PERFORMANCE OBSERVABILITY AND MONITORING OF HIGH PERFORMANCE COMPUTING WITH MICROSERVICES

by

SRINIVASAN RAMESH

A DISSERTATION

Presented to the Department of Computer and Information Science
and the Division of Graduate Studies of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

June 2022

DISSERTATION APPROVAL PAGE

Student: Srinivasan Ramesh

Title: Performance Observability and Monitoring of High Performance Computing With Microservices

This dissertation has been accepted and approved in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science by:

Allen Malony	Chair
Boyana R. Norris	Core Member
Hank R. Childs	Core Member
Robert B. Ross	Core Member
Dare Baldwin	Institutional Representative

and

Krista Chronister	Vice Provost of Graduate Studies

Original approval signatures are on file with the University of Oregon Division of Graduate Studies.

Degree awarded June 2022

DISSERTATION ABSTRACT

Srinivasan Ramesh

Doctor of Philosophy

Department of Computer and Information Science

June 2022

Title: Performance Observability and Monitoring of High Performance Computing With Microservices

Traditionally, High Performance Computing (HPC) software has been built and deployed as bulk-synchronous, parallel executables based on the message-passing interface (MPI) programming model. The rise of data-oriented computing paradigms and an explosion in the variety of applications that need to be supported on HPC platforms have forced a re-think of the appropriate programming and execution models to integrate this new functionality. In situ workflows demarcate a paradigm shift in HPC software development methodologies enabling a range of new applications — from user-level data services to machine learning (ML) workflows that run alongside traditional scientific simulations.

By tracing the evolution of HPC software development over the past 30 years, this dissertation identifies the key elements and trends responsible for the emergence of coupled, distributed, in situ workflows. This dissertation's focus is on coupled in situ workflows involving composable, high-performance microservices. After outlining the motivation to enable performance observability of these services and why existing HPC performance tools and techniques can not be applied in this context, this dissertation proposes a solution wherein a set of techniques gathers, analyzes, and orients performance data from different sources to generate observability. By leveraging microservice components initially designed to build high performance data services, this dissertation demonstrates their broader applicability for building and deploying performance monitoring and visualization as services within an in situ workflow. The results from this dissertation suggest that: (1) integration of performance data from different sources is vital to understanding the performance of service components, (2) the in situ (online) analysis of this performance data is needed to enable the adaptivity of distributed components and manage monitoring data volume, (3) statistical modeling combined with performance observations can help generate better service configurations, and (4) services are a promising architecture choice for deploying in situ performance monitoring and visualization functionality. This dissertation includes previously published and co-authored material and unpublished co-authored material.

CURRICULUM VITAE

NAME OF AUTHOR: Srinivasan Ramesh

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR, USA
Birla Institute of Technology and Science, Pilani, Rajasthan, India

DEGREES AWARDED:

Master of Science, Computer and Information Science, 2018, University of Oregon
Bachelor of Engineering (Hons.), Computer Science, 2014, Birla Institute of Technology and Science

AREAS OF SPECIAL INTEREST:

High Performance Computing
Performance Engineering
Distributed Systems
Accelerator Programming

PROFESSIONAL EXPERIENCE:

Research Collaborator/Affiliate, Brookhaven National Laboratory Computation and Data Driven Discovery Division, 2021, Advisor: Shantenu Jha, Matteo Turilli

Research Collaborator/Affiliate, Argonne National Laboratory Mathematics and Computer Science Division, 2020, Advisor: Robert B. Ross, Philip Carns

W.J Cody Summer Associate, Argonne National Laboratory Mathematics and Computer Science Division, 2019, Advisor: Robert B. Ross, Philip Carns

W.J Cody Summer Associate, Argonne National Laboratory Mathematics and Computer Science Division, 2018, Advisor: Swann Perarnau, Sridutt Bhalachandra

Computation Student Intern, Lawrence Livermore National Laboratory Computation and Scientific Computing Division, 2017, Advisor: Martin Schulz, David Boehme, Tapasya Patki

Graduate Research Assistant, University of Oregon, Advisor: Allen Malony, 2018, 2019, 2020, 2021, 2022

Graduate Research Assistant, University of Oregon, Advisor: Sameer Shende, Allen Malony, 2016, 2017, 2018

Project Assistant, Indian Institute of Science Department of Computational and Data Sciences, 2015-2016

Software Development Engineer 1, Amazon, 2014-2015

GRANTS, AWARDS AND HONORS:

Lokey Dissertation Fellowship, University of Oregon, 2021
General University Scholarship, University of Oregon, 2020
W.J. Cody Summer Associate, Argonne National Laboratory, 2018, 2019
Selected to attend the Argonne Training Program for Extreme-Scale Computing (ATPESC), 2018
Selected to attend the International High Performance Computing Summer School (IHPCSS), 2017
Best Paper Award, European MPI Users' Group Meeting (EuroMPI), 2017
Selected as a Student Volunteer, Supercomputing Conference (SC), 2017, 2018, 2019, 2020, 2021
Selected as a Student Volunteer, International Conference for High Performance Computing, Networking, Storage, and Analysis (SC), 2017, 2018, 2019, 2020, 2021
Received a Student Travel Grant to attend the MVAPICH Users Group Conference (MUG), 2016, 2017, 2019

PUBLICATIONS:

Ramesh, S., and Malony, A. SOMA: A Service-based Observability, Monitoring, and Analytics Framework for HPC. **In preparation.**

Ramesh, S., Titov, M., Turilli, M., Jha, S., and Malony, A. Performance Monitoring for Adaptive High Performance Computing Ensembles. **In preparation.**

Ramesh, S., Childs, H., and Malony, A. SERVIZ: A Shared, In Situ Visualization Service. **Under review.**

Ramesh, S., Ross, R., Dorier, M., Malony, A., Carns, P., and Huck, K. (2021). SYMBIOMON: A High-Performance, Composable Monitoring Service. *International Conference on High Performance Computing, Data, and Analytics (HiPC).*

Ramesh, S., Malony, A., Ross, R., Dorier, M., Carns, P., and Huck, K. (2021). SYMBIO: Enabling Observability and Adaptivity of High Performance Data Services. *A Poster at International Conference on High Performance Computing, Data, and Analytics (HiPC).*

Ramesh, S., Malony, A., Ross, R., Dorier, M., Carns, P., and Huck, K. (2021). SYMBIOMON: A High Performance, Composable Monitoring Service for Online Application Introspection and Adaptation. *A Poster at The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC).*

Ramesh, S., Malony, A., Carns, P., Ross, R., Dorier, M., Soumagne, J., and Snyder, S. (2021). SYMBIOSYS: A Methodology for Performance Analysis of Composable HPC Data Services. *International Parallel and Distributed Processing Symposium (IPDPS).*

Ramesh, S., Carns, P., Ross, R., Snyder, S., and Malony, A. (2019). Profiling Composable Data Services. *WIP Session Talk at International Parallel Data Systems Workshop (PDSW) at SC19.*

Malony, A., Ramesh, S., Huck, K., Chaimov, N., and Shende, S. (2019). A Plugin Architecture for the TAU Performance System. *International Conference on Parallel Processing (ICPP).*

Malony, A., Ramesh, S., Huck, K., Wood, C., and Shende, S. (2019). Towards Runtime Analytics in a Parallel Performance System. *International Conference on High Performance Computing & Simulation (HPCS).*

Ramesh, S., Perarnau, S., Bhalachandra, S., Malony, A., and Beckman, P. (2019). Understanding the Impact of Dynamic Power Capping on Application Progress. *International Parallel and Distributed Processing Symposium (IPDPS).*

ACKNOWLEDGEMENTS

I would like to express my sincerest gratitude to my advisor, Dr. Allen Malony. I thank Dr. Malony for granting me significant academic freedom and access to exciting research opportunities throughout my graduate studies. It is a privilege to have had him as my advisor. Under his tutelage, this Ph.D. journey has been as much fun as intellectually rewarding. I thank Dr. Hank Childs and Dr. Boyana Norris for supporting me throughout my graduate milestones. I appreciate the lessons in academic writing that I learned by working with Dr. Hank Childs. I thank Dr. Sameer Shende for his encouragement, technical advice, and teaching me how to be a more productive student researcher. Under the mentorship of Dr. Robert Ross, Dr. Philip Carns, and Dr. Matthieu Dorier at Argonne National Laboratory, I found an interesting research topic to pursue that would ultimately be the grounds for my thesis work. I am grateful to them for helping me shape my research agenda. I give Dr. Philip Carns credit for the initial version of Margo's distributed callpath profiling implementation that I took forward. I thank Dr. Kevin Huck for his assistance on the various research projects we collaborated on. I would also like to thank my mentors from my internships at the Department of Energy labs for their encouragement, feedback, and continued support. I would like to thank Cheri Smith for being responsive when I needed her help. This journey would not have been near as fun without having my dearest friends around. I miss the late-night conversations with Dr. Monil when he was a graduate student here. Pranit Shah has been a constant and undying source of positivity and encouragement. I thank Julianne Shepard for her affection, care, and encouragement when things were difficult.

To my father and my mother, for whom every small success I had was a joy; my brother, for supporting me throughout my journey here; my uncle, without whose encouragement I would not have pursued a graduate degree in the first place.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. Main Research Question 5

1.2. Chapter II — Evolution of HPC Software Development and Accompanying Changes in Performance Tools 7

1.3. Chapter III — Identifying What to Observe and Monitor 7

1.4. Chapter IV — Selecting and Combining Sources of Performance Data 8

1.5. Chapter V — Ubiquitous Monitoring of Services and In Situ Workflow Components 10

1.6. Chapter VI — Extending the Applicability of High Performance Microservices 11

II. EVOLUTION OF HPC SOFTWARE DEVELOPMENT AND ACCOMPANYING CHANGES IN PERFORMANCE TOOLS 13

2.1. Introduction 13

2.2. Background 15

2.2.1. MPI: The Dominant Distributed Programming Model 16

2.2.2. Shared-Memory Programming Models 17

2.2.3. Other Programming Models 17

2.2.4. Performance Analysis Tools 17

2.2.4.1. MPI Performance Analysis 18

2.2.4.2. Shared-Memory Performance Analysis 18

2.3. Definitions 19

2.3.1. Module 19

2.3.2. Component 19

Chapter Page

2.3.3. Service 19

2.3.4. Composition 20

2.4. Application Complexity and Modularization 20

2.4.1. Claim 20

2.4.2. Reasoning 20

2.4.2.1. Simulation Scale and Fidelity 21

2.4.2.2. Range of Applications and Platforms 21

2.4.2.3. Structure of Modern Scientific Research Teams . . . 22

2.5. Process-Local Modularization 22

2.5.1. Important Trends in the Computing Industry 22

2.5.2. Component Software For HPC 23

2.5.2.1. CCA Model 24

2.5.2.2. CCA: Performance Measurement 27

2.5.2.3. CCA: Performance Monitoring and Optimization . . 30

2.5.3. Other HPC Component Frameworks 32

2.5.3.1. High-Performance Grid Component Frameworks . . 32

2.5.3.2. Low-Level Component Framework (L2C) 33

2.5.3.3. directMOD Component Framework 34

2.5.4. Comparing Component Frameworks 34

2.5.5. Scientific Computing Frameworks 35

2.5.6. Modularization of MPI Libraries 37

2.5.6.1. LAM/MPI 37

2.5.6.2. OpenMPI 37

2.5.6.3. MVAPICH2 38

2.5.7. Tools for Performance Data Exchange 39

2.5.7.1. MPI Tools 40
2.5.7.2. OpenMP Tools 41
2.5.7.3. PAPI SDE 41
2.5.7.4. Comparing Techniques for Performance Data Exchange 42
2.6. Distributed Modularization 43
2.6.1. Overview 43
2.6.2. Important Trends in the Computing Industry 45
2.6.3. Composition Model 46
2.6.3.1. Distributed CCA Frameworks 48
2.6.3.2. HPC Data Services 51
2.6.3.3. In-situ Visualization and Analysis 57
2.6.3.4. HPC Ensemble Frameworks 59
2.6.4. Resource Allocation and Elasticity 62
2.6.4.1. Resource Allocation 62
2.6.4.2. Resource Elasticity 66
2.6.5. Data Management Strategy 69
2.6.5.1. MxN Problem 69
2.6.5.2. Data Staging and I/O Frameworks 72
2.6.6. Performance Tools 76
2.6.6.1. Performance Measurement 76
2.6.6.2. Performance Monitoring and Analysis 78
2.6.6.3. Control and Adaptivity 81
2.7. Trends and Open Areas 82
2.7.1. Trends 82
2.7.2. Open Areas 84

Chapter Page

2.7.2.1. Performance Instrumentation & Measurement . . . 84

2.7.2.2. Performance Monitoring & Analysis 85

2.7.2.3. Control & Adaptivity 86

2.8. Summary 86

III.IDENTIFYING WHAT TO OBSERVE AND MONITOR 88

3.1. Introduction 88

3.1.1. Benefits of Microservice Architectures 89

3.1.2. Challenges Posed by Microservice Architectures 90

3.2. High Performance Microservices: A Background 93

3.2.1. Mochi: A Background 93

3.2.2. Mochi: Core Components 96

3.2.2.1. Argobots 96

3.2.2.2. Mercury 97

3.2.2.3. Margo/Thallium 97

3.2.3. Mochi: RPC Execution Model 98

3.2.3.1. Service Discovery 98

3.2.3.2. Request Generation 99

3.2.3.3. Execution of the Request 99

3.2.3.4. Issuance of a Response 101

3.2.3.5. Receipt of a Response 101

3.2.4. Mochi: Microservices 101

3.2.5. Mochi: Composed Services 102

3.3. The Goal of Performance Observability 103

3.4. Elements of Observability 107

3.4.1. Query 1: Identifying Dominant Microservice Operations . . . 107

3.4.2. Query 2: Detecting Load Imbalance 108

3.4.3. Query 3: Eliciting Individual Request Structure 108

3.4.4. Query 4: Mapping RPC Resource Usage and Time 108

3.4.5. Query 5: Detecting Hardware and Software Resource Saturation 109

3.4.6. Query 6: Application-level API Performance 109

3.4.7. Query 7: Identifying Better Service Configurations 110

3.5. Summary 110

IV. SELECTING AND COMBINING SOURCES OF PERFORMANCE DATA 111

4.1. Introduction 111

4.2. Related Work 112

4.2.1. HPC Performance Tools 112

4.2.2. Cloud-Based Tools for Microservices 113

4.2.3. Tools That Integrate Data Sources 114

4.3. SYMBIOSYS: Distributed Callpath Profiling 114

4.3.1. Instrumentation 114

4.3.2. Analysis and Visualization 116

4.4. SYMBIOSYS: Distributed Request Tracing 118

4.4.1. Instrumentation 118

4.4.2. Analysis and Visualization 118

4.5. SYMBIOSYS: Performance Data Exchange 120

4.5.1. Performance Variables 121

4.5.2. Performance Tool Interface 122

4.6. SYMBIOSYS: Orienting Performance Data to Generate Observability 123

4.7. SYMBIOSYS: Sampling Node Resource Usage 125

4.8. SYMBIOMON: Time-series Metrics 125

4.8.1. COLLECTOR Microservice 126

4.8.2. Data Model and Metric API 127

4.8.2.1. Metric Creation 127

4.8.2.2. Metric Update 128

4.8.2.3. Management of Metric Buffer Size 128

4.9. Case Studies Addressing the Performance Queries 129

4.9.1. Mobject Composed Service 129

4.9.1.1. Query 1: Identifying Dominant Microservice Dependencies 129

4.9.1.2. Query 2: Detecting Load Imbalance 131

4.9.1.3. Query 3: Discovering Individual Request Structure . 132

4.9.2. SONATA Microservice 133

4.9.2.1. Query 4: Mapping RPC Resource Usage and Time 133

4.9.2.2. HEPnOS Composed Service 134

4.9.3. Query 5, 7: Observing Resource Saturation and Identifying a Better Service Configuration 136

4.9.3.1. Too Few Execution Streams 136

4.9.3.2. Too Many Databases 137

4.9.3.3. Effect of a Low Batch Size on Client Progress . . . 138

4.9.3.4. Query 6, 7: Improving Key-value Store Performance Under Concurrency 141

4.9.3.5. Establishing a Performance Baseline 143

4.9.3.6. Identifying Performance Bottlenecks 144

4.9.3.7. A Better Service Configuration 144

4.10. Overhead Analysis 148

4.10.1. Setup 148

4.10.1.1. Hardware 148

4.10.1.2. Software 148

4.10.2. SYMBIOSYS Overhead Study 148

4.10.3. SYMBIOMON Overhead Study 150

4.11. Limitations 151

4.12. Summary 152

V. UBIQUITOUS MONITORING OF SERVICES AND IN SITU WORKFLOW COMPONENTS 154

5.1. Introduction 155

5.2. SYMBIOMON: A Composable Service for Monitoring and Analysis . 156

5.2.1. Related Work 157

5.2.1.1. Cloud-Based Service Monitoring Tools 158

5.2.1.2. HPC Monitoring and Analysis Tools 158

5.2.2. Design 159

5.2.2.1. AGGREGATOR 160

5.2.2.2. REDUCER 160

5.2.2.3. Py-COLLECTOR: Python Client for Remote Monitoring 161

5.2.2.4. Flexible Integration 161

5.2.3. Implementation 162

5.2.3.1. AGGREGATOR 163

5.2.3.2. REDUCER 163

5.2.3.3. BEDROCK Integration 164

5.2.3.4. Service Discovery and Composition 164

5.2.3.5. Metric Reduction 164

Chapter Page

5.3. Plugin Architecture for Ubiquitous Monitoring 165

5.3.1. Related Work 167

5.3.2. Background 170

5.3.2.1. TAU Overview 170

5.3.2.2. TAU Operation 173

5.3.2.3. Constraints 176

5.3.3. TAU Plugin Architecture 177

5.3.3.1. TAU States and STATE Plugins 178

5.3.3.2. TRIGGER Plugins 180

5.3.3.3. AGENT Plugins 183

5.3.4. Plugin Implementation 183

5.3.4.1. Plugin Lifecycle 184

5.3.4.2. Enabling Customizability and Runtime Control . . . 185

5.3.5. Usage Scenarios 185

5.3.5.1. Event Counter 185

5.3.5.2. Selective Tracing 187

5.3.5.3. Filter Plugin 188

5.3.5.4. TAU SOS Plugin 189

5.3.5.5. Trigger: Aggregating Interval Events 190

5.3.5.6. Agent: Asynchronous Load Tracking 191

5.4. Monitoring of HPC Applications and Ensembles 191

5.4.1. TAU Plugin for SYMBIOMON 191

5.4.2. HPC Application Monitoring: Visualizing LULESH Load Imbalance 193

5.4.3. HPC Ensemble Monitoring: Performance Variation in GROMACS Ensembles 194

Chapter Page

5.4.3.1. Performance Variability of Ensemble Tasks 196

5.4.3.2. Ensemble Monitoring Solution 197

5.5. Overhead Analysis 200

5.5.1. TAU Plugin Operation 200

5.5.1.1. Hardware Setup and Software Background 201

5.5.1.2. Results 202

5.5.2. SYMBIOMON Monitoring and Analysis 204

5.5.2.1. Hardware and Software Setup 204

5.5.2.2. Results 205

5.5.2.3. Aggregator Sensitivity Analysis 205

5.6. Limitations 207

5.7. Summary 208

VI. EXTENDING THE APPLICABILITY OF HIGH PERFORMANCE MICROSERVICES 209

6.1. Introduction 209

6.2. Related Work 213

6.2.1. In situ Visualization 213

6.2.2. Services in HPC 215

6.3. Ascent Visualization Library 216

6.4. SERVIZ: A Shared Visualization Service 217

6.4.1. Original In transit Cost Model 217

6.4.1.1. Terminology and Base Cost Model 218

6.4.1.2. Shared-service Cost Model 219

6.4.2. Service Architecture and Implementation 220

6.4.2.1. SERVIZ API 220

6.4.2.2. SERVIZ Implementation 222

Chapter Page

6.4.2.3. SERVIZ Execution Model 224

6.4.2.4. SERVIZ Deployment Configurations 226

6.4.2.5. SERVIZ Operating Modes 226

6.5. Evaluation 228

6.5.1. Setup 228

6.5.1.1. Hardware 228

6.5.1.2. Software 229

6.5.2. Single Client Experiments 229

6.5.3. Shared-Server Experiments 233

6.5.3.1. Immediate Mode 234

6.5.3.2. Delayed Mode 237

6.6. Discussion 240

6.7. Summary 242

VII.CONCLUSION AND FUTURE WORK 243

7.1. Conclusion 243

7.2. Future Work 245

7.2.1. Enabling Online Service Adaptivity 245

7.2.2. Elastic In Situ Visualization 246

7.2.3. User-level Shared Monitoring + Learning Service 246

REFERENCES CITED 248

LIST OF FIGURES

Figure Page

1. Challenges and research questions addressed in different chapters. . . . 6

2. CCA: Framework Types (inspired from DCA [1]) 27

3. Modularization of MVAPICH2 (image credits: Dr. D.K. Panda, Network Based Computing Lab, The Ohio State University) . . 39

