] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the open trace format (otf). In Vassil N. Alexandrov, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science – ICCS 2006*, pages 526–533, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[72] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix A. Wolf. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Parallel Tools Workshop*, 2011.

[73] Andreas Knüpfer, Christian Rössel, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E Nagel, et al. Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[74] Pouya Kousha, D. KamalRajS., Hari Subramoni, Dhabaleswar Kumar Panda, Heechang Na, Trey Dockendorf, and Karen A. Tomko. Accelerated real-time network monitoring and profiling at scale using osu inam. *Practice and Experience in Advanced Research Computing*, 2020.

[75] Mahendra Kutare, Greg Eisenhauer, Chengwei Wang, Karsten Schwan, Vanish Talwar, and Matthew Wolf. Monalytics: online monitoring and analytics for managing large scale data centers. In *ICAC '10*, 2010.

[76] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *CACM*, 1978.

[77] Matthew Larsen, James P. Ahrens, Utkarsh Ayachit, Eric Brugger, Hank Childs, Berk Geveci, and Cyrus Harrison. The alpine in situ infrastructure: Ascending from the ashes of strawman. *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017.

[78] Chee Wai Lee, Allen D. Malony, and Alan Morris. TAUmon: Scalable Online Performance Data Analysis in TAU. In *Workshop on Productivity and Performance (PROPER 2010)*, September 2010.

[79] Hyungro Lee, Matteo Turilli, Shantenu Jha, Debsindhu Bhowmik, Heng Ma, and Arvind Ramanathan. DeepDriveMD: Deep-learning driven adaptive molecular simulations for protein folding. In *2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers (DLS)*, pages 12–19. IEEE, 2019.

[80] Matthew Leinhauser, René Widera, Sergei Bastrakov, Alexander Debus, Michael Bussmann, and Sunita Chandrasekaran. Metrics and design of an instruction roofline model for amd gpus. *ArXiv*, abs/2110.08221, 2021.

[81] Andre Luckow, Ioannis Paraskevakos, George Chantzialexiou, and Shantenu Jha. Hadoop on hpc: Integrating hadoop and pilot-based dynamic resource management. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1607–1616. IEEE, 2016.

[82] Andre Luckow, Mark Santcroos, Andre Merzky, Ole Weidner, Pradeep Mantha, and Shantenu Jha. P: a model of pilot-abstractions. In *2012 IEEE 8th International Conference on E-Science*, pages 1–10. IEEE, 2012.

[83] Jonathan R Madsen, Muaaz G Awan, Hugo Brunie, Jack Deslippe, Rahul Gayatri, Leonid Oliker, Yunsong Wang, Charlene Yang, and Samuel Williams. Timemory: modular performance analysis for hpc. In *International Conference on High Performance Computing*, pages 434–452. Springer, 2020.

[84] A. Malony, S. Shende, R. Bell, K. Li, L. Li, and N. Trebon. *Performance Analysis and Grid Computing*, chapter Advances in the TAU Performance System, pages 129–144. Kluwer, Norwell, MA, 2003.

[85] Allen Malony and Wolfgang Nagel. Open trace—the open trace format (otf) and open tracing for hpc. page 24, 01 2006.

[86] Allen D. Malony, Matthew Larsen, Kevin A. Huck, Chad Wood, Sudhanshu Sane, and Hank Childs. When parallel performance measurement and analysis meets in situ analytics and visualization. In *PARCO*, 2019.

[87] Allen D. Malony, Sameer S. Shende, Wyatt Spear, Chee Wai Lee, and Scott Biersdorff. Advances in the tau performance system. In *Parallel Tools Workshop*, 2011.

[88] Anirban Mandal, Paul Ruth, Ilya Baldin, Dariusz Krol, Gideon Juve, Rajiv Mayani, Rafael Ferreira Da Silva, Ewa Deelman, Jeremy Meredith, Jeffrey Vetter, et al. Toward an end-to-end framework for modeling, monitoring and anomaly detection for scientific workflows. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1370–1379. IEEE, 2016.