4. Overlap Between In-Situ Analysis and Visualization (ISAV) Tools, Data Management Libraries, and Data Services 47

5. Monolith vs Microservices 91

6. Microservices: Interactions Resulting From Requests Through the Service 94

7. Mochi: Interaction between Distributed Components and Software Stack 99

8. Mochi RPC Execution Model 100

9. Mobject: An Illustration (Image Credits: Dr. Matthieu Dorier, Argonne National Laboratory) 103

10. Functionally Equivalent Service Configurations 106

11. Annotated Mochi RPC Execution Model 116

12. Mobject + ior: Dominant Callpaths by Call Time 117

13. Mobject + ior: Trace Analysis for the mobject_write_op RPC 120

14. Mochi: Opaqueness of RPC Events to Callpath Profiling and Tracing . . 121

15. PVAR Interface Between Margo and Mercury 124

16. Mobject Illustration 130

17. ior + Mobject: Identifying the Dominant Callpaths 131

18. ior + Mobject: Raw Cumulative Distribution of Calltimes Among Origin Entities 132

Figure Page

19. ior + Mobject: OpenZipkin [2] Trace Visualization Depicting Discrete Steps for a Single mobject_write_op Request 133

20. Sonata: Mapping Execution Time to Individual Steps 134

21. HEPnOS Illustration 135

22. HEPnOS: Cumulative Target RPC Execution Time for sdskv_put_packed 137

23. HEPnOS: Sampling Blocked Tasks from Argobots for sdskv_put_packed . 139

24. HEPnOS: Unaccounted Component of RPC Execution 140

25. HEPnOS: Sampling OFI Events Read from Network Abstraction Layer for sdskv_put_packed 141

26. SDSKV: “num_entrants” Metric and Concurrency Regions 143

27. HEPnOS: Iteratively Improving Data Service Performance Under Concurrency 146

28. HEPnOS: SYMBIOSYS Measurement Overheads 149

29. Selecting and Combining Instrumentation Techniques 153

30. SYMBIOMON Conceptual Illustration 161

31. SYMBIOMON: Flexibility in Integrating Monitoring and Analysis Capabilities 162

32. TAU supports *interval* and *atomic* events. Event measurements are made for each event occurring on each thread of execution. The performance data is store in a thread-specific data structure. 174

33. TAU captures performance data for all threads in each application process. 175

34. TAU collects parallel profiles for different events across all threads and processes in an application’s execution. The “global” parallel profiles are distributed across all nodes of the application. 175

35. (Top) STATE plugin operation for interval and atomic events. (Bottom) TRIGGER plugin operation. 181

36. Paraprof: LULESH profile 186

Figure Page

37. Vampir: LULESH trace 188

38. TAU SYMBIOMON Plugin 193

39. LULESH: Monitoring the Range of MPI_Allreduce Exclusive Time . . . 194

40. Traditional HPC Applications vs. HPC Ensembles 195

41. Adaptive Ensembles 196

42. Performance Variation in GROMACS Ensembles 197

43. Performance Monitoring of GROMACS Ensembles 198

44. Online Monitoring of the Degree of Ensemble Task Runtime Variation . 200

45. LULESH: Overhead Study 206

46. Broad Applicability of Mochi Microservices 210

47. Different approaches to in situ visualization. The red arrow indicates data transfer in (b) and (c). 211

48. SERVIZ: Division Into Multiple Instances 224

49. SERVIZ: Deployment Configurations 227

50. Single Client In transit Visualization Experiments. X-axis Represents the Simulation x Server (MxN) Process Counts. 230

51. Server Idle Time in Different Modes (Single Client) 231

52. Pending RPC Queue Size: Immediate Mode (120x15) 232

53. VCEF and In transit Cost Savings (Single Client) 232

54. Multiple Simulations/Server: Immediate Mode (120x15) 235

55. Pending RPC Queue Size: Immediate Mode (Shared) 235

56. Simulation Execution Times: Immediate Mode (Shared) 236

57. Cost Savings and Idle Time: Immediate Mode (Shared) 238

58. Simulation Time and Cost Savings: Delayed Mode (Shared) 238

59. Memory Usage: Delayed Mode (Shared) 238

LIST OF TABLES

Table Page

1. Comparing Component Frameworks 35

2. Comparing Scientific Frameworks and CCA 36

3. Comparing Techniques for Performance Data Exchange 43

4. Distributed HPC Frameworks: Composition Models 52

5. Distributed HPC Frameworks: Coupling Strategies and Communication Protocols 57

6. Distributed HPC Frameworks: Resource Allocation Scheme 64

7. Distributed HPC Frameworks: Resource Elasticity 68

8. MxN Coupling Frameworks 69

9. Data Staging and I/O Frameworks 72

10. Performance Tools for Coupled Applications and Workflows 79

11. Improving Service Performance By Addressing a Set of Performance-related Queries 107

12. Performance Variable Classes 122

13. List of Available Performance Variables 123

14. Combining Instrumentation Strategies 125

15. SYMBIOMON Metric API 127

16. SYMBIOMON Metric Types 128

17. SYMBIOMON Metric Reduction Operators 128

18. HEPnOS: Service Configurations 136

19. HEPnOS: Generating Better Service Configurations 147

20. HEPnOS: SYMBIOSYS Analysis Overheads 150

Table Page

21. HEPnOS: SYMBIOMON Measurement Overheads 150

22. LULESH: Processing state counter values on thread 0, MPI rank 0 (other ranks also captured) 187

23. LULESH: Overheads with multiple plugins 203

24. LULESH: Trace size with selective tracing 203

25. XSBench: Overheads with multiple plugins 204

26. Aggregator Sensitivity Analysis 206

27. Cost Model: Definition of Terms 221

28. SERVIZ API Description 222

29. AMR-Wind Inline Visualization Time 229

30. Shared-Service Configurations (**M**ode (immediate, delayed), Visualization **F**requency) 237

31. SERVIZ: Lines of Code 240

CHAPTER I

INTRODUCTION

Traditionally, high performance computing (HPC) has been the domain of bulk-synchronous, parallel MPI applications deployed as single program, multiple data (SPMD) executables. The past decade has seen a steady and marked rise in the number of separate, parallel applications and distributed services operating together to achieve a common scientific goal [3]. This new class of coupled applications and services, broadly referred to as *in situ workflows*, involves the *collective execution* of distributed "entities" comprising MPI applications, analysis tasks, visualization modules, and distributed services providing a host of specialized functionality. In situ workflows present new challenges concerning their performance observation, monitoring, and optimization. This dissertation focuses on in situ workflows involving high performance, distributed services.

The definition of what constitutes a "service" is indeed broad, originating in the broader cloud computing community as "how needs and capabilities are brought together" [4]. While that abstract definition can support several different implementations of services, historically, the implied definition for what constitutes a service involves (at the very least) the following characteristics: (1) the service module runs inside a dedicated (set of) process(es) separate from the client invoking the service, (2) the service module is a distinct entity that exposes a well-defined application programming interface (API) to access its functionality, usually involving a network call. Service-oriented architectures have enjoyed great success in the cloud community due to the need and support for extensible software modules, rapid development, integration of new and varied functionality, and the need to scale individual software components and development teams independently.

While the concept of services *per se* is not new to the HPC software community, until the last decade or so, their scope has primarily been limited to providing an interface to access core system functionality such as parallel file storage, job launch, and scheduling mechanisms. Put another way, the end-user of the HPC platform, such as a domain scientist or application programmer, has rarely had to worry about programming or configuring these services. That job was usually reserved for the system administrators and developers of lower-level communication libraries such as MPI. However, the advent of user-level service frameworks such as ADIOS [5], Mochi [6], Faodel [7], DataSpaces [8] and Decaf [9] is beginning to change this rhetoric. A common observation that can be made about most, if not all of these emerging service frameworks is that they each provide access to either transient data storage, data analysis, data visualization, or some form of "in-memory" computing. This trend can arguably be attributed to two significant factors. On the one hand, the slow growth of parallel I/O bandwidth on HPC clusters in relation to the computational capabilities has forced applications to minimize their access to parallel file storage. The rise of in situ schemes for data analysis and visualization and asynchronous I/O techniques that hide the cost of expensive parallel file system operations stand as a testament to this fact. On the other hand, the range of applications requiring high performance capabilities has broadened rapidly due to the advent of data-oriented statistical computing techniques such as machine learning (ML). As a result, HPC data frameworks have evolved to support a broader range of data models for storage and analysis, resembling their cloud counterparts in form while being required to take advantage of the HPC platform on which they operate.

The explicit coupling of user-level data storage, visualization, and analysis services alongside traditional MPI-based applications puts the onus on the end-user to identify the "optimal" service configuration and make decisions concerning the allocation of computing resources to different in situ workflow components. A haphazard allocation of resources to different workflow components can yield an overall poor performance for the workflow. The first step in helping the user identify a poorly performing service (workflow) configuration is to enable the service operation's performance observability (insight). The ultimate goal is to use this insight to develop a working performance model of the workflow and design appropriate solutions for service adaptivity. The main focus of this dissertation is (1) to identify how to enable performance observability of services that are built by composing high performance microservices, and (2) how to enable the performance monitoring of the services along with the other in situ workflow components. Enabling performance observability and monitoring with HPC microservices involves several challenges:

Challenge 1: Identifying What To Observe and Monitor

The first step in enabling performance observability is to identify the *appropriate* set of performance data to observe. Enabling observability is a challenge for two reasons. First, the performance observations must accurately represent the key aspects affecting service performance, implying that a good understanding of the service execution model is needed. If this execution model is inaccurate, performance observations may prove inadequate in helping understand service performance. Further, the performance observations must be able to give us a *holistic* picture of all aspects of service performance, such as identifying the high-level service interactions and providing a resource-centric view of the execution. Second, the performance observations must be able to guide the user (or an external entity) in understanding how to modify the service configuration to result in better service performance.

Challenge 2: Selecting Instrumentation Techniques

The services that this dissertation considers are built and composed using high performance remote procedure calls (RPC). In other words, the flow of control resulting from an execution of a service request can traverse several process boundaries before a response is returned to the caller. Traditional HPC applications are built and deployed as MPI applications, and therefore, most HPC performance tools that are designed to operate within an MPI context implicitly assume that control is not passed *between* processes and applications. Given that HPC performance tools are primarily inapplicable in this context, this dissertation finds the need to look to the broader cloud computing community for inspiration. At the same time, the challenge is to understand how to adapt existing cloud-based observability and monitoring techniques to be applicable and valuable in an HPC execution environment.

Challenge 3: Combining and Orienting Performance Data

The third challenge lies in understanding how to combine and orient the captured performance data to yield helpful insight and actionable knowledge. This challenge has three components to it. First, knowledge of the desired visualization and analysis is required before performance data can be combined. Further, the decision of *what* to visualize and analyze must pertain directly to the goals of performance observability. Second, combining and orienting the performance data from different sources requires an understanding of *how* to implement a "hand-shake" operation between the different software layers that hold the said data. Third, the decision about *when* to collect the performance data affects both the quality of the resulting observations and the runtime overheads involved.

Challenge 4: Managing Large Data Volumes

The fourth challenge pertains primarily to enabling online monitoring and analysis of in situ workflows that involve HPC microservices. The HPC services that this dissertation studies operate in a highly concurrent environment, potentially serving multiple MPI clients simultaneously. The quantity of captured performance data or traces captured from different distributed components could quickly become intractable. Therefore, it is necessary to manage the large trace data volume to reduce the storage requirements and speed up the extraction of performance insight from the data.

Challenge 5: Ubiquitous Monitoring

The final challenge also pertains to enabling online monitoring and analysis of in situ workflows. Given that HPC services operate alongside other workflow components such as MPI applications, analysis tasks, and ML ensembles, any proposed monitoring solution must be ubiquitously applicable to all workflow components, requiring a minimal amount of additional setup and glue code to ensure easy operation. Further, it is a desirable property of such a monitoring solution to be seamlessly integrated while leveraging existing solutions to gather performance data for monitoring.

1.1 Main Research Question

This dissertation strives to investigate these challenges and answer the following research question: **How to enable and use performance insight to improve service configuration when the service is a part of a coupled, HPC in situ workflow?** This broad research question is decomposed into more minor, constituent questions and addressed in subsequent chapters of this dissertation. Figure 1 depicts the specific questions addressed in each chapter. Brief descriptions of each chapter and connections are presented here.

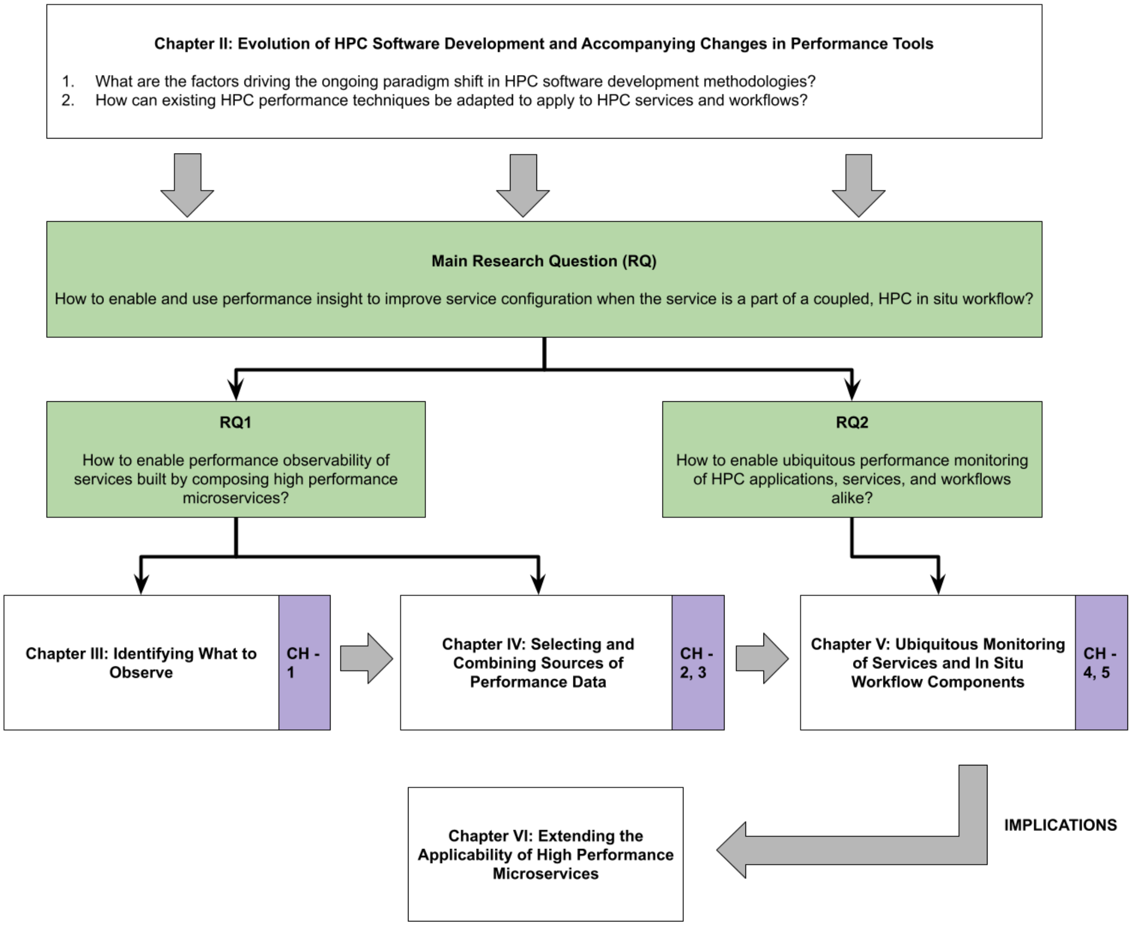

Chapter II: Evolution of HPC Software Development and Accompanying Changes in Performance Tools
1. What are the factors driving the ongoing paradigm shift in HPC software development methodologies?
2. How can existing HPC performance techniques be adapted to apply to HPC services and workflows?
Main Research Question (RQ)
How to enable and use performance insight to improve service configuration when the service is a part of a coupled, HPC in situ workflow?
RQ1
How to enable performance observability of services built by composing high performance microservices?
RQ2
How to enable ubiquitous performance monitoring of HPC applications, services, and workflows alike?
Chapter III: Identifying What to Observe
CH - 1
Chapter IV: Selecting and Combining Sources of Performance Data
CH - 2, 3
Chapter V: Ubiquitous Monitoring of Services and In Situ Workflow Components
CH - 4, 5
IMPLICATIONS
Chapter VI: Extending the Applicability of High Performance Microservices

Figure 1. Challenges and research questions addressed in different chapters. Research questions are presented in green boxes. The challenges are presented in violet boxes.
Challenge 1 - *Identifying What To Observe and Monitor*,
Challenge 2 - *Selecting Instrumentation Techniques*,
Challenge 3 - *Combining and Orienting Performance Data*,
Challenge 4 - *Managing Large Data Volumes*,
Challenge 5 - *Ubiquitous Monitoring*.

1.2 Chapter II — Evolution of HPC Software Development and Accompanying Changes in Performance Tools

Designing maintainable and scalable modular software has always been a long-standing goal within the HPC community. Tracking the evolution of HPC software development over the past 30 years yields several insights into the origins of dominant software development paradigms. These insights put the current trends in context and allow for a systematic study of the performance-related challenges exposed by increasingly modular software. The late 1990s saw the emergence of a community-wide effort to develop and deploy modular software through the common component architecture (CCA) [10, 11, 12]. Microservices arguably represents an end of the spectrum of distributed component software, allowing for rapid development and a high degree of code reuse between projects. This dissertation follows the parallel evolution of HPC performance tools for modular software and, in the process, identifies critical techniques that can be adapted to generate performance observability of HPC microservices.

The material in this chapter is unpublished with no co-authorship. However, revision suggestions were given by the dissertation advisory committee (Allen Malony, Hank Childs, and Boyana Norris) during the Area Exam.

1.3 Chapter III — Identifying What to Observe and Monitor

Chapter III partly addresses **RQ1**: How to enable performance observability (insight) of services built by composing high performance microservices? Chapter III provides an introduction to microservices and motivates the performance challenges associated with such highly modular software. While these techniques are broadly applicable to any high performance microservice environment, this dissertation uses the Mochi [6] software stack as a development vehicle. Beginning with a broad description of microservices and an overview of what makes them attractive to the development of distributed software, Chapter III describes the critical components of the Mochi [6] high performance service framework in detail. In particular, this chapter pays special attention to elicit the key aspects of Mochi's RPC execution model to identify *what* aspects of performance to observe and monitor.

The content of this chapter is published at IPDPS, 2021 [13]. This publication is co-authored by Dr. Allen Malony from the University of Oregon and Dr. Philip Carns, Dr. Robert Ross, Dr. Matthieu Dorier, and Shane Snyder from Argonne National Laboratory, and Dr. Jerome Soumagne from The HDF Group.