[89] Diogo Marques, Helder Duarte, Aleksandar Ilic, Leonel Sousa, Roman Belenov, Philippe Thierry, and Zakhar Matveev. Performance analysis with cache-aware roofline model in intel advisor. *2017 International Conference on High Performance Computing and Simulation (HPCS)*, pages 898–907, 2017.

[90] Diogo Marques, Aleksandar Ilic, Zakhar Matveev, and Leonel Sousa. Application-driven cache-aware roofline model. *Future Gener. Comput. Syst.*, 107:257–273, 2020.

[91] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.

[92] Nancy Matthew. Why visual analytics?

[93] Marta Mattoso, Jonas Dias, Kary ACS Ocana, Eduardo Ogasawara, Flavio Costa, Felipe Horta, Vitor Silva, and Daniel De Oliveira. Dynamic steering of hpc scientific workflows: A survey. *Future Generation Computer Systems*, 46:100–113, 2015.

[94] Andre Merzky, Pavlo Svirin, and Matteo Turilli. Panda and radical-pilot integration: Enabling the pilot paradigm on hpc resources. In *EPJ Web of Conferences*, volume 214, page 03057. EDP Sciences, 2019.

[95] Andre Merzky, Matteo Turilli, and Shantenu Jha. Raptor: Ravenous throughput computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 595–604. IEEE, 2022.

[96] Andre Merzky, Matteo Turilli, Manuel Maldonado, Mark Santcroos, and Shantenu Jha. Using pilot systems to execute many task workloads on supercomputers. In Dalibor Klusáček, Walfredo Cirne, and Narayan Desai, editors, *Job Scheduling Strategies for Parallel Processing: 22nd International Workshop, JSSPP 2018, Vancouver, BC, Canada, May 25, 2018, Revised Selected Paper*, pages 61–82. Springer, 2019.

[97] André Merzky, Matteo Turilli, Mikhail Titov, Aymen Al-Saadi, and Shantenu Jha. Design and performance characterization of radical-pilot on leadership-class platforms. *IEEE Transactions on Parallel and Distributed Systems*, 33:818–829, 2021.

[98] André Merzky, Matteo Turilli, Mikhail Titov, Aymen Al-Saadi, and Shantenu Jha. Design and performance characterization of radical-pilot on leadership-class platforms. *IEEE Transactions on Parallel and Distributed Systems*, 33:818–829, 2021.

[99] Aroon Nataraj, Allen D. Malony, Alan Morris, Dorian Arnold, and Barton Miller. TAUoverMRNet (ToM): A Framework for Scalable Parallel Performance Monitoring. In *International Workshop on Scalable Tools for High-End Computing (STHEC '08)*, 2008.

[100] Aroon Nataraj, Matthew Sottile, Alan Morris, Allen D Malony, and Sameer Shende. Tauoversupermon: low-overhead online parallel performance monitoring. In *Euro-Par 2007 Parallel Processing: 13th International Euro-Par Conference, Rennes, France, August 28-31, 2007. Proceedings 13*, pages 85–96. Springer, 2007.

[101] Patrick Jay Nichols. Nmsba: Continuous application benchmarking and analysis – caba. 1 2021.

[102] John R. Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *2008 IEEE Hot Chips 20 Symposium (HCS)*, pages 1–2, 2008.

[103] OpenMP. Openmp specifications, 2023.

[104] Tapasya Patki, Dong Ahn, Daniel Milroy, Jae-Seung Yeom, Jim Garlick, Mark Grondona, Stephen Herbein, and Thomas Scogland. Fluxion: A scalable graph-based resource model for hpc scheduling challenges. *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, 2023.

[105] Johannes Pekkilä, Miikka S. Väisälä, Maarit J. Käpylä, Matthias Rheinhardt, and Oskar Lappi. Scalable communication for high-order stencil computations using CUDA-aware MPI. *Parallel Computing*, 111:102904, 2022.

[106] Vincent Pillet, Jesús Labarta, Toni Cortes, and Sergi Girona. Paraver: A tool to visualize and analyze parallel code. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 17–31. mar, 1995.