Connection to Chapter IV: The broad question **RQ1** can be addressed in three parts: (1) identifying what to observe and monitor, (2) selecting the right set of performance instrumentation techniques, and (3) combining the resulting performance data to yield performance insight. While Chapter III provides an answer to part (1), Chapter IV provides an answer to part (2) and part (3), describing in detail a set of techniques to gather and orient the performance observation data.

1.4 Chapter IV — Selecting and Combining Sources of Performance Data

Chapter IV partly addresses **RQ1**: How to enable performance observability of services built by composing high performance microservices? Chapter IV introduces SYMBIOSYS, a methodology for integrated performance observation and analysis of HPC microservices. The SYMBIOSYS approach embellishes distributed callpath profiles and traces with rich performance data from lower levels in the Mochi microservice software stack. Distributed callpath profiling and tracing techniques are inspired by cloud-based tools such as Dapper [14] at Google, and the embellishment of callpath profiles occurs through a performance data exchange strategy that is inspired by the MPI Tools Information Interface [15]. While distributed callpath profiling can provide an overview of the dominant microservice interactions, SYMBIOSYS demonstrates that embellishing them with rich performance data from different software layers provides insight into how these microservice interactions affect time and resource usage. SYMBIOSYS has a low operating overhead at scale, resulting in no more than 4% of runtime overheads when applied to a workflow involving the HEPnOS [6] data service. Concerning the performance analysis of the microservice API under concurrent access, this dissertation finds time-series metrics (event traces) to be helpful. Further, combining time-series metrics with statistical data models generated offline can prove helpful in identifying more optimal service configurations.

The content of this chapter is published at IPDPS 2021 [13]. This publication is co-authored by Dr. Allen Malony from the University of Oregon and Dr. Philip Carns, Dr. Robert Ross, Dr. Matthieu Dorier, Shane Snyder from Argonne National Laboratory, and Dr. Jerome Soumagne from The HDF Group. The content of this chapter is also published at HiPC 20211 [16, 17] and SC 2021 [18]. These publications are co-authored by Dr. Kevin Huck and Dr. Allen Malony from the University of Oregon, and Dr. Philip Carns, Dr. Robert Ross, and Dr. Matthieu Dorier from Argonne National Laboratory.

Connection to RQ2 and Chapter V: Chapter IV addresses part (2) and part (3) of **RQ1**. The consolidated RPC callpath profiles that SYMBIOSYS generates enable insight into the operation of the HPC service components. No existing HPC performance tool could be applied for this purpose. Using SYMBIOSYS, we could better identify the root causes for poorly performing service configurations and rectify these issues, resulting in better performing service configurations. However, SYMBIOSYS is limited to offline analysis of performance data. Given that HPC services operate as a part of in situ workflows, monitoring aspects of service performance online in conjunction with the other workflow components can help identify poorly performing service configurations during execution and help guide the workflow to a better performing state.

1.5 Chapter V — Ubiquitous Monitoring of Services and In Situ Workflow Components

Chapter V addresses **RQ2**: How to enable ubiquitous performance monitoring of HPC applications, services, and workflows alike? Observability tools for distributed cloud services often include metric data collection to complement performance data gathered through distributed request tracing. Given that services operate *as a part* of the workflow, it might be necessary to monitor and evaluate service performance in the context of the performance of other workflow components, such as MPI applications. Therefore, a ubiquitously applicable performance monitoring infrastructure that can enable the remote, online monitoring of services and other in situ workflow components is required. This chapter introduces SYMBIOMON, a metric monitoring service tailored for HPC platforms. The design of this service draws inspiration from cloud-based monitoring tools such as Prometheus [19] to allow the definition and export of metrics with custom *taglists*. SYMBIOMON is specifically designed to operate efficiently in sub-millisecond time intervals, enabling the management of large volumes of metric trace data through the AGGREGATOR and REDUCER microservices for online metric data reduction and analysis. Notably, every SYMBIOMON component is implemented as a Mochi microservice, allowing flexibility in composing and toggling different functionality while simultaneously leveraging Mochi's high performance RPC stack to operate with reasonable overhead. This chapter also demonstrates the seamless integration of SYMBIOMON's monitoring and analysis capabilities for traditional MPI-based applications (LULESH) and HPC ensembles by leveraging the previously developed TAU plugin system [20].

The content of this chapter is published at HiPC 2021 [16], HIPC 2021 (Poster) [17], and SC 2021 (Poster) [18]. These publications are co-authored by Dr. Kevin Huck and Dr. Allen Malony from the University of Oregon, and Dr. Philip Carns, Dr. Robert Ross, and Dr. Matthieu Dorier from Argonne National Laboratory. The content of this chapter is also published at ICPP 2019 [21]. This publication was co-authored by Dr. Allen Malony, Dr. Kevin Huck, Dr. Sameer Shende and Dr. Nick Chaimov from the University of Oregon. The content of this chapter also represents unpublished work resulting from a collaboration with Dr. Allen Malony from the University of Oregon, Dr. Matteo Turilli from RUTGERS, and Dr. Shantenu Jha, Dr. Tan Li, and Dr. Mikhail Titov from Brookhaven National Laboratory.

Connection to Chapter VI: Chapter V addresses **RQ2**. In connection to Chapter VI, the key takeaway from Chapter V is applying the Mochi RPC framework to develop a high performance metric monitoring service. Note that the Mochi framework was originally designed to support the development of composable, high performance data services. Chapter VI demonstrates that the same core Mochi components can be broadly applicable to *rapidly* integrate microservices for in situ visualization of MPI applications.

1.6 Chapter VI — Extending the Applicability of High Performance Microservices

Chapter VI presents the implications of the key findings from attempting to answer **RQ2**. In particular, this chapter explores the broader application of microservices for in situ visualization. This chapter presents SERVIZ, a high performance in situ visualization service built and deployed as a hybrid MPI + RPC Mochi microservice.

SERVIZ operates in a shared setting, allowing multiple MPI simulations (clients) to access its visualization API simultaneously to result in significant cost savings over prior approaches. The results from SERVIZ suggest that building shared HPC services out of microservice components provides three comprehensive benefits: (1) cost savings, (2) rapid development, and (3) high degree of code reuse, promoting maintainability of HPC software. The content of this chapter is currently under review at SC 2022 [22]. This publication effort is co-authored by Dr. Allen Malony and Dr. Hank Childs from the University of Oregon.

With Chapter VII, this dissertation concludes its exploration to answer the main question presented in Section 1.1 concerning the generation of optimal service configurations and the implications of the resulting tool solution. This dissertation includes prose, figures, and tables from previously published conferences, workshops, and journal proceedings.

CHAPTER II

EVOLUTION OF HPC SOFTWARE DEVELOPMENT AND ACCOMPANYING CHANGES IN PERFORMANCE TOOLS

This chapter contains unpublished material with co-authorship. The content presented in this chapter was developed as a part of the departmental Area Exam, where I received guidance from my advisor Dr. Allen Malony. While working on the Area Exam, I received feedback and suggestions from my dissertation advisory committee members (Dr. Hank Childs and Dr. Boyana Norris). I did all the data collection and writing while the committee members helped proofread the Area Exam document.

2.1 Introduction

Chapter II presents an overview of the evolution of the HPC software development methodologies over the past 30 years. Special attention is devoted to describe the key factors underlying these changes while also presenting a categorization of the state-of-the-art HPC software frameworks under various axes of analysis. At the same time, the accompanying changes in HPC performance tool development are discussed to elicit the open areas requiring tool solutions to be put in place.

Over the past three decades, there has been a constant evolution in how HPC distributed software is conceptualized, implemented, and deployed. Traditional HPC software development has been centered around the message-passing programming model. In particular, the message-passing interface (MPI) has been the de-facto programming model of choice for developing distributed HPC applications. In response to the recent explosion of data-centric and machine learning (ML) workloads in scientific computing [23], HPC systems and software are rapidly evolving to meet the demands of diversified applications. These new applications do not fit into the MPI programming model [23, 24], thus necessitating a change in the fundamental methodologies for distributed HPC software development. In particular, the emergence of coupled applications, ensembles, and in-situ software services running alongside traditional HPC simulations [25] are the key indicators of such change. Scientific workflows are beginning to move away from traditional MPI monoliths to resemble a mix of several different pieces of specialized distributed software working in concert to achieve some larger goal [3].

Within a process running inside the broader distributed application, increasing software complexity and the need to perform ever-more-realistic simulations have been the driving forces behind the componentization of HPC software [26]. Parallels can be drawn between adopting componentization in the industry [27] and the subsequent push to componentize HPC software to manage complexity. At the same time, HPC performance tools have also been updated to reflect this change [28]. Over the last 20 years, the push to componentize software has resulted in evolving a service-oriented architecture in the industry. The HPC community has recently been actively looking into similar software architectures to support heterogeneous, data-centric workloads.

The key aspect that sets HPC applications from other forms of distributed software is the need to achieve high performance, high efficiency, and a high degree of scalability on exotic HPC hardware. In such environments, performance measurement and analysis tools play a critical role in identifying sources of performance inefficiencies. State-of-the-art HPC performance tools such as TAU, HPCToolkit, and Caliper [29, 30, 31] excel at the performance analysis of monolithic MPI applications. However, when faced with the task of holistically analyzing the performance of coupled multi-physics codes or distributed HPC data services, applying these performance tools without change finds limited application because these tools implicitly rely on the existence of an MPI library to bootstrap their measurement frameworks. Studying the evolution of HPC performance tools in this context is necessary to identify opportunities and future tool design requirements.

This area exam explores the evolution of HPC software and performance tool development. Starting with a collection of source files built into one monolithic MPI executable, HPC software has evolved to support coupled applications, distributed data services, in-situ ML, visualization, and analysis modules running together on a single machine allocation. A novel narrative of the tension between the need to manage software complexity while simultaneously achieving high performance is presented. Wherever appropriate, notable trends from the general computing industry are cited as key technology enablers of such change. The parallel timeline and evolution of performance measurement, analysis, and online monitoring tools and techniques are also presented in this research document.

2.2 Background

To familiarize the reader with standard HPC programming practices and common terminology, a brief overview of the state-of-the-art in HPC system architectures, applications, and performance analysis software is necessary. HPC machines, also known as *supercomputers*, represent the largest networked computers designated for scientific computing. Although official figures of the cost of such machines are rarely released, speculations [32] suggest that the hardware cost alone is several hundreds of millions of dollars. Also, the annual operating costs of running these machines are in the order of tens of millions of dollars. Thus, the applications that run on these machines must do so at the highest efficiency possible to maximize scientific output, maximize machine occupancy, and minimize cost. Today's typical HPC machine architecture consists of a heterogeneous mix of general-purpose CPUs and accelerator architectures such as the graphics processing unit (GPU) [33]. These computing elements are connected through a high-bandwidth, low-latency interconnect such as Infiniband [34]. Further, all the computing and networking elements are typically situated within the same IT infrastructure or building. These key characteristics separate HPC architectures from more general distributed grid computing architectures.

2.2.1 MPI: The Dominant Distributed Programming Model.

The MPI programming model [35] has dominated the HPC software development landscape for a large portion of the past three decades. An MPI application is launched as a set of N communicating processes. Traditional MPI applications [36, 37] divide an application domain, such as a computational grid into several logical sub-domains. Each MPI process is assigned one or more sub-domains on which they perform local computation. When necessary, these MPI processes communicate to exchange or aggregate intermediate results. This communication can either be point-to-point or collective. A typical scientific application [36] contains a discretized time domain and a computational grid (spatial domain) and runs for a certain number of fixed timesteps. Communication and computation proceed in phases within a timestep, with periodic synchronization between the different processes. Such a model of parallel computation is referred to as the bulk-synchronous parallel model (BSP).

Given the importance of MPI, there have been several large-scale, ongoing efforts to implement high-performance MPI implementations that are portable as well [38, 39]. Communication requires processes to synchronize with each other. Besides, communication over the network can significantly slow down a parallel program. At a large scale, synchronous, collective communication can degrade the application's overall performance and quickly limit the application's scalability.

Therefore, several HPC performance engineering efforts have been centered around improving MPI library communication performance.

2.2.2 Shared-Memory Programming Models. Multi-core and many-core CPU architectures such as the Intel Xeon Phi [40] and accelerator architectures such as the GPU have become commonplace on leadership-class HPC systems[33]. HPC applications have evolved to support and extract performance from the increased on-node parallelism. Specifically, shared-memory parallel programming has been a key focus area for performance optimizations. Notable programming models offering shared-memory parallel programming capabilities include OpenMP [41], TBB [42], pthreads [43], and OpenACC [44]. NVIDIA CUDA [45] is arguably the most popular GPU programming model, followed by OpenCL [46].

These shared-memory programming models expose their functionality either through a library-based API or through compiler *pragmas* or hints to aid with the automatic identification and generation of parallel code. Invariably, shared-memory parallel programming involves the generation of *parallel threads* of execution. These threads share the same process address space, may have their local stacks, and communicate through shared memory regions.

2.2.3 Other Programming Models. While many conventional HPC applications employ a distributed model such as MPI combined with a shared-memory model such as OpenMP, other applications employ hybrid programming models and runtimes. Notable examples include Charm++ [47], a machine-agnostic task-based programming approach, and partitioned global address space (PGAS) programming models such as UPC [48] and Chapel [49].

2.2.4 Performance Analysis Tools. This section discusses the state-of-art in the performance analysis of traditional HPC applications.

2.2.4.1 MPI Performance Analysis. Performance tools for HPC applications have primarily catered to those applications that employ the MPI programming model. Typically, the parallel profiling and tracing tools build on the presence of an MPI library to bootstrap their measurement frameworks [29, 31, 30]. The PMPI-based library interposition technique has successfully enabled performance tools to intercept MPI calls to perform timing measurements and capture other relevant performance data such as message sizes. The PMPI approach is often the first step in analyzing the performance of MPI applications. Key performance metrics include the sizes of MPI messages, contributions of MPI collective routines, and the contributions of MPI synchronization operations to the overall execution time.

2.2.4.2 Shared-Memory Performance Analysis. Regarding the capture of application-level performance information (function-level timers), HPC performance tools follow one of two schools of thought. Instrumentation-based tools [29, 31] rely on intrusive instrumentation to elicit the exact measurements of events. Tools based on statistical sampling [29, 30] rely on lightweight sampling and call stack unwinding to capture statistical features of the performance data. Hardware counters are commonly used to track the efficiency of various routines based on their hardware resource usage characteristics. The performance API (PAPI) [50] has grown into a standard and portable way of exposing hardware performance data.

Shared-memory programming libraries expose their profiling APIs to allow insight into their operation and performance. Notably, the OMPT interface [51] allows performance tools to register callbacks for several events defined by the OpenMP specification. Likewise, the CUDA CUPTI API allows insight into the operation of the CUDA API. After the node-level performance data from various sources is collected, performance tools typically orient and aggregate this data around the MPI processes involved in the particular execution instance.

2.3 Definitions

This section defines the various terms that are used in the sections that follow. Unless specified otherwise, any usage of these terms pertains to the following definitions.

2.3.1 Module. A *module* is *any* piece of software or code entity with well-defined boundaries used as a general building block for higher-level functionality. A module can be a library, a class object, a file, a service, or a component. Throughout this document, the term "module" is used in the broadest context possible, i.e., it does not refer to any specific software, implementation, technique, or specification. Therefore, it follows that modularization is the process by which a piece of software is divided into separate, independent entities by following the general software design principle of "separation of concerns". *Process-local modularization* results in software modules that run within the same address space (process). *Distributed modularization* results in software modules that are separated by different address spaces (processes).

2.3.2 Component. The term *component* is interchangeably used with the term module everywhere *except* in Section 2.5. In Section 2.5, the term "component" has a special meaning and refers to modules that adhere to a particular type of component architecture and interface specification. Componentization, as used in Section 2.5, is converting a piece of software into components that follow the component architecture.

2.3.3 Service. *Services* are regarded as loosely connected software modules running on *separate* processes with well-defined public interfaces specifying access to the service functionality. Usually (but not always), the interaction between two service entities involves calls over the network. Again, every service is a module (or a component), but not every module is a service.

2.3.4 Composition. *Composition* is the process by which two or more modules are connected (coupled) to form a larger entity that functions as a whole. By this definition, the composed modules can be within the same process, within different processes on the same computing node, or inside processes running on separate computing nodes. Composition is a natural outcome of the modularization of software.

2.4 Application Complexity and Modularization

This section presents the fundamental claim regarding the evolution of HPC software along with the reasoning supporting this claim.

2.4.1 Claim. HPC software development approaches have become increasingly modular to manage software complexity. As a direct consequence of this modularization, performance analysis tools have had to reinvent themselves to stay relevant and practical.

2.4.2 Reasoning. By modularizing software, the complexity of software is compartmentalized [26], and modules become re-usable. Individual teams or developers can focus on building just a few specialized modules with clearly defined interfaces instead of having to deal with a massive, complicated codebase representing the entire application. Individual components can be portable by having multiple "backend" implementations. Componentization can also enable fine-grained resource allocation and management. Besides, modularization is attractive because it allows for the rapid composition of modules to create many composed applications targeting different usage scenarios. The complexity in developing HPC software primarily arises from the following sources.

2.4.2.1 Simulation Scale and Fidelity. There are two challenges to programming at large scale. First, parallel programming is inherently hard to get right. There are several classes of software bugs related to "program correctness" that show up only on highly concurrent systems. The one constant in high-end computing has been the need to simulate natural systems with ever-increasing fidelity and at a larger scale. A study of the largest HPC systems globally over the past 25 years [33] supports this claim. Second, with a billion-way parallelism available on modern machines, the software must be constantly re-written and updated to reflect and utilize new sources of parallelism. An analysis of popular large-scale applications such as CESM [52], LAMMPS [53], HACC [54] reveals that each of these applications has consistently been updated with additional modules to simulate individual physical phenomena with increased fidelity.

2.4.2.2 Range of Applications and Platforms. In the past decade, the range and diversity of applications requiring high-end computing capabilities have exploded. The US Department of Energy develops and publishes *mini-applications* called CORAL benchmarks that represent the core computations within applications of national interest. Performance optimization of these benchmarks would likely result in an improvement in the performance of the larger scientific applications they represent. An analysis of the CORAL-1 benchmark suite [55] released in 2014 and the CORAL-2 benchmark suite [56] released in 2020 reveals a telling story. In 6 years, data science and machine learning (ML) workloads have become Tier-1 applications. These data science workloads are *not* regular MPI applications. This explosion in application variety has resulted in the search for a broader set of programming models and supporting services to accommodate the newer applications.

2.4.2.3 Structure of Modern Scientific Research Teams. Due to the number of different components involved in modern scientific software development, it has become impossible for one person or team to develop all the software components [26, 57].

With an increase in the number of interacting components or modules that must simultaneously run at high efficiency, performance data exchange with analysis and monitoring tools has become more complex. When modularization results in black-box software components, it is challenging for performance tools to instrument and extract the necessary performance data. There is tension between the need to manage software complexity and the simultaneous requirement of running software components at their optimal efficiency. The rest of this paper attempts to present evidence in support of this reasoning.

2.5 Process-Local Modularization

This section presents an overview of the development of component-based HPC software. An in-depth account of the techniques implemented by HPC performance tools to analyze component-based software is also discussed.

2.5.1 Important Trends in the Computing Industry. Arguably, the need for designing modular software can be traced to the popularization of object-oriented methodologies in the 1970s and early 1980s [58]. Software complexity had exploded, and the computing industry was beginning to realize the importance of *software architectural patterns* as a way to implement and manage software.

Eventually, the focus on separating concerns led to the development of *component-based software engineering* (CBSE) [59]. CBSE aimed to go beyond object or module re-use by defining *components* as “executable units of independent production, acquisition, and deployment that can be composed into a functioning whole”. The fundamental notions of independent deployment, re-usability, and composition are the recurring themes underlying major revolutions in software architecture over the past two decades.

In the early 1990s, the industry began to adopt component-based software engineering frameworks such as the Common Object Request Broker Architecture (CORBA) [60], the Java Remote Method Invocation (RMI), and the Component Object Model (COM) [61]. Component models aimed to address the shortcomings of object-oriented methodologies. Specifically, components were designed to be modular, re-usable, and language-independent, allowing for their rapid composition to build higher-level functionality.

2.5.2 Component Software For HPC. In the late 1990s, the HPC research community had realized that the ballooning scientific software complexity had to be controlled and managed. Scientific software developers were beginning to develop coupled, multi-physics models for plasma simulations, and nuclear fusion codes. There was a need to employ high-performance, re-usable, plug-and-play components that were language-agnostic.

The Common Component Architecture (CCA) [62, 63, 64, 65, 66] sought to address several shortcomings of object-oriented methodologies, libraries, and commercial component frameworks [60, 61]. First, while object-oriented frameworks had done well to encourage software reuse within a project, they offered little to no cross-project reuse. Second, object-oriented frameworks were limited in their ability to form the basis of component software as they were applicable only in compile-time coupling scenarios. HPC component frameworks required that components expose a way for compatible implementations to be “swapped” at runtime. Third, component frameworks enabled language independence by relying on meta-language interfaces, a feature that was not typically available with object-oriented frameworks at the time. Fourth, the main issue with employing commercial component frameworks in HPC was the exorbitant performance overheads for "local" inter-component interactions. Lastly, multiple component implementations with the same interface could co-exist within a framework. Doing so was not possible with libraries. After initial attempts to develop independent, disparate component frameworks, the HPC community came together to form the CCA Forum, mainly consisting of members from various academic institutions and US Department of Energy laboratories.

2.5.2.1 CCA Model. The objectives of the CCA specification are found in [62]. Specifically, the CCA was designed within the context of single-program-multiple-data (SPMD) or multiple-program-multiple-data (MPMD) codes. The CCA-MPI marriage was destined given the popularity of the MPI programming model and the integration objectives of the CCA specification.

The following list defines some of the critical elements of the CCA model.

- Components: In the CCA model, a component is an encapsulated piece of software that exposes a well-defined public interface to its internal functionality. Notably, this definition allows components to be composed together to form more complex software.

- Local and Remote Components: Components that live within the same address space are local components. Interactions among local components are ideally no more expensive than regular function calls. A process boundary separates remote components. Although the CCA specification supported remote component interactions, it implicitly incentivized components to perform a bulk of the interactions locally, if possible.

– Scientific Interface Definition Language (SIDL): The CCA specification introduced the scientific interface definition language (SIDL) as a means of enabling composition and interaction amongst components written in different languages. Given the prevalence of "legacy" HPC codes written in C and Fortran and the growing use of Python for scripting and analysis tasks, language interoperability was an essential CCA requirement. The SIDL is a meta-language that is used to describe component interfaces. Notably, it supported complex data types, a feature that commercial component frameworks did not support at that time. Other tools such as Babel [67] read in the SIDL specification to generate glue code allowing components written in different languages to interact.

– Frameworks: Frameworks are the software that manage component interactions. They are responsible for connecting components through the use of ports. The notion of a framework implies a certain level of orchestration necessary for the functioning of CCA components.

– CCA Ports: The CCA specification described two types of ports — *provides* and *uses* ports. A component allows access to its functionality through the provides port, and it registers its intent to interact with other components through the uses ports. The framework is ultimately responsible for actuating the interaction by connecting the provides and uses ports.

– CCA Services: Every CCA-compliant framework provides the registered components with a set of essential services. The components access these framework services similar to how they interact with other components — through ports. One of the most important functionalities of the services object is to provide methods for the components to register their uses and provides ports.

- CCA Repository: The CCA specification included a public CCA repository for components. The key idea was to enable the rapid, "plug-and-play" design of scientific applications using off-the-shelf components available in the CCA repository. The other motivation behind providing a repository was to encourage large-scale community reuse of software components and widespread adoption of the CCA specification.

- Cohort: A collection of components of the same type (running within different address spaces) is referred to as a cohort.

- Direct-Connected Framework: As Figure 2 depicts, there are two ways in which CCA components can be composed. In a direct-connected framework, each process consists of the same set of parallel components. A notable feature of such a framework is that it does not allow *diagonal* interactions among components, i.e., inter-component interactions are limited to function calls within a process. Direct-connected frameworks only support the SPMD model of parallelism. Parallel components within a cohort can interact through any distributed communication library available. This latter type of communication was outside the scope of the CCA specification.

- Distributed Framework: Distributed frameworks, depicted by Figure 2, allow diagonal interactions and a more general MPMD model of parallelism. Specifically, inter-component interaction can occur between components belonging to different processes. In addition to providing a remote-method-invocation (RMI) interface, distributed frameworks need to address data distribution between coupled applications. This problem shall be revisited in Section 2.6.

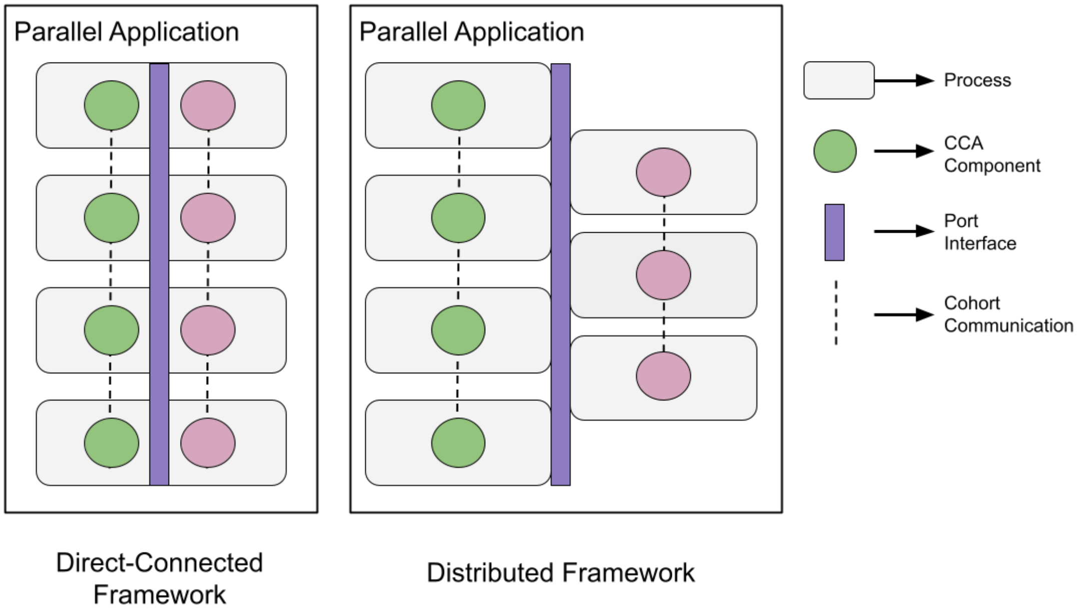


Figure 2. CCA: Framework Types (inspired from DCA [1])

2.5.2.2 CCA: Performance Measurement. Given the composition model of CCA applications, it was imperative to generate a performance model of the component assembly and judge the efficacy of the instantiation [68, 69]. Specifically, the community found it necessary to measure local component performance and inter-component interactions in a non-intrusive, cohesive way. These measurements would then be used to generate performance models of individual components and their interactions. Finally, the generated performance models would be employed to select an optimal set of components for the particular application context. The research on performance measurement and analysis of high-performance CCA applications can be divided into two categories: (1) Intra-component performance measurement and (2) Inter-component performance measurement.

Intra-component performance measurement entails capturing performance data about the execution of functions and routines within a component. Literature [28] presents two ways in which intra-component interactions can be captured:

- Direct Instrumentation: This is the most straightforward way to extract measurement data. Existing HPC performance tools could instrument component routines directly (either manually or through automatic instrumentation). The component invokes the measurement library to generate the necessary performance data.

- Performance Component: Direct instrumentation techniques suffer from the disadvantage of being tightly coupled with component implementations. The use of an abstract measurement component interface and a performance component that implements the measurement interface circumvents this problem of tight-coupling, and it allows for a more flexible approach to performance measurement of intra-component interactions. Specifically, any compliant performance tool (TAU being just one example) can implement the performance component interface. Moreover, the use of an abstract measurement interface ensures that the overheads of performance instrumentation are effectively zero when no performance component is connected.

Inter-component performance measurement is necessary to study the interactions between components. Specifically, components are connected via provides and uses ports. The interactions between components occur on these ports and contain valuable information such as data transfer sizes and source and destination identifiers used in message-passing routines. These interactions are not visible to an external entity, and thus, unique instrumentation is required to capture them.

The instrumentation and measurement techniques used for inter-component performance analysis can be categorized as either (1) direct instrumentation, (2) instrumentation during interface definition and generation, or (3) Proxy-based instrumentation and measurement.

- Direct Instrumentation: A few CCA frameworks, such as CCAFFEINE [70], are based entirely on C++ as the language for implementing component interfaces. In such scenarios (often not the case), direct instrumentation techniques can measure and observe inter-component interactions.

- Instrumentation of Interface Generation Code: When component interfaces are specified using an interface definition language such as SIDL [71], direct instrumentation can be applied only once the interface language compiler (such as Babel) has generated the language-specific component interface glue code. Another approach is to build the instrumentation directly into the process of generating the glue code. Both approaches are feasible. However, the latter technique is likely to yield more optimal code [28].

- Proxy-based Instrumentation and Measurement: Arguably, the most popular technique to instrument and measure component interactions involves component proxies to snoop for invoked methods on the provides and uses ports [68, 72]. Component proxies are stub component implementations presenting the same interface as the components they represent. Component proxies placed “in-front” of “caller” components trap method calls on the uses ports to enable performance measurement. A “Mastermind” component invokes the measurement API of a backend performance component (such as the TAU component) and is responsible for storing and exposing the performance data for external analysis and query.

2.5.2.3 CCA: Performance Monitoring and Optimization. Aside from managing software complexity, component frameworks also present *logical* boundaries for performance optimization. Recall that, unlike standard libraries, the CCA component specification allowed multiple component implementations presenting the same interface to coexist within the application. In a CCA-enabled application, a sub-optimal component can be dynamically replaced with a more optimal component. For example, in a scientific application composed of *solver* components performing a linear-algebra calculation, the solver component implementation can be switched at runtime depending on how well the component performs on traditional metrics such as execution time as well as *functional* metrics such as solver residual.

Literature [73, 74, 26] describes *computational quality of service* (CQoS) as a general methodology for optimizing CCA-enabled application. CQoS is the "automatic selection and configuration of components to suit a particular computational need". Essentially, the selection of an optimal set of components involves a trade-off between accuracy, performance, stability, and efficiency [74]. The cycle of performance measurement and optimization of CCA applications has four distinct parts: (1) performance measurement, (2) performance analysis, (3) performance model generation, and (4) a control system to implement optimizations.

Performance measurement has been discussed in Section 2.5.2.2. Thus, here we discuss performance analysis, performance model generation, and control systems for CCA optimization. The application is instrumented to report traditional performance metrics such as the execution time and component-specific *functional* performance metrics such as the solver residual. This performance information is used to train analytical and empirical models of component performance. Specifically, the performance information is written to a performance database component [75]. The analysis tools query the database component to generate *component performance models*. These component models are written into a "substitution assertion database" that acts as the link between the analysis and control infrastructure.

The control system is driven by *control laws* that dictate the actions of the control infrastructure. Control laws are essentially the "rules" that drive dynamic adaptation of CCA components. A control law executes by combining the application's state information with the appropriate model information within the substitution database to output a recommendation for optimization. The control infrastructure is ultimately responsible for implementing the recommendation.

The control infrastructure consists of the *reparameterization decision service* and the *replacement service* as the critical pieces. CQoS control is accessed seamlessly via proxy components. The use of a proxy allows applications to benefit from CQoS with minimal addition of intrusive instrumentation. Further, CQoS can be dynamically turned on or off. When CQoS is disabled, proxy components function as gateways to CCA performance measurement. When CQoS is enabled, the proxy component is connected to the optimization components that inform the proxy of the optimal provides port to use (among many candidate component ports). Effectively, the proxy component functions as a switch that connects the caller (application component) with the optimal implementation of the callee component.

2.5.3 Other HPC Component Frameworks. Although the CCA specification formed the majority of efforts to componentize HPC software, there were other similar projects that had related goals.

2.5.3.1 High-Performance Grid Component Frameworks. Component-based grid scientific computing infrastructures explore methodologies for optimization [76] that share similarities with CCA. First, like CCA, grid component compositions are indicated in the component metadata. CXML is a markup-based composition specification that is similar to the SIDL language used in CCA frameworks. Such a specification enables the component framework to generate and analyze static call graphs. Second, the "application mapper" is an optimization component that functions as the control system within the framework. The application mapper takes the abstract component composition (known as an application description document) and generates a runtime representation of the composition. It takes system resource metadata as input from the grid deployment services and combines the existing component performance models to form an optimal *execution plan.* If there is a change in grid resources, the grid application can contact the application mapper at runtime to generate a new execution plan.

Aside from the hardware on which they operate, there are two crucial differences between high-performance grid component frameworks and traditional high-performance computing frameworks such as CCA. First, although both frameworks operate on distributed systems, inter-component interactions in grid frameworks typically involve the network. In direct-connected CCA frameworks, most inter-component interactions are reduced to a sequence of regular function calls. As a result, grid component frameworks are designed more like distributed "services", and traditional HPC frameworks operate more like libraries. Second, grid component frameworks assume that the component repository contains performance model metadata that allows the application mapper to make reconfiguration decisions. In other words, *dynamic* component reconfiguration is treated as a first-class design requirement, given that the fluctuation in available resources is a common occurrence. The CCA specification as such does not pay special attention to ensuring the dynamic reconfiguration of components. Instead, CCA treats dynamic component reconfiguration as an activity to be performed on a per-application need basis. Despite this, the HPC community has invented clever ways of seamlessly and incrementally integrating control capabilities through the use of proxy components.

2.5.3.2 Low-Level Component Framework (L2C). The L2C [77] attempts to address the issue of *performance portability* on HPC systems. The authors observe that multi-platform support for large applications is usually achieved through means that offer little code reuse (conditional compilation, component software, runtime switches). They attempt to resolve this problem through a low-level component model that is (1) composable, (2) offers portable performance, and (3) offers a high degree of code reuse. Components are implemented as *annotated objects* with well-defined entry points, resembling a plugin architecture.

The components are written in C++, Fortran, or Charm++. Multiple instances of a particular component type can co-exist within the application. The composition is specified through an L2C assembly descriptor file. A small L2C runtime is responsible for managing the component interactions. Notably, there is no support for multi-language component compositions (and associated glue code generation). The key idea lies in breaking down the application into several fine-grained components. Doing so allows for a high degree of code reuse between performance-portable implementations designed for different platforms. This design choice also leads to an explosion in the number of components required to implement even a relatively simple application such as a Jacobi solver.

2.5.3.3 directMOD Component Framework. HPC applications such as adaptive mesh refinement (AMR) codes form a particular category of applications whose structure (data and communication distribution) changes over time. As the application structure informs the component assembly, AMR applications require a component framework that supports dynamic reconfiguration. Not only this, multiple dynamic reconfigurations co-occur inside different application processes, introducing synchronization and consistency issues among them. The directMOD [78] component framework addresses these problems by offering a component model that introduces two new concepts: domains and transformations. Domains are components that lock specific portions of the application and ensure safety. Transformations are ports that connect a transformation to its target sub-assembly. However, directMOD is not broadly applicable to any general application.

2.5.4 Comparing Component Frameworks. A comparison of the various component frameworks is presented in Table 1. Several of these comply with the CCA architecture. CCAFFEINE [70] was intended to be a model implementation of the CCA specification. Notably, it is the only major HPC-optimized CCA implementation that does not support the MPMD model. In other words, all CCAFFEINE-enabled applications are limited to SPMD parallelism, and peer components interact through direct connections only. Some component frameworks such as MOCCA [79], VGE-CCA [80], CCAT [81], LegionCCA [82], and XCAT3 [83] are not HPC-optimized. These frameworks are geared primarily to operate in grid environments, and thus, the distributed communication libraries they employ are not HPC-aware.

Table 1. Comparing Component Frameworks

Property	**CCaffeine**	**DCA**	**XCAT3**	**Uintah**	**SCIRun2**	**MOCCA**	**VGE-CCA**	**LegionCCA**	**L2C**	**directMOD**	**CCAT**
CCA Compliant?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Optimized for HPC?	Yes	Yes	No	Yes	Yes	No	No	No	Yes	Yes	No
General Purpose?	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes
Distributed Communication Support	N/A	RMI	Multiple	RMI	RMI	Multiple	SOAP-RPC	RMI	MPI	MPI	RMI
Built-In Performance Optimization?	No	No	No	Yes	Yes	No	Yes	No	No	Yes	No
Cross-Framework Compatibility?	No	No	No	No	Yes	Yes	No	No	No	No	No
Cross-Language Support?	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
Code Reuse	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	Moderate	High	Moderate	Moderate

Among the HPC-optimized frameworks, SCIrun2 [84] and Uintah [85] are the only CCA frameworks that have some form of in-built performance optimization support for dynamic reconfiguration of their components. However, the performance analysis and optimization techniques discussed in Section 2.5.2.2 and Section 2.5.2.3 can be generally employed in CCA frameworks that do not have built-in support. SCIRun2 is the only HPC-optimized framework that supports cross-framework compatibility. That is, SCIRun2 is a *meta-framework* that allows interoperability of components adhering to different specifications. For example, SCIRun2 allows the composition of a CCA component with a CORBA component. It is also worth mentioning that most CCA frameworks support some form of data distribution among components that run within a coupled, MPMD-style architecture. A detailed discussion of this support is presented in Section2.6.5.1.

2.5.5 Scientific Computing Frameworks.

Component architectures have helped manage the complexity of HPC software development and increase developer productivity. At the same time, there have been other noteworthy, smaller-scale efforts in this direction. Specifically, scientific computing frameworks such as POOMA [86], PETSc[87], HYPRE [88], Grace [89], and OVERTURE [90] have enabled the rapid development of scientific software from "building blocks". These frameworks are built into libraries and export an interface in either C, C++, or Fortran (the three most commonly used languages to develop HPC software).

Except for HYPRE, the building blocks used to compose higher-level functionality are explicitly implemented as objects.

While these frameworks share with component architectures the general principle of composition, they target a different user and application space. Table 2 enlists the similarities and differences between scientific computing frameworks and the CCA component framework. Most importantly, scientific frameworks are tailored for a specific, narrow domain and are not applicable to build arbitrary HPC applications. HYPRE, for example, is a library of pre-conditioners for use in linear algebra calculations. On the other hand, PETSc is a library designed specifically for matrix operations. Given that scientific frameworks are implemented as libraries, traditional performance analysis techniques such as library interposition, sampling, and compiler instrumentation can be employed directly without any special modifications. Compared to CCA, scientific frameworks typically offer an unmatched speed of development, productivity, and out-of-the-box performance for applications that fit the particular domain supported by the framework. However, scientific frameworks generally offer little to no support for adaptivity. It is generally assumed to be the responsibility of the application developer for fine-tuning the performance of the library on a novel platform.

Table 2. Comparing Scientific Frameworks and CCA

Property	**Scientific Frameworks**	**CCA**
Domain Specific?	Yes	No
Distributed Computing Support?	No	Yes
Cross-language Support?	Partial	Yes
Traditional Performance Tools Applicable?	Yes	No
Support for Performance Portability?	Moderate	High
Speed of Development of Higher-Level Functionality?	High	Moderate
Support for Adaptivity?	Moderate	High

2.5.6 Modularization of MPI Libraries.

MPI libraries were among the first to adopt a modular architecture. The rising complexity of MPI library implementations, the need to be portable across HPC platforms, and the scale of development teams were the primary motivating factors behind the push to modularize MPI libraries. Three prominent MPI libraries are discussed, compared, and contrasted based on how they choose to implement modular architectures.