[107] RADICAL Development Team. Radical-pilot: Integration with flux, 2023. https://github.com/radical-cybertools/radical.pilot/blob/devel/src/radical/pilot/agent/executing/flux.py, last accessed 2023-08-07.

[108] Srinivasan Ramesh. Performance observability and monitoring of high performance computing with microservices. 2022.

[109] Srinivasan Ramesh, Hank Childs, and Allen Malony. Serviz: A shared in situ visualization service. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 277–290, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[110] Srinivasan Ramesh, Robert Ross, Matthieu Dorier, Allen Malony, Philip Carns, and Kevin Huck. SYMBIOMON: A high-performance, composable monitoring service. In *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 332–342, 2021.

[111] Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, 2001.

[112] Marcus Ritter, Alexandru Calotoiu, Sebastian Rinke, Thorsten Reimann, Torsten Hoefler, and Felix Wolf. Learning cost-effective sampling strategies for empirical performance modeling. pages 884–895, 05 2020.

[113] Robert Robey and Yuliana Zamora. *Parallel and High Performance Computing.* Manning Publications, Shelter Island, NY, 2021.

[114] Robert B. Ross, George Amvrosiadis, Philip H. Carns, Charles D. Cranor, Matthieu Dorier, Kevin Harms, Gregory R. Ganger, Garth A. Gibson, Samuel Keith Gutierrez, Robert Latham, Robert W. Robey, Dana Robinson, Bradley W. Settlemyer, Galen M. Shipman, Shane Snyder, Jérome Soumagne, and Qing Zheng. Mochi: Composing data services for high-performance computing environments. *Journal of Computer Science and Technology*, 35:121–144, 2020.

[115] Thomas Röhl, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Likwid monitoring stack: A flexible framework enabling job specific performance monitoring for the masses. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 781–784, 2017.

[116] Lucas Mello Schnorr, Guillaume Huard, and Philippe Olivier Alexandre Navaux. Triva: Interactive 3d visualization for performance analysis of parallel applications. *Future Gener. Comput. Syst.*, 26:348–358, 2010.

[117] Lucas Mello Schnorr, Philippe Olivier Alexandre Navaux, and Benhur de Oliveira Stein. Dimvisual: Data integration model for visualization of parallel programs behavior. *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, 1:473–480, 2006.

[118] Connor Scully-Allison, Ian G. Lumsden, Katy Williams, Jesse Bartels, Michela Taufer, Stephanie Brink, Abhinav Bhatele, Olga Pearce, and Katherine E. Isaacs. Designing an interactive, notebook-embedded, tree visualization to support exploratory performance analysis. *ArXiv*, abs/2205.04557, 2022.

[119] Sangmin Seo, Abdelhalim Amer, Pavan Balaji, Cyril Bordage, George Bosilca, Alex Brooks, Philip Carns, Adrián Castelló, Damien Genet, Thomas Herault, Shintaro Iwasaki, Prateek Jindal, Laxmikant V. Kalé, Sriram Krishnamoorthy, Jonathan Lifflander, Huiwei Lu, Esteban Meneses, Marc Snir, Yanhua Sun, Kenjiro Taura, and Pete Beckman. Argobots: A lightweight low-level threading and tasking framework. *IEEE Transactions on Parallel and Distributed Systems*, 29(3):512–526, 2018.

[120] S. Shende and A. Malony. The TAU Parallel Performance System. *International Journal of High Performance Computing Applications*, 20(2):287–331, 2006.

[121] Sameer S Shende and Allen D Malony. The tau parallel performance system. *The International Journal of High Performance Computing Applications*, 20(2):287–311, 2006.

[122] Marc Snir, Steve W. Otto, David W. Walker, Jack J. Dongarra, and Steven Huss-Lederman. MPI: The complete reference. 1996.

[123] J. Soumagne, D. Kimpe, J. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. Ross. Mercury: Enabling remote procedure call for high-performance computing. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, 2013.