2.5.6.1 LAM/MPI.

The LAM/MPI project [91] was the first production-ready MPI implementation to implement a component architecture explicitly. The LAM project was initially structured as an extensive collection of source files and directories. However, it was observed that new developers found it increasingly hard to understand the source code, contribute to the project, and experiment with novel optimization strategies. As a result, the LAM project adopted a component architecture focused on being lightweight, high-performance, and domain-specific, as opposed to more general frameworks such as the CCA [65] architecture.

LAM/MPI supports four types of components — the RPI (Request Progression Interface) component, the COLL (COLLector) component, the CR (Checkpoint-Restart) component, and the BOOT (BOOTstrapping) component. LAM supports multiple implementations of the same component type (through a plugin framework) to coexist within an MPI process. Doing so allows for a dynamic selection of component implementation to optimize runtime behavior. Notably, the re-implementation of LAM using a component architecture improved MPI communication performance by a small margin.

2.5.6.2 OpenMPI.

The OpenMPI project [92] is a successor to the LAM/MPI library. The OpenMPI community recognized the need for an MPI library that explicitly supports and encourages third-party developer contributions.

When OpenMPI was being developed, algorithms for process control, fault tolerance, checkpoint-restart, and collective communication were beginning to form separate research areas in their own right. Thus, there was a need to support several different versions of these algorithms within a single larger framework.

At the heart of the OpenMPI implementation is a component architecture that aims to resolve both of these challenges. The design element that sets OpenMPI apart from previous implementations is a *multi-level* component architecture. The principal, higher-level component framework ("meta-framework") supports several component frameworks underneath. Each of these lower-level component frameworks targets one specific function, such as collective communication or checkpoint-restart. Further, these lower-level component frameworks manage one or more modules. It is the responsibility of the individual component frameworks to load, discover, and manage the life-cycle of their respective modules. Like LAM/MPI, OpenMPI modules are implemented as plugins. They are integrated into the MPI library statically or as shared libraries, allowing for compile-time or runtime module discovery and initialization.

2.5.6.3 MVAPICH2. MVAPICH2 [39] is a state-of-the-art, high-performance MPI implementation that does not explicitly follow a component architecture. However, due to the same factors described in Section 2.4.2, the design of the library has become increasingly modular over time. Figure 3 depicts this modularization of MVAPICH2. The current version of the library delineates the same set of *logically separate* modules as OpenMPI — fault tolerance, job startup, and collective algorithms. However, these separate modules are not explicitly managed as independent units. Therefore, the primary method by which MVAPICH2 allows an external user to control its behavior is through the use of environment variables

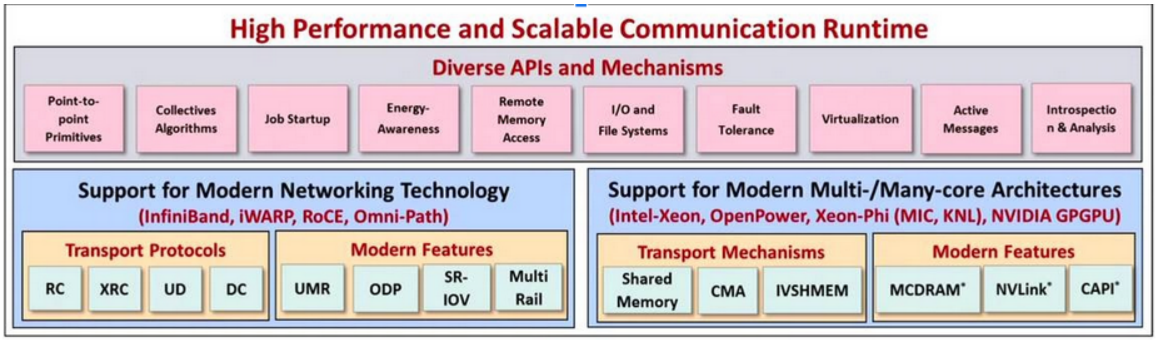


Figure 3. Modularization of MVAPICH2 (image credits: Dr. D.K. Panda, Network Based Computing Lab, The Ohio State University)

and compile-time configuration flags. Notably, MVAPICH2 does not support custom implementations of these modules, nor does it offer an easy way to replace them at runtime dynamically. Arguably, its monolithic architecture makes it more difficult for external contributors to make changes to the library source code.

2.5.7 Tools for Performance Data Exchange. A side-effect of the increasing software complexity and scale is an update in how performance tools instrument the software to measure performance. While modularization can be considered a good engineering practice and a necessity when considering the sizeable cross-institutional nature of HPC software development, modularization hinders the exchange of necessary performance essential information between software layers. At the same time, the use of modular software on large, high-end computing systems has (1) necessitated their dynamic, online adaptation and (2) enabled fine-grained optimization of the various modules and algorithms that comprise the software. Traditionally, HPC performance tools have been passive participants in the optimization of HPC applications. They are primarily employed for offline analysis of performance data. The various costs associated with running applications at exceedingly large scales have motivated the tighter integration of performance tools into the software stack.

2.5.7.1 MPI Tools. There have been various attempts to design and implement techniques that enable closer interaction between a performance tool and the MPI library. PERUSE [93] was an initial attempt at using callbacks to gain access to significant events that occur inside the MPI library. The performance tool installs callbacks into its code for events that it is interested in measuring. When these events occur, the tool callback is invoked, allowing the tool to gather and analyze the pertinent performance data. Ultimately, PERUSE failed to gain traction in the MPI community due to a mismatch between the proposed events and the capabilities of existing MPI implementations.

The MPI Tools Information Interface (MPI_T), introduced as a part of the MPI 3.0 standard, has received significantly more attention from tool developers than previous efforts. MPI_T defines two variable types — performance variables (PVARs) and control variables (CVARs). Tools need to query the MPI_T interface to access the list of PVARs and CVARs that the MPI library wishes to export. PVARs represent counters and resource usage levels within the MPI library, while CVARs represent the "control knobs" that can affect dynamic MPI library reconfiguration. Several tools have been developed [94, 95, 96] to take advantage of the MPI_T interface to gather performance data, while only one previous work [95] implements a tool architecture that enables the dynamic control of the MPI library through CVARs. MPI libraries such as MVAPICH2 and OpenMPI export a plethora of PVARs to be queried at runtime, but they currently lack support for CVARs that control *online* behavior. As a result, the effective use of MPI_T for dynamic reconfiguration of MPI libraries remains an open problem. More recently, callback-driven event support through MPI_T is once again gaining traction within the MPI community [97].

2.5.7.2 OpenMP Tools. The first notable effort to enable profiling of the OpenMP runtime is the POMP [98] profiling interface. POMP was designed as an OpenMP equivalent of the PMPI interface for MPI applications. POMP allows for seamless, portable profiling of OpenMP sections to gather context information using the OPARI source-to-source instrumentor. However, it can impose noticeable runtime overheads for short-running loops. More importantly, POMP does not allow access to internal OpenMP runtime information and thus has limited application in gathering internal event data. The Sun (Oracle) profiling interface [99] implements a callback-driven model to gain partial access to OpenMP runtime state through the asynchronous sampling of call stacks. However, a lack of support for static executables and gathering of context information resulted in the interface not gaining traction within the community.

Like MPI, these various efforts to enable low-overhead profiling and tracing of OpenMP applications have culminated in developing the OpenMP Tools Interface specification (OMPT)[51]. OMPT is a *standard* that defines how tools and OpenMP runtimes should interact to enable profiling and tracing of OpenMP applications. It borrows ideas from past efforts to present a callback-driven interface that supports several *mandatory* and *optional* events. Each supported event is associated with a specific data structure provided to the tool for generating context information. Additionally, the OpenMP runtime manages state information on a per-thread basis. Since its introduction into the OpenMP standard, the OMPT interface has grown to support callback-driven profiling of accelerators such as GPUs.

2.5.7.3 PAPI SDE. While the MPI and OpenMP tool interfaces are limited to enabling performance data exchange within their respective domains, the PAPI Software-Defined-Events (SDE) [100] is an attempt to standardize the exchange of software performance counters between *any* two software layers within a process. The PAPI SDE project recognizes that library-specific approaches such as MPI_T, albeit standardized, are not widely applicable directly. Through the existing PAPI API, software modules can export software performance metrics of interest to other libraries or modules running within the process. There are three ways in which SDEs can be created and used. One, a library can declare an internal variable as an SDE to be read directly by other modules. Two, the library can register a callback that returns the value of the variable. Three, the library can create and update a variable that lives inside the PAPI library.

2.5.7.4 Comparing Techniques for Performance Data Exchange.

Table 3 compares various tools on the basis of how they enable performance data exchange. Notably, the PAPI SDE effort is unique in its ability to be generally applicable to *any* type of software module. Over the past fifteen years, the HPC community has iterated on various designs for performance introspection, and it can be argued that event-based callbacks are generally favored over instrumentation. Moreover, the push to include performance introspection capabilities as a part of the standards specification of major communication libraries such as OpenMP and MPI has resulted in the widespread adoption and support of performance tools. In other words, performance tools are increasingly viewed as first-class citizens as opposed to an afterthought within the performance optimization process.

A notable limitation of the MPI_T effort is a lack of tool portability. Specifically, the MPI_T standard allows the MPI implementation the freedom to export any counter or gauge. The standard does not require any mandatory counters or events to be exported. As a result, performance tools need to discover the specific list and names of PVARs and CVARs exported by an MPI library. These names and types

Table 3. Comparing Techniques for Performance Data Exchange

Property	**PERUSE**	**MPI_T**	**POMP**	**Sun OpenMP Interface**	**OMPT**	**PAPI SDE**
Data Exchange Strategy	Event Callbacks	Counters, Event Callbacks	Source-to-Source Instrumentation	Event Callbacks	Event Callbacks	Counters
Generally Applicable?	No; MPI-only	No; MPI-only	No; OpenMP-only	No; OpenMP-only	No; OpenMP-only	Yes
Access to Fine-Grained Events?	Yes	Yes	No	Partial	Yes	Yes
Standardized Technique?	No	Yes	No	No	Yes	No
Widespread Support?	No	Yes	Yes	No	Yes	Yes
Direct Support for Control?	No	Yes	No	No	No	No
Level of Insight into Module Internals?	High	High	Low	Moderate	High	Low
Tool Portability?	Moderate	Low	High	Moderate	High	Low
Support for Accelerators?	No	Yes	No	No	Yes	Yes

are not portable between MPI implementations, and thus, there is little reuse for tool logic that generates performance recommendations or optimizations. OMPT, on the other hand, resolves this problem by separating salient events into mandatory and optional events. This approach can facilitate tool portability across different library implementations.

2.6 Distributed Modularization

This section presents how distributed modularization of HPC software has resulted in the formation of coupled application ("in-situ") workflows, ensembles, and services. At the same time, an overview of the accompanying changes within the performance analysis and monitoring tools landscape is also discussed.

2.6.1 Overview.

Since the late 1990s, there have been several efforts to support *task-coupling* on HPC systems. Specifically in this context, task-coupling is defined as the simultaneous execution of two or more distributed entities in an inter-dependent manner. The shift towards a coupled distributed architecture began in the late 1990s and early 2000s. This process has been accelerated in the last decade by several factors described in Section 2.4.2. Logan et al. [101] define three categories of task-coupling that capture how emerging HPC software architectures are being designed. In a *strongly coupled* architecture, the coupled entities are tightly integrated and intimately dependent on one another to make progress. Most multi-physics applications such as the XGC-GENE [102] coupled code operate with an assumption of strong coupling.

In a *weakly coupled* architecture, the producer of data can proceed without being concerned about how the data is being consumed. This interaction model is the *modus operandi* for most in-situ data analysis, visualization, monitoring, and ML services. These services run alongside a "primary" application, typically an MPI-based simulation. Sarkar et al [103] allure to this type of software architecture by giving it the moniker "new-era weak-scaling".

A third emerging type of a coupled architecture is *ensemble computing*. Ensembles find application in domains such as weather modeling, molecular biology [24], and the training of ML models [23]. Ensembles involve simultaneously executing collections of parallel tasks (each of which may be an MPI application) within a single HPC node allocation. Deelman et al.[104] refer to this kind of architecture as "in-situ workflows", distinguishing them from "distributed workflows" that span multiple HPC platforms and scientific instruments. Unless specified, the primary focus of this document is to delineate the critical questions surrounding in-situ workflows.

Distributed modularization has several benefits. By breaking up a large, monolithic code into several smaller distributed modules, the user has increased control on scaling individual entities. In doing so, the application can be executed on a larger node count as compared to a traditional monolithic MPI-based executable. Two, when software modules are deployed as separate parallel executables, it allows larger software development teams to collaborate without worrying about the logistical issues associated with a mammoth code base. Third, specific capabilities such as data-processing and ML-based analysis tasks can not be leveraged directly within the constraints of the MPI programming model. Thus, their integration requires a software architecture that allocates a set of dedicated computing resources for their operation. Four, when modules run inside separate processes, they can be implemented in the language that best suits their specific need. The development of a separate, intermediate language-interoperability tool such as Babel [67] is not required — this job is usually performed by the communication library.

However, several challenges need to be addressed when considering a distributed, modular architecture. First, it is necessary to identify, among the available options, the correct way of splitting up the monolithic application into the constituent (parallel) modules. Second, even when the modularization itself is straightforward, it is not always clear how to allocate the appropriate computing resources to each parallel module. An optimal configuration can be orders of magnitude more performant than a haphazardly configured setup. Third, distributed modularization can result in vast amounts of data traversing the network between the parallel modules. Thus, an efficient middleware or data-transfer mechanism assumes vital importance. Four, when dealing with multiple simultaneously executing components and transient services, performance monitoring and analysis challenges are notably different from those posed by traditional monolithic MPI executables. Distributed components require performance data to be extracted, exported, aggregated, and analyzed online from multiple sources. Given the transient nature of these distributed components and the scale of operation, it is often infeasible for this data to be written out to disk and analyzed offline. We touch upon these opportunities and challenges in the sections that follow.

2.6.2 Important Trends in the Computing Industry. A key observation that can be made when surveying the origins of several defining shifts in HPC software development methodologies is that they are usually predated by similar changes within the broader computing industry. Specifically, two computing industry trends bear importance in the context of distributed modularization. The first of these is the notion of a *service-oriented-architecture*[4]. This reference model for SOA defines “services” as to how needs and capabilities are brought together. Fundamentally, SOA embodies the principle of “separation of concerns”. Further, the services within an SOA are assumed to have potentially different owners (software development teams) and are developed and deployed independently of each other. SOA architectures directly reflect the structure of software development teams within a larger organization. Arguably, this bears a resemblance to the emerging methodologies for designing and deploying coupled HPC applications.

The other relevant trend is that of the “enterprise service bus” (ESB) [105]. The ESB is the mechanism by which different services within a framework discover and connect. The ESB assumes the job of connecting a service requestor with a service provider, transporting the message requests correctly, implementing load-balancing, and making the necessary protocol conversions. In other words, the ESB functions within the confines of a “publish-subscribe” model of distributed communication. It orchestrates the interaction between the different entities in the system. Arguably, data transfer and staging software such as ADIOS [5] and DataSpaces [8] perform the same duties within a coupled HPC workflow.

2.6.3 Composition Model. This section presents a discussion of the various types of distributed HPC frameworks and compares them based on their composition models, coupling strategies, and distributed communication protocols. A *framework* in this context is defined as any software that either (1) functions as a standalone distributed component offering a distinct functionality or (2) enables the development of specialized distributed components through a programming library or platform. This study does not regard I/O libraries such as ADIOS and Decaf [9] as "frameworks". Instead, they are a part of a larger body of work addressing the problem of distributed data management and are discussed separately in Section 2.6.5.2. At the same time, it is worth mentioning that there is significant overlap between in-situ analysis tools, data services, and data management libraries. This overlap is depicted by Figure 4.

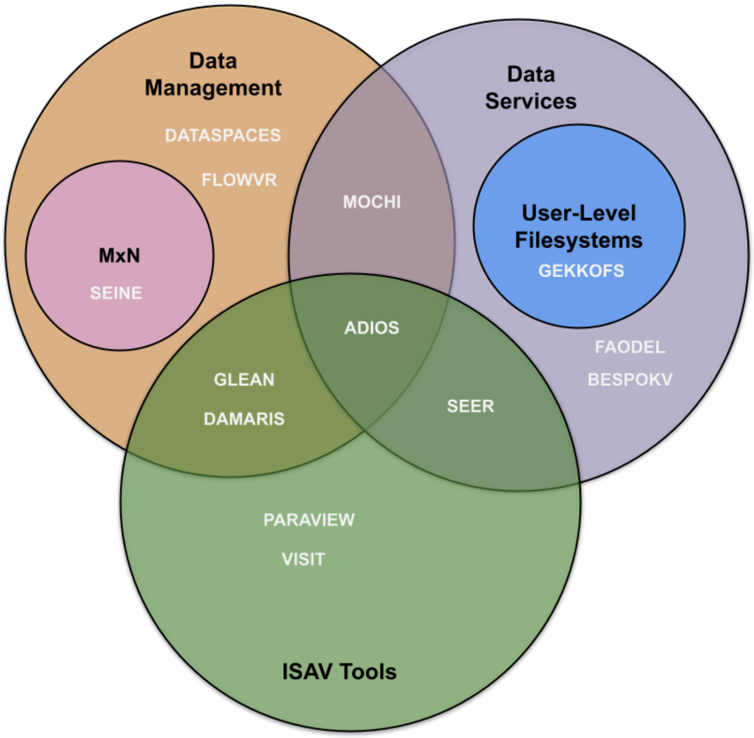


Figure 4. Overlap Between In-Situ Analysis and Visualization (ISAV) Tools, Data Management Libraries, and Data Services

Composition models define the functional relationship between coupled modules or tasks. Here, we follow and extend the definitions for composition models provided by Logan et al. [101]:

- Strongly-coupled: Two or more coupled modules are tightly integrated and interact in a back-and-forth manner to exchange data during execution.

– Weakly-coupled: Distinct sets of producers and consumers characterize this composition model. Notably, the execution of producer logic does not depend on the consumer's execution, performance, or failure state.

– Hybrid: In a hybrid model, the coupled modules can either be strongly-coupled or weakly-coupled depending on how the system is set up.

– Ensemble: An ensemble represents a distinct type of distributed coupling between tasks, and it is characterized by the execution of a large number of concurrent tasks (each of which may be an MPI application). The tasks may or may not depend on one another. When the tasks are completely independent, the ensemble is "fully decoupled". Such a model resembles "embarrassingly parallel" computation, except that it occurs at a higher level of task granularity.

A coupling strategy determines how the framework functionality is accessed from an external component. For example, the remote procedure call (RPC) is a popular coupling strategy for HPC data services, while most CCA component frameworks are limited to using the parallel remote method invocation (PRMI) model for distributed, inter-component interaction. Table 4 presents a list of popular distributed HPC frameworks covering a broad spectrum, a brief description of the functionality enabled by each of these frameworks, and their composition models. Table 5 is a list of the coupling strategies and the internal communication protocols used by these particular frameworks.

2.6.3.1 Distributed CCA Frameworks. Distributed CCA frameworks were among the first HPC software to develop a solution to enable communication between distributed CCA components. Several efforts such as DCA [1] and PAWS [106] recognized that the CCA specification primarily targeted SPMD parallelism within direct-connected frameworks and had little provision or advice for enabling distributed frameworks. However, when CCA frameworks were beginning to be integrated into production HPC applications, the community realized the need to enable CCA-based componentization of multi-physics codes such as the XGC-GENE [102] code, molecular dynamics applications such as LAMMPS [53], and fusion codes.

Each module in a coupled code is typically implemented as a separate MPI application consisting of one or more CCA components. Within a purely direct-connected framework, enabling collective port invocation for a subset of components is straightforward — the set of communicating processes can make use of a separate MPI communicator. The CCA framework need not be involved in this collective communication process (refer to Figure 2). However, when the collective port invocation needs to happen between two parallel, distributed components belonging to different MPI programs (as in a coupled code), several challenges arise. First, the use of MPI communicators is meaningless when passed across different MPI programs. Second, the CCA framework needs to know and decide which set of processes (or components) participate in a collective, distributed port invocation. Third, there needs to be an agreed-upon synchronization strategy for components participating in the port invocation. Fourth, the framework needs to know how to distribute the data between the callee and the caller components involved in the collective port invocation call. The CCA forum recognized these issues and drafted a communication model called the parallel remote method invocation (PRMI).