[124] Jeanette Sperhac, Robert L DeLeon, Joseph P White, Matthew Jones, Andrew E Bruno, Renette Jones Ivey, Thomas R Furlani, Jonathan E Bard, and Vipin Chaudhary. Towards performant workflows, monitoring and measuring. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9. IEEE, 2020.

[125] Piper Stull-Lane. Define analytics: The changing role of bi's favorite catch-all term.

[126] Hari Subramoni, Albert Mathews Augustine, Mark Daniel Arnold, Jonathan L. Perkins, Xiaoyi Lu, Khaled Hamidouche, and Dhabaleswar Kumar Panda. Inam2: Infiniband network analysis and monitoring with mpi. In *Information Security Conference*, 2016.

[127] Cristian Tapus, I-Hsin Chung, and Jeffrey K Hollingsworth. Active harmony: Towards automated performance tuning. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 44–44. IEEE, 2002.

[128] Astaroth Development Team. Astaroth Repository, 2020.

[129] Grafana Development Team. Grafana documentation, 2022.

[130] Intel Development Team. Intel vtune profiler performance analysis cookbook, 2021.

[131] Perfetto Development Team. Perfetto, August 2022.

[132] Survey Development Team. Overview survey, 2022.

[133] Tableau Development Team. Tableau and big data: An overview.

[134] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 157–173, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[135] Rafael Keller Tesser and Philippe Olivier Alexandre Navaux. Dimvhcm: An on-line distributed monitoring data collection model. *2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 37–41, 2012.

[136] The OpenFOAM Foundation. OpenFOAM, open source software for Computational Fluid Dynamics (CFD), 2023.

[137] Mikhail Titov, Matteo Turilli, Andre Merzky, Thomas Naughton, Wael Elwasif, and Shantenu Jha. Radical-pilot and pmix/prrte: Executing heterogeneous workloads at large scale on partitioned hpc resources. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 88–107. Springer, 2022.

[138] Matteo Turilli, Vivek Balasubramanian, Andre Merzky, Ioannis Paraskevakos, and Shantenu Jha. Middleware building blocks for workflow systems. *Computing in Science & Engineering*, 21(4):62–75, 2019.

[139] Matteo Turilli, Feng Liu, Zhao Zhang, Andre Merzky, Michael Wilde, Jon Weissman, Daniel S Katz, and Shantenu Jha. Integrating abstractions to enhance the execution of distributed applications. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 953–962. IEEE, 2016.

[140] Matteo Turilli, Andre Merzky, Thomas Naughton, Wael Elwasif, and Shantenu Jha. Characterizing the performance of executing many-tasks on summit. In *2019 IEEE/ACM Third Annual Workshop on Emerging Parallel and Distributed Runtime Systems and Middleware (IPDRM)*, pages 18–25. IEEE, 2019.

[141] Matteo Turilli, Mark Santcroos, and Shantenu Jha. A comprehensive perspective on pilot-job systems. *ACM Computing Surveys (CSUR)*, 51(2):43, 2018.

[142] Miikka S. Väisälä, Johannes Pekkilä, Maarit J. Käpylä, Matthias Rheinhardt, Hsien Shang, and Ruben Krasnopolsky. Interaction of Large- and Small-scale Dynamos in Isotropic Turbulent Flows from GPU-accelerated Simulations. *The Astrophysical Journal*, 907(2):83, February 2021.

[143] Yunsong Wang, Charlene Yang, Steven Andrew Farrell, Thorsten Kurth, and Samuel Williams. Hierarchical roofline performance analysis for deep learning applications. *ArXiv*, abs/2009.05257, 2020.

[144] Jorn Warnecke, Maarit J. Korpi-Lagg, Frederick A.Gent, and Matthias Rheinhardt. Numerical evidence for a small-scale dynamo approaching solar magnetic Prandtl numbers. *Research Square*, 2022.

[145] Gunther H. Weber, Hank Childs, and Jeremy S. Meredith. Recent advances in visit: Parallel crack-free isosurface extraction. 2012.