In the PRMI model, the callee component blocks until the caller has completed the method invocation. When this requirement is imposed in a collective communication routine involving the participation of multiple components in the RMI port call, it categorizes distributed CCA frameworks as implementing a strongly-coupled composition model. Specifically, the PRMI call serves as a potentially unwanted synchronization point and reduces the system's effective concurrency. At the same time, the coupled codes that were built using distributed CCA frameworks *required* such patterns of communication between M processes of one component and N processes of another. The data redistribution issues that arise from this communication pattern are broadly referred to as the *MxN* problem and shall be discussed in depth in Section 2.6.5.2.

Among the CCA frameworks described in Table 1, CCAFFEINE [70] is a purely direct-connected framework. MOCCA [79], VGE-CCA [80], XCAT [83], CCAT [81], and LegionCCA [82] support distributed component interactions through RMI, but these frameworks are optimized for scientific applications in the grid as opposed to those employed on HPC platforms. The individual components in these frameworks are not MPI programs, and as a result, the restrictions of the PRMI model do not apply. The grid frameworks employ RMI over SOAP/HTTP or another web-services protocol for distributed interaction. The two notable general-purpose, HPC-optimized distributed CCA frameworks are DCA [1] and SCIRun2 [84]. As depicted in Table 5, both DCA and SCIrun2 employ MPI for internal communication within a component cohort. DCA allows collective PRMI communication among a subset of caller components by re-using the MPI communicator support but stipulates that all the caller components take part in the communication. SCIrun2, on the other hand, provides two types of PRMI calls: (1) independent calls that involve one component on both sides of the caller-callee cohort pair, and (2) collective calls that involve every component on both sides of the caller-callee cohort pair [107].

In a more recent work that bears a resemblance to distributed CCA, Peng et al. [108] propose a new strategy to decouple MPI applications into sets of custom process groups. This research stems from the need to address the scalability limitations of the BSP programming model, particularly concerning load imbalance. Instead of building BSP programs where each process is essentially a replica of the same executable (SPMD), the authors propose to break down the application's functionality into a set of specialized operations. The process space is divided into groups of processes implementing these specialized operations. These process groups are organized into a cohesive distributed processing system through a data stream pipeline. An evaluation of this methodology on a Map-Reduce application at a large scale improved the performance of the application by over four times as compared to a standard BSP-style implementation. Arguably, this distributed architecture is one step toward a services-style coupled model that is commonly used in the broader computing industry.

2.6.3.2 HPC Data Services. Two distinct trends have given rise to a class of applications broadly categorized as "data services". The first noteworthy trend is that the performance of traditional file-based parallel HPC I/O storage systems has not been able to keep up with the increase in the concurrency available on the platform. In other words, the total computational performance is growing faster than the total storage performance of the HPC system. As a result, these systems are forced to integrate faster storage technologies such as burst-buffers, non-volatile storage-class memories, and NVMe technology to provide a cost-efficient, performant storage stack.

The second trend is the broadening of the variety of HPC applications and accompanying I/O access patterns that need to be supported on these platforms.

Table 4. Distributed HPC Frameworks: Composition Models

Framework	**Short Description**	**Composition Model**
DCA	Distributed CCA framework	Strongly-coupled
SCIRun2	Distributed CCA framework	Strongly-coupled
Mochi	HPC data service	Hybrid
Faodel	HPC data service	Hybrid
BESPOKV	HPC data service	Hybrid
ParaView Catalyst	In-situ viz. and analysis	Weakly-coupled
VisIt Libsim	In-situ viz. and analysis	Weakly-coupled
SENSEI	In-situ viz. and analysis	Weakly-coupled
Ascent	In-situ viz. and analysis	Weakly-coupled
TINS	In-situ analysis	Weakly-coupled
Henson	In-situ analysis	Weakly-coupled
Damaris-viz	In-situ viz. and analysis	Weakly-coupled
Seer	In-situ steering	Hybrid
Swift/T	HPC dataflow programming	Ensemble
RADICAL-PILOT	HPC task-based ensembles	Ensemble
Merlin	ML-ready HPC ensembles	Ensemble

The traditional interaction between MPI-based HPC applications and the storage system is characterized by an input read phase and one or more large, parallel bulk-synchronous writes of structured data for check-pointing purposes. Machine-learning and data-intensive applications such as CANDLE [23] are characterized by irregular read access and the writing of a large number of small files [6]. Further, these new HPC applications require various data types such as key-value (KV) stores and document stores.

Although there are areas of overlap, data services are distinct from data management libraries. Data services offer transient, high-performance data storage and notably richer functionality than just helping move data between different components in the workflow. Broadly, there are two classes of data services that are of interest — those that function as user-level, distributed file systems, and more general programmable data services that can be employed to serve various application needs. The latter class of data services is of primary interest to this study. General programmable data services such as Mochi [6] can be used to build custom distributed file systems.

There have been attempts to leverage the portability and performance offered by MPI for building HPC file systems and storage services [109, 110]. These studies conclude that while MPI is sufficiently capable of serving as the platform upon which these services can be built. However, there are several missing features ("wish-lists"), if implemented, would give MPI the best chance of widespread adoption. Specifically, these features include extended support for non-blocking calls, one-sided communication, and the flexibility to continue operating in a situation of failure.

User-level distributed file systems have been developed primarily to improve the application performance on platforms that support burst-buffers or node-local, fast storage hardware. Examples of the state-of-the-art, user-level distributed file systems include FusionFS [111], GekkoFS [112], and UnifyFS [113]. These file systems can be seamlessly integrated into any HPC application by specifying a mount point for storage operations. The user-level file system intercepts regular POSIX I/O calls and routes them to the burst-buffer if the file path matches the mount point. GekkoFS implements relaxed semantics for POSIX I/O calls. GekkoFS and UnifyFS employ a background daemon to serve local client requests and store file metadata.

General programmable data services are fundamentally different from user-level file systems in two ways. First, they are designed to support various functionality in addition to simply improving application storage performance. Second, they employ the principle of composability to enable higher-level functionality to be developed from relatively simpler building blocks. The three general programmable data services that we consider here are BESPOKV [114], Faodel [7], and Mochi [6]. A fourth, composable storage service, Malocology [115] employs the principle of composition to decompose Ceph [116] to make it more programmable [6]. However, Malocology operates more like a storage service than a data service and is not considered here. Specifically, it targets the composition of the storage stack that is typically out of the end-users control and within the purview of a system administrator. As a result, Malocology can not be used to build transient user-level services.

BESPOKV [114] is a high-performance distributed KV store. By recognizing the growing importance of KV stores in HPC for coupling, analysis, and visualization purposes, BESPOKV introduces a flexible design for a distributed service based on the decoupling of the data plane and the control plane. The fundamental unit of the control plane is referred to as a *controlet.* The control plane receives client requests and forwards them to one of the distributed *datalets* in the system. Each user-supplied datalet implements a standard KV store API and manages a customizable "backend". Further, the user has complete control over the number and topology of datalets and controlets in the system, thus making the BESPOKV service customizable, flexible, and extensible. However, the evaluation of the BESPOKV system was performed on a virtualized cloud-based system. Thus, its performance when coupled with HPC applications is unknown.

Faodel [7] is a composable data service that aims to serve the general data storage and analysis needs of in-situ workflow components. There are three Faodel components — Kelpie, a distributed KV blob store, OpBox, a library offering primitives for distributed communication patterns, and Lunasa, a memory-management library for network operations. Faodel is intended to be a sink for data from bulk-synchronous applications, asynchronous many-task runtimes, and other

ML-based services running inside the workflow. At the same time, Faodel also acts as a source for in-situ visualization and analysis (ISAV) tools that run either within the workflow node allocation or remotely.

The Mochi project [6] arguably represents the largest-scale effort to build customizable, high-performance data services. The fundamental premise behind this effort is the observation that each member of an ever-broadening set of HPC applications has unique data storage requirements and access patterns. Thus, a one-size-fits-all approach is not a good strategy for developing data services. Instead, the Mochi project relies on the composition of microservice building blocks to enable the rapid development of higher-level functionality and customized data services.

The term "microservice" is a concept that originated in the cloud computing industry and is defined by Fowler [117] to be a "building block that implements a set of specific, cohesive, minimal functionality and can be updated and scaled independently". The works by Dragoni et al.[58] and Zimmermann [118] elucidate the various tenets surrounding the development of microservices in the cloud industry. Essentially, a microservice represents an end of the spectrum of distributed services and encapsulates minimal functionality (separation of concerns). The explosion in the number of cloud-computing services such as Amazon, Netflix, and Facebook that have adopted this architecture has given rise to the debate regarding whether microservices represent an "evolutionary" or "revolutionary" step in distributed software development. Jamshidi et al.[119] present both sides of this argument and conclude that the consensus among industry experts is that microservices are "SOA done right", i.e., they are an evolutionary trend in distributed service architectures.

The Mochi framework offers a set of microservices such as the SDS KV store (SDSKV), the BAKE object store, the REMI resource migration service, the

SONATA document store (to name a few). An HPC application can compose these microservices in any manner to serve its custom needs. Mochi depends on the Mercury [120] RPC library for communication and the Argobots [121] library for managing concurrency on the server. Notably, Mochi is a multi-institution effort involving five primary organizations contributing to the software's core development. Several other organizations and research teams across national research laboratories in the US have utilized the Mochi framework to develop a wide range of data services. Examples of these data services include the HEPnOS [122] data store for high-energy physics applications, the FlameStore [6] service for storing the results of ML-trained models in a distributed manner, the UnifyFS [113] user-level distributed file system, and the Mobject [6] object-store.

Table 4 and Table 5 present a comparison of the different HPC data services based on their composition model and coupling strategies. HPC data services fall under a category of components that are composed with HPC applications in a hybrid manner. This categorization is due to the flexibility offered by data services. For example, applications could use the data service purely for storing some partial results during execution. In this first case, data services are weakly-coupled with the HPC application. However, the application could also be simultaneously reading and writing from the data service. In other words, the application depends on the stored results in order to proceed with its computation. In this second case, the data services are strongly-coupled with the HPC application.

The fundamental unit of composition within the Mochi framework is a microservice, while Faodel offers three fixed components. BESPOKV offers customizable units called "controlets" and "datalets" that bear some resemblance to Mochi microservices in their design. Both Mochi and BESPOKV support multiple

Table 5. Distributed HPC Frameworks: Coupling Strategies and Communication Protocols

Framework Name	**Basic Computation Unit**	**Coupling Strategy**	**Internal Communication Protocol**
DCA	CCA component	PRMI	MPI
SCIRun2	CCA component	PRMI	MPI
Mochi	Microservice	RPC	Mercury RPC
Faodel	OpBox, Lunasa, Kelpie	RPC	Shared-memory
BESPOKV	Datalet, Controlet	N/A	Shared-memory
ParaView Catalyst	MPI process	Shared-memory	MPI
VisIt Libsim	MPI process	Shared-memory	MPI
SENSEI	MPI process	Shared-memory	MPI/ADIOS
TINS	Task	Shared-memory	MPI+Intel TBB
Henson	Henson Puppet	Shared-memory	MPI
Ascent	MPI process	Shared-memory	MPI/ADIOS
Damaris-viz	MPI process	Shared-memory	MPI
Seer	MPI process/Microservice	Shared-memory/RPC	RPC
Swift/T	Turbine Task	Dataflow	MPI
RADICAL-PILOT	Compute Unit	Task DAG	ZeroMQ
Merlin	Celery Worker	Task DAG	RabbitMQ

database backends, while Faodel doesn't appear to do so. Mochi is unique as it recursively uses RPC to compose operations internal to the service and the operations exposed for external application use. BESPOKV and Faodel employ a relatively flat structure and use shared-memory for internal communication within data service components. Mochi is also unique in the sense that it is the only framework that offers more than just key-value store capabilities. Mochi microservices span a more comprehensive range of functionality and thus are more broadly applicable.

2.6.3.3 In-situ Visualization and Analysis. Over the past decade, the flourishing research within the in-situ visualization and analysis (ISAV) research community has resulted in several loose definitions for the term "in-situ". To reduce the confusion over the use of this term, the ISAV community got together to define and categorize ISAV tools under six unique axes. Thus, unless specified otherwise, any reference to the methodologies surrounding ISAV tools in this section follows the definitions laid out by the community[123].

In-situ refers to the processing of analyzing the simulation data as it is being generated. This type of analysis is distinctly different from data processing after it has been written out to a storage medium. The motivation to perform in-situ analysis primarily arises from the inability of traditional HPC I/O systems to absorb the large volumes of data being generated by HPC applications [124]. Specifically, computation capabilities are growing faster than the storage I/O bandwidth. As a result, it is simply infeasible to write the entire simulation data to long-term storage for offline analysis.

In-situ methods enable the online, parallel processing of large amounts of simulation data to result in significantly smaller volumes of "interesting" simulation features written to disk. All the ISAV tools considered here have either (1) on-node proximity or (2) off-node proximity. This study does not consider an in-depth study of the third variety of ISAV tools that run on "distinct computing resources", *except* under the circumstance that this distinct computing resource happens to be a remote monitoring client (human-in-the-loop interaction). Specifically, the tools that utilize distributed computing resources spanning multiple geographical sites to perform computation fall under the category of grid computing systems, and thus, they are not a primary focus of our study.

As depicted by Table 4, most ISAV frameworks are composed along with the application in a weakly-coupled manner. In other words, the application is the producer, and the ISAV tool is the consumer. The application does not depend on the result of the analysis to proceed with its computation. The general assumption made by ISAV tools is that the analysis to be run is pre-determined (automatic, adaptive, or non-adaptive). One exception to this rule is the Seer [125] in-situ steering framework. In Seer, the simulation takes input from a human-in-the-loop in a non-blocking way, i.e., there is a back-and-forth interaction between the simulation, ISAV tool, and the human user. Thus, Seer is composed along with the application in a hybrid manner. Notably, Seer uses the Mochi [6] data service as an intermediate communication module to enable this interaction.

Concerning the coupling strategy (a combination of proximity, access, and integration type), Paraview Catalyst [126], VisIt Libsim [127], SENSEI [128], TINS, [129], Ascent [130], HENSON [131], Damaris-viz [132], and Seer [125] primarily interact with the application through shared memory. Except for Ascent, they all implement a dedicated API. Ascent supports multiple backends and hence implements a multi-purpose API. Typically, ISAV tool integration with an MPI application occurs through the use of a client library. The simulation invokes the ISAV routine locally on each MPI process, and the ISAV tool client converts the simulation data into a format suitable for analysis and visualization. Some tools such as SENSEI support the off-node transfer of this ISAV data to other components running within the in-situ workflow. Further, VisIt and ParaView support remote visualization of this data. Except for Damaris-viz and SENSEI, all the ISAV tools considered here support only time division between the application and the ISAV routines. Damaris-viz is implemented using the Damaris [133] I/O framework that allocates a dedicated compute core for the execution of ISAV routines. Damaris-viz communicates with the simulation through shared-memory belonging to the operating system, and thus, it can support space division and time division.

2.6.3.4 HPC Ensemble Frameworks. Among the many changes in the HPC landscape over the past few years, an important one is the emergence of a new class of scientific workloads referred to as *HPC ensembles.* Traditional workloads on HPC clusters are characterized by a small number of large, long-running jobs.

The push towards uncertainty quantification (UQ) [134] has resulted in ensemble workloads that consist of a large number of small, short-running jobs [135]. These ensembles are also referred to as “in-situ workflows” [104]. An analysis of the batch job submissions on Lawrence Livermore National Laboratory (LLNL)’s Sierra machine reveals that 48.5% of all submitted jobs reveal a pattern that typifies ensembles [135].

Individual jobs or tasks within an ensemble can be fully decoupled (such as the UQ pipeline [134]), or their coupling can be represented by a directed acyclic graph (DAG) or dataflow graph. The latter form represents the more general case. Notably, it is not uncommon for these tasks to represent a collection of different executables. There are a few fundamental capabilities that ensemble frameworks must provide — an ensemble programming system that includes a way to specify task dependence, support for inter-task communication, and hardware resource management (scheduling).

Swift/T [136] is a programming language and runtime for in-situ ensemble workflows. Swift is the scripting language used to specify the composition of workflow tasks, and Turbine [137] is the runtime used to manage the execution of tasks on an HPC cluster. Swift is a naturally concurrent language that uses a dataflow graph to infer dependencies between tasks, each of which can, in turn, be an MPI program itself. Swift is a scripting language that can natively invoke code written in C, C++, or Fortran and scripts in Python or Tcl.

Notably, the dataflow specification is not a static graph but dynamically discovered as the program executes. Internally, the Swift/T program is converted into an MPI program that runs multiple copies of the Turbine runtime on a machine allocation. These Turbine instances (MPI processes) manage the execution of Swift/T tasks by balancing the load between the available resources. When the Swift/T task is itself a parallel MPI program, Turbine creates a separate MPI communicator group to represent the MPI program. MPI is also used by Swift/T tasks to communicate with one another. Swift/T and UQ Pipeline [134] share the distinction of being single-cluster ensemble frameworks. These frameworks cannot be used to run tasks across multiple HPC clusters. At the same time, by bootstrapping on top of MPI, they do not need to deal with the complexities of interacting with a job scheduler and working around the security limitations posed by traditional HPC batch systems.

Merlin [138] and RADICAL-PILOT (RP) [139] are ensemble workflow frameworks that blur the line between HPC and grid computing. While Merlin is a workflow framework tailored for HPC ensembles that result in data being used for training ML models, RP is a general-purpose workflow framework applicable to any system. These two frameworks share several common design elements. One, they both support task execution across multiple HPC clusters. Two, task dependence is inferred through an internal task DAG. Three, both these frameworks employ a script-based programming "frontend". Four, there is an *external* centralized service or node that hosts the task queue. The workers that are launched on compute nodes within the batch job allocation (Celery tasks in Merlin, Agents in RP) pull from this external task queue in what resembles a producer-consumer model. Five, there is an entity that manages the worker instances within a batch job allocation and performs the work of a scheduler. This entity is referred to as a "Pilot" in the RP framework and is the Flux [135] component within Merlin.

However, there are some differences between these two frameworks. One, while Merlin supports inter-task communication through a data management library called Conduit [140], RP appears to use the file system to perform this action. Two, while Merlin uses RabbitMQ for internal communication between its components, RP uses the ZeroMQ [141] messaging platform. While there exist mature, general-purpose, distributed workflow management systems such as Pegasus [142], they are generally not applicable for ensemble HPC workflows. Three reasons are provided by Peterson et al. [138] to support this claim: (1) they often have a considerable upfront user training cost, (2) they do not support accelerators such as GPUs and FPGAs, and (3) they do not work well under the security constraints imposed by HPC data centers.

2.6.4 Resource Allocation and Elasticity. When two or more components are coupled together, one of the most fundamental questions that need to be addressed is how they share resources. *Resource allocation* refers to the methodology by which computing resources are divided among a set of simultaneously executing components. A related problem is the ability to change an existing resource allocation scheme dynamically. *Resource elasticity* refers to the ability of a framework to dynamically shrink or expand the number of resources being utilized in response to an internal change in application requirements or external factors such as performance variability and power constraints.

2.6.4.1 Resource Allocation. Table 6 lists the resource allocation schemes for the set of distributed HPC frameworks introduced in Section 2.6.3. Broadly, the common resource allocation schemes can be divided into five categories. Starting from the schemes that involve the highest degree of resource sharing to the ones that involve the lowest degree of resource sharing, these categories are:

1. Local node, same process: The framework and the "application" that utilizes the framework live in the same address space, interact through regular function calls and share the same computing resources. They may or may not share processing threads.

2. Local node, separate processing core: The coupled components live on the same computing node and share the computing resources. However, they do not live in the same address space, and thus they do not share processing threads.

3. Distinct nodes, same machine: The coupled components simultaneously execute on distinct computing nodes within the batch job allocation. The only resources they share are network resources (switches and routers) and the parallel I/O filesystem.

4. Hybrid: In a hybrid resource allocation scheme, there is significant flexibility in how the resources are divided up among the coupled components. Specifically, any one of the schemes (1), (2), or (3) can be employed.

5. Distinct nodes, different machines: The coupled components share a minimal amount of resources. They can run across multiple HPC machines and communicate through a centralized messaging or communication framework.

Most distributed CCA component frameworks such as DCA [1] and SCIRun2 [84] employ a resource allocation scheme in which the individual components are executed on distinct computing nodes within the same machine. Typically, these components are deployed as independent MPI programs within the same batch job allocation. Faodel [7] and BESPOKV [114] also employ this same type of resource allocation strategy. Within the class of frameworks referred to as "data services", the Mochi [6] infrastructure is unique because it offers a hybrid resource allocation scheme. Mochi microservices can be configured to run inside the same process as the "client" (MPI simulation), on different processes running on the same node as the client, or distinct computing nodes within the batch job allocation. Notably, the Mercury RPC

Table 6. Distributed HPC Frameworks: Resource Allocation Scheme

Framework	**Resource Allocation Scheme**
DCA	Distinct nodes, same machine
SCIRun2	Distinct nodes, same machine
Mochi	Hybrid
Faodel	Distinct nodes, same machine
BESPOKV	Distinct nodes, same machine
ParaView Catalyst	Local node, same process
VisIt Libsim	Local node, same process
SENSEI	Hybrid
Ascent	Local node, same process
TINS	Local node, separate processing core
Henson	Local node, same process
Damaris-viz	Local node, separate processing core
Seer	Hybrid
Swift/T	Distinct nodes, same machine
RADICAL-PILOT	Distinct nodes, different machines
Merlin	Distinct nodes, different machines

framework [120] employed by Mochi offers an RPC API that abstracts the specific resource allocation scheme in use.

Typically, ISAV tools are tightly integrated with the MPI application and employ a time-division or space division scheme to share data within the application. Their proximity to the application ensures that the data transfer overheads are minimized. ParaView [126] VisIt [127], and Ascent [130] employ the resource allocation scheme of type (1) and run inside the same process as the MPI application. However, they support an external remote-monitoring client (such as a Jupyter notebook) to monitor the results of the in-situ analysis.

SENSEI [128] functions as a generic bridge between an HPC application and several in-situ implementations such as ParaView and VisIt. Additionally, SENSEI can be coupled to an external component running on the system through ADIOS. Thus, SENSEI implements a hybrid resource allocation scheme. TINS is built upon IntelTBB [42], and the analysis routines are launched within separate threads sharing the local computing resources with the application. Among ISAV tools, Damaris-viz [132] and TINS [129] are unique because they run within a dedicated processing core on each computing node. Damaris-viz is itself an MPI application that is launched beside the simulation. Each Damaris-viz MPI process communicates *only* with the MPI processes belonging to the simulation that runs on the same computing node. Seer [125] employs a hybrid scheme wherein the simulation is coupled with an external Mochi SDSKV service. Seer offers the user the flexibility to determine the exact resource allocation for the Mochi service.

Swift/T [136] tasks are scheduled onto processes belonging to a single MPI application. Here, we consider the resource allocation scheme employed at the task level. Since multiple tasks can run simultaneously on distinct nodes (where each task is itself an MPI program), Swift/T employs a scheme of type (3). In contrast, Merlin [138] and RP [139] are unique as their tasks can span multiple HPC machines. These two ensemble frameworks employ a resource allocation scheme of type (5).

2.6.4.2 Resource Elasticity. Resource elasticity is the ability to *dynamically* expand or shrink the number of resources being utilized *in response* to external stimuli or a change in application requirements. Note that resource elasticity is just one method by which a system can adapt itself, and it does not have the same meaning as dynamic adaptivity. Table 7 lists the current support for elasticity within different distributed HPC frameworks.

Most distributed HPC frameworks do not natively support resource elasticity. As pointed out by Dorier et al. [143], one of the factors is the dependence on MPI as a bootstrapping mechanism. Although the MPI standard has provisions to support the dynamic addition of new processes, most widely used MPI implementations do not support this feature [144]. The ones that do support elasticity (such as Adaptive MPI [145]) require a significant modifications to the application code. Supporting elasticity within a traditional HPC cluster requires changes to the scheduler and cost model as well. Specifically, *over-provisioning* is not a feature supported by most existing HPC cluster schedulers. Motivated by the pay-as-you-go cost model and the elasticity supported on cloud platforms, previous efforts [144, 146] explore the elastic execution of MPI programs within the cloud. Typically, the techniques implemented by these efforts require some form of checkpoint-restart combined with a monitoring and decision-making framework.

However, none of these efforts have successfully demonstrated the elastic execution of MPI programs within the context of a traditional HPC cluster. Therefore, it is safe to say that currently, MPI programs do not support elasticity. ISAV tools such as ParaView and VisIt run within the context of an MPI process and employ time-division coupling to share computing resources. Therefore, any ISAV tool that depends on MPI is limited in its support for elasticity. TINS is an ISAV tool wherein the analysis routines run within a separate TBB thread associated with a dedicated "helper core". When analytics routines are available to run, this core is used exclusively to execute the analytics routines to prevent interference with the simulation. Otherwise, the helper core is utilized as a common core for processing simulation tasks.

The only class of frameworks that support elasticity are data services. Specifically, BESPOKE supports "scale-out" resource elasticity through the dynamic addition and removal of its core components — controlets and datalets. The support for elasticity has recently been added to the Mochi framework. Specifically, the BEDROCK microservice functions as a bootstrapping mechanism through which other microservice instances can be dynamically instantiated. Note that although these frameworks support resource elasticity in some form, none provide the ability to do so *automatically*.

Further, none of the ensemble frameworks surveyed here support resource elasticity. Each of these frameworks requires the user to either specify a *fixed* number of MPI tasks (Swift/T) or a fixed batch allocation size for a given set of tasks (Merlin and RP). Once these tasks are mapped onto the computing resources, there is no way for them to request more (or less) resources dynamically should the need arise. Note, however, that the *resource utilization* levels within a batch job allocation can naturally wax and wane depending on the particular sequence of task execution. Arguably, this is not the same as the ability of a framework to enable resource elasticity.

Table 7. Distributed HPC Frameworks: Resource Elasticity

Framework	**Supports Elasticity?**	**Unit of Elasticity**
DCA	No	N/A
SCIRun2	No	N/A
Mochi	Yes	Microservice
Faodel	No	N/A
BESPOKV	Yes	Controlet, Datalet
ParaView Catalyst	No	N/A
VisIt Libsim	No	N/A
SENSEI	No	N/A
Ascent	No	N/A
TINS	Yes	Task
Henson	No	N/A
Damaris-viz	No	N/A
Seer	Partially	Microservice
Swift/T	No	N/A
RADICAL-PILOT	No	N/A
Merlin	No	N/A

2.6.5 Data Management Strategy.

A fundamental question that arises when coupling two or more distributed HPC frameworks or applications is how to transfer and stage data between them efficiently. Within the context of our study, HPC data management frameworks are differentiated from more general HPC data services. From a functional standpoint, HPC data management frameworks exist solely to transfer data between coupled components, while data services offer a more broad set of capabilities.

Table 8. MxN Coupling Frameworks

Framework	Conceptual Technique	General Framework?	Data Redistribution	Communication Schedule Calculation	Data Transfer
DDB	Component-based	Yes	MxN	Centralized	Parallel
CUMULVS	Component-based	No	Mx1	Centralized	Parallel
Seine	Component-based	Yes	MxN	Centralized	Parallel
MCT	Component-based	No	MxN	Distributed	Coupler
PAWS	Component-based	Yes	MxN	Centralized	Parallel
InterComm	Component-based	Yes	MxN	Distributed	Parallel
DCA	PRMI	Yes	MxN	Distributed	Parallel
SCIRun2	PRMI	Yes	MxN	Distributed	Parallel

2.6.5.1 MxN Problem.

Multi-physics applications launched as two or more strongly-coupled, separate MPI programs need some way to communicate and exchange data with each other. The general problem of redistributing data from an application launched with M processes to an application launched with N processes came to be known as the *MxN* problem [107]. The MxN problem was recognized as a major research area within the general field of distributed CCA framework design. As elucidated by Zhao and Jarvis [147], the MxN communication typically involves the following steps:

- Data translation: Data stored in one format (for example, row-major form) may need to be translated into another form (column-major form).
- Data redistribution: The sender and the receiver must be aware of the exact set of elements they expect to communicate with each other.

– Computing a communication schedule: Once the set of elements to be sent are identified, each sender needs to identify the portions of these data elements to be sent to specific receivers and accordingly compute a communication schedule.

– Data transfer: After the communication schedule has been computed, the last step involves the actual data transfer itself.

Table 8 uses these steps to list out and differentiate a set of popular MxN frameworks. Importantly, MxN frameworks use one of two methodologies to enable MxN data redistribution — PRMI or a component-based implementation. Distributed CCA frameworks such as DCA [1] and SCIrun2 [84] employ the PRMI model to perform complete MxN data redistribution. DCA is built upon MPI and allows a subset of components on the sender side to redistribute data. However, it requires all receiving components to take part in the data redistribution process. SCIrun2 allows two forms of data redistribution — collective and point-to-point. In collective data redistribution, all the components across the sender and receiver side must necessarily be involved in the communication. All the other frameworks presented here — DDB [148], CUMULVS [149], Seine [150], MCT [151], PAWS [106], and InterComm [152] employ a separate CCA component to perform the data redistribution.

Among the component-based frameworks, all of them are generally applicable to any distributed CCA application except for CUMULVS and MCT. CUMULVS is designed as a distributed component that allows a human user to visualize, interact with, and steer the MPI-based simulation as it is running. Specifically, this visualization component is implemented as a serial application that generates a set of requests to "pull in" the necessary portions of the domain (multi-dimensional array) from multiple MPI processes while they are running. As a result, CUMULVS only supports an Mx1 data redistribution scheme. Further, the communication schedule is calculated in a centralized manner within the serial visualization application. The MPI processes transfer their portions of the requested domain in a parallel fashion. On the other hand, the MCT component is explicitly designed to work with earth science applications such as CESM [52]. Although MCT supports an MxN data distribution scheme and the calculation of communication schedules locally within every sender process, the data transfer mechanism is unique. Each of the M sender processes routes their individual data elements through a “coupler” module that performs the data redistribution and forwards them to N receiver processes.

DDB, InterComm, Seine, and PAWS are general-purpose component-based MxN frameworks that support full MxN data distribution. Among these, InterComm is unique concerning the methodology by which the communication schedule is calculated. Specifically, InterComm identifies a set of “responsible” processes that compute the communication schedule after gathering the source and destination data structure representations. Once this task is complete, these “responsible” processes transfer the schedules to the processes that need them. Finally, each sender process proceeds to transfer the data in a parallel manner.

On the other hand, DDB, Seine, and PAWS employ a dedicated centralized component to calculate the communication schedule. DDB employs a two-level scheme where each coupled application nominates a control process (CP) to communicate domain information and data layout to a single registration broker (RB) during the registration phase. Once the RB receives all the information from each CP, the RB performs the matching between data producers and consumers and calculates the communication schedule. This communication schedule is broadcast to all other processes when they finish their registration phase.

PAWS and Seine are similar because they introduce a distinct distributed component to orchestrate the data redistribution. Both these frameworks introduce the notion of a “shared virtual space” used to inform the communication schedule. Notably, the calculation of the communication schedule is performed during the registration phase, making this phase the most expensive routine in terms of execution time. The notion of a shared virtual space and the introduction of a dedicated central component to perform data redistribution allows these frameworks to support process dynamism elegantly. New processes register themselves and their portion of the domain (multi-dimensional array) with the Seine framework, after which they can seamlessly take part in the MxN communication without being aware of who the actual senders are.

Table 9. Data Staging and I/O Frameworks

Framework Name	**Data Staging Nodes?**	**Asynchronous Data Transfers?**	**Allows Process Dynamism?**
ADIOS	Flexible	Yes	Yes
DataSpaces	Yes	Yes	Yes
FlexPath	No	Yes	Yes
GLEAN	Yes	Yes	Yes
Decaf	Yes	No	No
FlowVR	No	Yes	No
Damaris	No	Yes	No

2.6.5.2 Data Staging and I/O Frameworks. While MxN frameworks address a critical problem that arises when coupling two MPI applications, it is not difficult to see that the class of applications (and hence the coupling methodologies) they support are limited. Specifically, most, if not all MxN frameworks assume that the data needs to be transferred (1) immediately, (2) synchronously, and (3) *only* between two MPI applications. In other words, MxN frameworks are generally associated with strongly-coupled applications.

The advent of transient, in-situ visualization and analysis tasks together with the growing disparity between computing and parallel storage performance has given rise to a class of frameworks that offer *general* data staging and I/O management capabilities. Specifically, these data staging frameworks offer the ability to transfer and stage data asynchronously and between more than two coupled components. Moreover, they allow significant flexibility in describing the data to be staged and potentially also support process dynamism, i.e., the ability to continue operating when processes enter or leave the system.

ADIOS [5] is arguably the most widely supported data staging and I/O framework for HPC applications. ADIOS originally started as an API that offered seamless asynchronous I/O capabilities. The growing disparity between compute and storage performance meant that applications could no longer afford to synchronously write massive amounts of data to disk without severely damaging their overall performance. ADIOS offered a way out of this problem by staging the data locally and performing the write operation only during the application's compute phase. The ADIOS API was initially designed to be POSIX-like while decoupling the *action* of performing a parallel write with *when* and *where* the data is written out. Since its initial release, ADIOS has become synonymous with an I/O API that offers various data staging and in-situ analysis capabilities. Specifically, ADIOS employs several backend "transport methods" that effectively determine the sink for the data. Examples of transport methods include specialized data staging software such as DataSpaces [8] and FlexPath [153]. ADIOS' modular design has ensured widespread adoption as a "high-level" I/O API across various classes of HPC applications.

GLEAN [154] is a data staging and in-situ analysis library that offers asynchronous data staging and offloading capabilities. GLEAN can be leveraged either directly through its API or by the transparent library interposition of HDF5 routines. Note that the latter technique has the benefit of not requiring any application code changes. Data from GLEAN-instrumented applications is asynchronously transferred to a dedicated set of staging nodes, effectively functioning as an in-memory "burst-buffer". After the data is transferred to the staging nodes, the data is available to other components such as in-situ analysis tools. As such, GLEAN does not offer any special features to transform application data into different formats. Because of the passive, decoupled way in which it operates, GLEAN is indifferent to process dynamism within the source application.

DataSpaces [8] is a project that leverages the "shared virtual space" concept first introduced by the Seine [150] MxN coupling framework. DataSpaces builds upon the multi-dimensional data representation and linearization scheme in Seine and offers data coupling via a separate *dataspaces* distributed component. This MPI-based dataspaces component runs on a dedicated set of computing nodes that asynchronously stage data from multiple coupled applications. Further, the dataspaces component offers an API that allows in-situ analysis to be performed and an API to install a monitor that checks for updates on a region of interest. A benefit of having a separate data staging component is the implicit ability to support process dynamism.

The FlexPath [153] system offers a typed publish-subscribe mechanism for connecting data producers with data consumers within a coupled HPC application. The data producers effectively define and generate a data stream to be consumed by any distributed component running on the system or remotely. Notably, FlexPath employs a direct-connect scheme to transfer data objects directly between a publisher and subscriber. This scheme is different from a traditional brokered architecture employed by other data staging software such as DataSpaces. A direct-connect design choice implies that data is staged locally within every publisher process. FlexPath hooks into application I/O routines through the ADIOS I/O API. FlexPath utilizes the EVPath [155] communication substrate to transfer data between different components. Every new process entering the system communicates the data objects of interest to a local message coordinator that, in turn, calculates the publishers from which it must fetch data. This decoupled approach enables FlexPath to support process dynamism.

FlowVR [156] and Decaf [9] represent data staging software that depend on the concept of *dataflows* to transfer data between coupled components ("nodes"). Decaf is a data staging software that depends on MPI. Specifically, the Decaf system takes as input a JSON file representing the coupled applications and splits the global MPI communicator among the various "nodes" (coupled MPI applications) and "links" (data staging processes). Note that a link separates two nodes. The producer node transfers data to the link, and the link can optionally transform the data or perform specialized analysis on the data before forwarding it to the consumer node. In FlowVR, however, the data transfer is performed in the background by a FlowVR daemon process on every computing node, with limited support to stage data. However, FlowVR data in transit can be acted upon by a set of pre-defined "filters" to transform it before it is passed on to the next component in the flow. Both FlowVR and Decaf need to be informed of the task graph before execution, and as a result, they cannot handle process dynamism.

Damaris [133] is an I/O management framework that relies on a dedicated processing core on each computing node to perform asynchronous I/O. The global MPI communicator of the application is split into two — one for the main application itself and the other for the Damaris component. Processes within a computing node asynchronously communicate their I/O data with the local Damaris process through shared memory. The Damaris process optionally hosts a set of plugins that act upon this data to compress, analyze, and finally commit it to a long-term storage medium. Due to its reliance on MPI, Damaris is limited in serving as a general distributed data staging software. Instead, it can serve as a data source for in-situ analysis tools such as Damaris-viz [132].

2.6.6 Performance Tools. As the number of simultaneously executing components within a distributed, in-situ workflow continues to rise along with the scale of the HPC machine, the application of performance tools to ensure the proper and optimal operation of the workflow is growing in importance as well. Traditionally, HPC performance tools are employed for the *offline* performance analysis of monolithic MPI applications. State-of-the-art performance tools such as Score-P [157], TAU [29], CALIPER [31], and HPCToolkit [30] collect a rich profile and trace that is ultimately written out to disk for offline performance analysis.

With the advent of coupled applications and in-situ workflows, a different approach is needed for practical performance analysis at scale. Specifically, several challenges must be addressed within each of the three classical performance engineering categories — performance measurement, performance monitoring and analysis, and performance control and adaptivity. This section considers these challenges in detail. Table 10 summarizes the level of tool support currently available within each of these categories for the different types of coupled applications and in-situ workflows previously introduced.

2.6.6.1 Performance Measurement. Performance instrumentation, measurement, and sampling often represent the first steps in the performance engineering of an application. The tool API instrumentation is added either explicitly (source instrumentation) or implicitly (library interposition). Measurements are then made when the application executes. Traditionally, these measurements are gathered and written out to disk when the application finishes executing.

An important observation to be made about in-situ workflows is that many depend on one or more MPI-based components. Specifically, this is the case for commonly used strongly-coupled MPI applications (XGC-GENE and LAMMPS) and ISAV tools such as ParaView [126], SENSEI [128], and Ascent [130]. In such a situation, the rich support for measurement in existing HPC performance tools can be leveraged and extended as needed. Specifically, there are two *types* of measurements to be made. One, the measurements that represent internal function execution times and metrics for each coupled component. Second, the measurements that correspond to the interactions between the components. Wolf et al. [158] identify key ADIOS [5] routines to instrument for capturing data movement between coupled applications. Given its widespread use, instrumenting high-level ADIOS routines automatically enables insight into any transport method utilized underneath.

Malony et al. [159] demonstrate a methodology by which Ascent routines are instrumented and analyzed using TAU's plugin architecture [160]. ISAV tools are typically tightly integrated with the MPI application, and the ISAV tool routines are invoked synchronously by each MPI process. This design presents an opportunity for TAU plugins to "hook into" these synchronous executions to dynamically calculate the contributions of the Ascent routines to the captured performance events. A key observation here is that plugin architecture offers a doorway to both the performance event data and the tool measurement API.