[146] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, 2009.

[147] Felix Wolf and Bernd Mohr. Epilog binary trace-data format. 01 2004.

[148] Felix Wolf, Brian Wylie, Erika Ábrahám, Daniel Becker, Wolfgang Frings, Karl Fürlinger, Markus Geimer, Marc-Andre Hermanns, Bernd Mohr, Shirley Moore, Matthias Pfeifer, and Zoltan Szebenyi. Usage of the SCALASCA toolset for scalable performance analysis of large-scale parallel applications. In *Proc. of the 2nd HLRS Parallel Tools Workshop*, pages 157–167, Stuttgart, Germany, July 2008. Springer.

[149] Chad Wood. Online monitoring for high-performance computing systems. 2021.

[150] Chad Wood, Matthew Larsen, Alfredo Giménez, Kevin A. Huck, Cyrus Harrison, Todd Gamblin, and Allen D. Malony. Projecting performance data over simulation geometry using sosflow and alpine. In *ESPT/VPA@SC*, 2017.

[151] Chad Wood, Sudhanshu Sane, Daniel A. Ellsworth, Alfredo Giménez, Kevin A. Huck, Todd Gamblin, and Allen D. Malony. A scalable observation system for introspection and in situ analytics. *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT)*, pages 42–49, 2016.

[152] Charlene Yang, Rahulkumar Gayatri, Thorsten Kurth, Protonu Basu, Zahra Ronaghi, A Oyelese Adetokunbo, Brian Friesen, Brandon Cook, Douglas Doerfler, Leonid Oliker, Jack R. Deslippe, and Samuel Williams. An empirical roofline methodology for quantitatively assessing performance portability. *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 14–23, 2018.

[153] Charlene Yang, Thorsten Kurth, and Samuel Williams. Hierarchical roofline analysis for gpus: Accelerating performance optimization for the nersc-9 perlmutter system. *Concurrency and Computation: Practice and Experience*, 32, 2020.

[154] Ahmad Yasin. A top-down method for performance analysis and counters architecture. *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 35–44, 2014.

[155] Dewi Yokelson, David Bohme, Stephanie Brink, Olga Pearce, and Allen Malony. Poster: Timeseries visualization of performance metrics, May 2024.

[156] Dewi Yokelson, Marc Robert Joseph Charest, and Ying Wai Li. Hpc application performance prediction with machine learning on new architectures. In *Proceedings of the 2023 on Performance EngineeRing, Modelling, Analysis, and VisualizatiOn Strategy*, PERMAVOST '23, page 1–8, New York, NY, USA, 2023. Association for Computing Machinery.

[157] Dewi Yokelson, Oskar Lappi, Srinivasan Ramesh, Miikka Vaisala, Kevin Huck, Touko Puro, Boyana Norris, Maarit Korpi-Lagg, Keijo Heljanko, and Allen D. Malony. Observability, monitoring, and in situ analytics in exascale applications. *Cray User Group*, 2023.

[158] Dewi Yokelson, Oskar Lappi, Srinivasan Ramesh, Miikka S. Väisälä, Kevin Huck, Touko Puro, Boyana Norris, Maarit Korpi-Lagg, Keijo Heljanko, and Allen D. Malony. Soma: Observability, monitoring, and in situ analytics for exascale applications. *Concurrency and Computation: Practice and Experience*, Special Issue Paper(Early View):e8141, June 2024.

[159] Xuechen Zhang, Hasan Abbasi, Kevin Huck, and Allen D Malony. Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows. *Procedia Computer Science*, 80:1507–1518, 2016.

[160] Zhiyuan Zhang, Kevin T. McDonnell, and Klaus Mueller. A network-based interface for the exploration of high-dimensional data spaces. *2012 IEEE Pacific Visualization Symposium*, pages 17–24, 2012.

[161] Fang Zheng, Hongfeng Yu, Can Hantas, Matthew Wolf, Greg Eisenhauer, Karsten Schwan, Hasan Abbasi, and Scott Klasky. Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference-aware execution. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.