Traditional HPC performance analysis tools built for MPI-based applications cannot be generally applied to gather performance measurements from HPC data services such as Mochi [6] and BESPOKV [114]. HPC performance tools implicitly assume that control is not passed between two different distributed applications. Data services break this assumption through an RPC-based client-server communication model. Thus, the HPC community needs to look to the general cloud computing industry for answers to measuring data service performance. Sambasivan et al. [161] summarize the extensive body of research on a class of distributed tracing tools that implement request metadata propagation. Briefly, this technique involves generating a unique "request ID" and the subsequent propagation of this request ID through the system by RPC invocations. The request ID is then used to tie together events that are *causally related* and thus, this technique can be used to capture distributed callpaths, request structures, and also be used to compare request flows [162]. Industry tracing tools such as Dapper [14] and Jaeger [163] employ request metadata propagation on production-scale cloud computing systems.

Performance measurement of HPC ensembles is an open area that is yet to be targeted by the HPC tools community. Table 10 enlists the current level of performance measurement support for HPC ensembles as "partial". While traditional measurement tools can be used to capture the execution time of individual ensemble tasks that happen to be MPI-based applications (or serial tasks), there is no existing tool to provide a holistic picture of the task execution in conjunction with the dynamic task interactions and resource utilization measurements. An integrated approach is required to capture and correlate all three types of performance measurements.

2.6.6.2 Performance Monitoring and Analysis. Arguably, the bulk of performance solutions for coupled in-situ workflows fall into the category of

Table 10. Performance Tools for Coupled Applications and Workflows

Application Type	**Measurement Tools Exist?**	**Monitoring & Analysis Tools Exist?**	**Control & Adaptivity Tools Exist?**
Strongly-coupled MPI	Yes	Yes	Partial
ISAV Tools	Yes	Yes	Partial
Data Services	No	Partial	Partial
Ensembles	Partial	Partial	No

monitoring and analysis tools. Partly, this is due to the observation that several existing performance measurement tools can be leveraged directly for *most* components within the coupled in-situ workflow. Therefore, the existing research focuses on monitoring and exporting this data to an external entity for aggregation and analysis.

WOWMON [164] is a monitoring and analysis infrastructure for in-situ workflows. WOWMON instruments coupled applications using traditional HPC performance tools to generate performance measurements. These performance measurements are buffered and sent over EVPath [155] to a central workflow manager. The performance data is analyzed to gather the end-to-end latency of the workflow. Further, this performance data is passed through a machine learning profiler to rank the instrumented metrics according to their correlation with the end-to-end latency of the workflow.

SOS [165] is a distributed monitoring tool that offers the ability to collect, aggregate, and analyze performance data from multiple simultaneously executing coupled applications. The SOS client interfaces with the application to collect performance data which it then forwards to a collector daemon running on the same computing node. The daemon processes are organized into an overlay network that aggregates local performance data. The Lightweight Distributed Messaging System (LDMS) [166] and MRNet [167] tools also employ an overlay network to aggregate performance metrics within an HPC cluster. LDMS, together with Ganglia [168] represent a class of monitoring tools that can be employed for system resource monitoring. Unlike SOS, they do not offer a client instrumentation library, and thus, they can not be used to capture application performance data directly.

As the level of concurrency on modern HPC systems continues to rise, the volume of performance monitoring data produced can significantly perturb application performance [158, 101]. Thus, there is a growing interest in sub-sampling and analyzing performance data *in-situ* to reduce trace sizes before a global aggregation is carried out. The MOnitoring Analytics (MONA) [158, 57] approach speaks to this kind of a technique. Specifically, MONA employs the SOS [165] monitoring tool to aggregate and analyze TAU performance data from a coupled MPI application. The TAU performance data is piped to SOS through a TAU plugin. The aggregated data is analyzed and visualized on an interactive dashboard.

Chimbuko [169] is a workflow-level in-situ trace analysis tool. Chimbuko analyzes the performance data from a coupled application workflow to generate *performance anomalies*. Specifically, the TAU plugin infrastructure is utilized to export performance traces to a process-local anomaly detection (AD) module. The AD module periodically communicates with a central AD parameter server to update its internal anomaly thresholds based on a *global* view of statistical outlier information. Finally, when Chimbuko detects an anomaly, it captures and stores provenance information that helps identify the context in which the anomalous value was recorded.

While the design of distributed tools such as SOS can be generally applied to monitor HPC data services, the data model used to capture performance information must be carefully studied. HPC data services that run in highly concurrent environments handle thousands of requests per second. Thus, the instrumentation library must be able to operate efficiently under a high degree of concurrency. Not only this, the accompanying in-situ analysis needs to be able to track changes *across time* to be able to observe poor service performance. Thus, a time-series monitoring approach combined with sophisticated node-local analysis for trace data reduction may be a viable strategy. There are several state-of-the-art cloud-based tools such as Prometheus [19] and Graphite [170] that implement a time-series database. However, these tools operate within the constraints of a commodity hardware and stack, and thus, they need to be appropriately modified to suit HPC service requirements.

2.6.6.3 Control and Adaptivity. Several tools implement adaptive algorithms within the context of individual applications. The MPI_T interface is a notable effort to enable tool integration for control and adaptivity of MPI applications. Specifically, performance tools can use control variables (CVARs) to effect dynamic adaptation and control. The APEX [171] monitoring system exposes a set of *listeners* that external tools can use to implement control policies. The TINS [129] in-situ framework implements a naturally elastic threading model that enables the sharing of computing resources between simulation and analysis routines.

However, fewer tools enable control and adaptivity resulting from data analysis of a coupled execution. The MONA [158] project studies the cross-application interactions resulting from an XGC-GENE coupling to determine a more optimal task placement for both applications. However, this more optimal task placement cannot be implemented immediately. Instead, it is a valuable starting point for subsequent coupled executions. The Seer [125] in-situ steering framework enables a human user to interact with a running simulation to execute custom, dynamic in-situ analysis routines. Older monitoring tools such as Falcon [172] also enable user-interactive simulation steering of traditional monolithic executables. Pufferscale [173] is a multi-objective optimization framework that can simultaneously and dynamically balance load *and* data across a set of distributed Mochi [6] microservices. This re-balancing is enabled through the REMI resource migration microservice. However, Pufferscale cannot enable an online re-scaling (or resizing) of the number of distributed microservices.

2.7 Trends and Open Areas

This section describes the important trends and open areas that inform future work for HPC performance tools.

2.7.1 Trends. This section describes the major trends and open areas that inform future work for HPC performance tools.

There are several significant trends concerning the evolution of modern HPC applications. First, the number of distinct components coupled together has been steadily increasing over the past two decades. As distributed CCA frameworks became popular, strongly-coupled MPI applications were developed. ISAV tools and data management frameworks arrived on the scene, increasing the number of coupled, distributed components. HPC ensembles push the barrier even further, resulting in hundreds to thousands of small, short-running tasks.

Second, the types of applications that require high-performance capabilities have exploded. HPC platforms that were once strictly the domain of bulk-synchronous parallel applications now share the space with transient data analysis tools and ML tasks that were traditionally executed on desktop-class single-node machines. On the one hand, tools based on statistical analysis extract helpful knowledge from the large amounts of data generated by HPC applications, and their integration with traditional BSP-style codes requires careful thought. As a result, the HPC community has borrowed ideas and techniques from the general cloud-computing and artificial intelligence (AI) communities. On the other hand, some of these ML and AI tools are large-scale distributed applications in their own right. Their special needs are driving the decisions behind the procurement of these multi-million dollar HPC machines [56].

Third, the relatively slow growth of traditional file-storage performance on HPC machines compared to the computational performance is the single most significant hardware factor contributing to the emergence of several new classes of distributed frameworks described in this document. The resulting storage heterogeneity and the inclusion of faster, storage-class memories is only one part of the solution to this problem. The second part consists of the development of appropriate software abstractions such as data services and I/O frameworks. This latter part is particularly challenging to get right — the integration of these new services: (1) should ensure high performance of the resulting coupled application, (2) should present a unified interface that hides the complexity of programming the storage hardware, (3) should not hamper developer productivity, and (4) should not adversely affect the operation of existing legacy codes.

Fourth, performance tools are always the last to be updated. In other words, their evolution has always *followed* the applications they measure, instead of *co-evolving* with the application itself. For example, the CCA specification in its initial form did not appear to have any special provision for the design of CCA-capable performance tools. Instead, the tool community invented clever ways to integrate performance measurements through component proxies seamlessly. Aside from SCIRun2 [84] and Uintah [85], no other CCA framework considered performance optimization as a first-class design requirement. However, there are indications that this is changing.

The MPI and OpenMP communities have recognized the need to tightly integrate performance tools by including a “tools section” in their respective specifications.

A fifth, bold prediction can be made by comparing the evolution of HPC software architectures with the respective changes within the general computing industry. The industry has shifted from an ESB-style tightly-coupled model to a more loosely-coupled services model where each service is highly cohesive in terms of functionality and can be independently updated from other services or components. This paradigm shift has increased the scalability of the overall application and allowed for faster, dynamic updates to service functionality without hampering other distributed components. If the initial signs [108, 6] are anything to go by, HPC software architectures are likely to resemble their industry counterparts in the future.

2.7.2 Open Areas. The trends discussed in this section naturally point to some critical open areas for future tool development. Table 10 presents a brief overview of the level of tool support for the different classes of distributed HPC frameworks discussed in this document.

2.7.2.1 Performance Instrumentation & Measurement. As discussed in Section 2.6.6, relatively few techniques exist to instrument and measure HPC data service performance. Their key interactions cannot be captured using existing techniques such as compiler instrumentation or PMPI-based library interposition. There are two reasons for this. One, these services typically do not use MPI for communication. Instead, the data services considered in this study primarily rely on RPC for passing control between coupled components. Traditional HPC performance tools are not designed to handle a client-server architecture. Second, and more importantly, control is passed *between* two or more distributed components. In the case of a microservice architecture such as Mochi [6], RPC calls can span microservices running on different computing nodes. Tracking these microservice interactions ("callpaths") through the system requires HPC tools to borrow some ideas from the cloud-based performance tools. At the same time, these microservice callpaths need to be annotated with *context* to associate performance inefficiencies occurring at lower levels in the software stack with higher-level interactions. Thus, a careful application of a combination of ideas borrowed from decades of HPC tool research with novel distributed request-tracing techniques can be a practical approach.

Regarding performance measurement, HPC ensembles only partially succumb to the application of existing HPC performance tools. For example, applying traditional PMPI-based measurement techniques to a Swift/T [136] workflow execution can yield information about the data transfers between individual Swift/T tasks. However, little information can be gathered this way about the execution details of individual Swift/T tasks. Individual tasks need some way of exchanging their identity with the performance tool so that the tool's measurement infrastructure can separate the performance events belonging to the task from the performance events belonging to the underlying Turbine [137] runtime.

2.7.2.2 Performance Monitoring & Analysis. While several robust performance monitoring tools such as LDMS [166] exist, these tools primarily target the monitoring of hardware resources. Recently, tools such as SOS [165] have been developed to monitor and aggregate performance data simultaneously from multiple data sources. They can be broadly applied to monitor any distributed application. However, most existing monitoring tools collect the data, aggregate this data in a central location (database), and then optionally provide the ability for a user to analyze the data from within this central location.

While this technique can scale well when the volume of data aggregated is minimal, it may not work for newer types of applications such as data services. Specifically, HPC data services operate in a highly concurrent environment. Thus, they require monitoring, aggregation, and analysis of large volumes of event traces to detect performance inefficiencies. Given the large storage footprint of event tracing, there is a need to analyze data at the source, *before* the aggregation is performed. Monitoring tools need to be flexible enough to support this type of analysis.

2.7.2.3 Control & Adaptivity. Few tools exist that can dynamically control and guide the execution of a coupled application. Fewer (if any) tools offer the ability to do so automatically. While adaptive algorithms have been studied and developed for individual modules in isolation, the guided execution of a coupled application requires performance data to be captured from multiple sources, aggregated, and finally analyzed to result in a control decision.

Further, the question of *who* actuates the control mechanism is also essential. Currently, the power to make these control decisions rests with a human user [125, 158]. This solution may work well when the number of coupled modules is relatively small, and the timescales involved in the control loop are large. However, HPC ensembles and transient data services involve tens or hundreds of individual modules and tasks that complete in a short span. Thus, they may require an automatic control system that relies on predefined policies. Further, as depicted in Table 7, many existing frameworks need to be updated to support resource elasticity before they can reap the full benefits of a control infrastructure.

2.8 Summary

Chapter II presented a novel narrative of the evolution of HPC software development methodologies and the accompanying changes in HPC performance tools. Modularization was identified as the recurring theme underlying major revolutions in HPC software development. Over the past thirty years, HPC software has become increasingly modular due to the following broad factors:

1. **Simulation Scale and Fidelity**: The need to run simulations efficiently on ever-increasing node counts continues to impact the design choices for implementing HPC software.

2. **Number and Variety of Applications Requiring HPC**: The recent emergence of ML workloads requiring HPC capabilities has forced the community to consider alternatives to the MPI programming model to integrate them into traditional workloads.

3. **Hardware Trends**: Hardware trends, particularly the slow growth of parallel I/O bandwidth and the emergence of accelerators such as GPUs and FPGAs, have resulted in an update to how HPC software is built and deployed.

4. **Structure and Complexity of HPC Software Development**: Today's state-of-the-art HPC software is notably more complex and modular than three decades ago, reflecting the structure and the number of collaborating development teams.

This chapter categorized various emerging types of HPC frameworks and applications based on their composition model, resource allocation scheme, and data management strategies. A discussion of the various techniques HPC performance tools have implemented to stay relevant was also presented. Finally, this chapter touched upon some trends and open areas informing future work. Chapter III explores these open areas in the context of performance observation tools for HPC services.

CHAPTER III

IDENTIFYING WHAT TO OBSERVE AND MONITOR

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Argonne National Laboratory. The research work in this chapter was presented at IPDPS 2021 [13] and HiPC 2021 [16]. I was the first author of the IPDPS 2021 and HiPC 2021 papers. Dr. Philip Carns and Dr. Robert Ross were instrumental in helping me formulate the main research questions presented in this chapter. In both of these publications, I wrote all of the sections constituting the papers with suggestions and edits from Dr. Philip Carns and Dr. Robert Ross. All of the co-authors helped in proofreading. This chapter is formulated by gathering my contributions from these two publications.

3.1 Introduction

Chapter II presented the factors driving the trend toward increasingly modular HPC software and the open areas for tool development, particularly concerning tools for observation and analysis of in situ workflows. Chapter III narrows the focus of this dissertation to in situ workflows involving high performance distributed services. In particular, Chapter III focuses on breaking down the broader question of improving service performance into a set of concrete performance queries for a tool solution to address. This chapter also describes why existing HPC performance tools are primarily inapplicable. By addressing **Challenge 1**, Chapter III partially addresses the research question **RQ1**: How to enable performance observability of services built by composing high performance microservices?

Microservices [117] originated in the broader cloud computing community as a way to implement a particular form of distributed services architecture involving highly independent, specialized *components.* The definition of a component (used interchangeably with the term "module") in this context is "any piece of software that is independently upgradeable and replaceable". By this definition, every service is a component, but not every component is a service. In other words, libraries can also be used to implement component software, with the distinction that accesses to library components involve a local function call instead of a remote function call that traverses the process (and network) boundary. Componentizing software is a logical way to represent boundaries and a clean way to implement separation of concerns. Not only that, component software in the cloud industry was designed to reflect the structure of software development teams — independent units with well-defined functionality and interfaces to exchange information.

3.1.1 Benefits of Microservice Architectures.

Prior to the advent of microservices, cloud software and services were primarily deployed as so-called "monoliths" — large pieces of software with several components *tightly coupled* together either within a process or a framework abstraction such as an enterprise service bus that disallowed the independent scaling or modification of constituent components. Figure 5 depicts this difference between monolithic and microservice architectures. Specifically, microservices are attractive for developing distributed software for the following reasons:

- **Reuse**: Microservices promote a higher degree of reuse than traditional monolithic services. Designing microservices with highly specialized functionality can achieve significantly more code reuse between invoking modules.

- **Independent Scalability**: Perhaps the single biggest benefit of using microservices over monoliths is the ability of microservices to be scaled independently of one another. For example, consider a situation in Figure 5 wherein component B happens to be a bottleneck for a given client workload and needs to be scaled out. With a monolithic architecture, there is no option but to scale the *entire* application and create duplicate copies to resolve the bottleneck situation. With a microservice architecture, only microservice (component) B needs to be scaled out while the other components can remain as they are. In this manner, microservices can more efficiently use computing resources.

- **Increased Fault Tolerance**: Microservices also promote a higher degree of fault tolerance than their traditional monolithic service counterparts. Under the assumption that process boundaries separate services, a microservice implementation would necessarily involve *a higher* number of such services than monolithic architectures. Therefore, elegant response codes can control the "blast radius" of a crash or failure in one microservice. In a monolithic implementation, a failure in one component can bring down the entire application due to their tight coupling.

- **Support for Software Diversity**: Another advantage of building and integrating new components as services is that the limitation of programming these new components under a single "umbrella language" or framework model no longer applies. The separation of components into their processes allows flexibility in choosing the appropriate programming language to implement the functionality in question, provided that the component can interact with the outside world through some standard means.

3.1.2 Challenges Posed by Microservice Architectures. Splitting up a monolith into several microservices can potentially have functional side-effects such

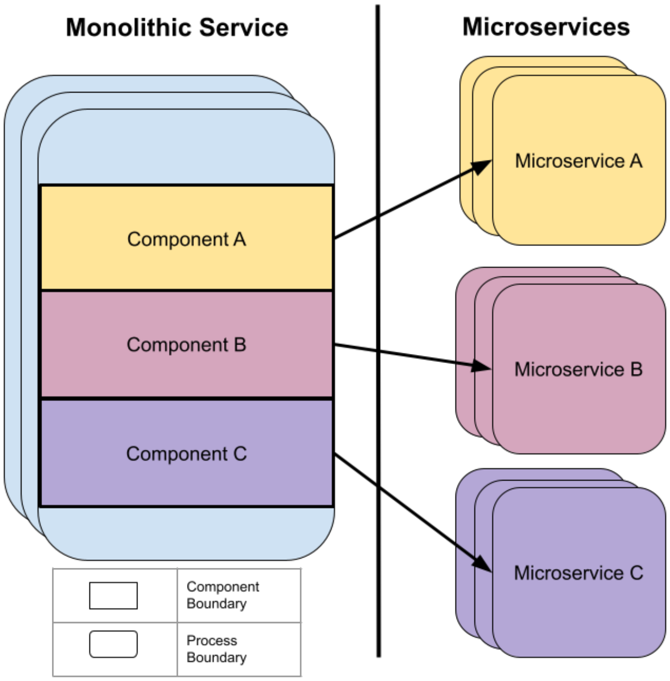


Figure 5. Monolith vs Microservices

as changing the *consistency model* for the service. For example, if a monolithic database is spread (split up) across multiple microservices, updates to different parts of the (overall) database may complete at different points in time. Therefore, the design of such a distributed database may involve a choice between a low-performance design guaranteeing ACID transactions and a higher performance design based on an eventual consistency model. While this is a broad research area in its own right, this dissertation focuses on understanding the performance challenges posed by microservices. Though the scale and the nature of the usage of cloud and high performance microservices vary significantly, a common set of performance challenges between cloud and high performance microservices is presented here:

– **Allocation of Resources**: Arguably, the most fundamental challenge relates to the question of what constitutes the optimal resource allocation to each microservice within a larger, composed distributed service. A sub-optimal allocation of resources can profoundly negatively impact the performance of the overall client-service workflow.

- **Modeling Performance and Identifying Bottlenecks**: The goal of identifying an optimal allocation of resources would necessarily involve the generation of a performance model for the service. The presence of tens or several hundreds of microservices interacting to serve several millions of concurrent client requests can make it difficult to model the overall service performance. Further, identifying the root cause and the correct resolution for a given performance bottleneck observation can prove challenging. As Figure 6 illustrates, there can be *different* paths traced in the system by (any) two requests — request 1 and request 2. Request 1 traces the path A→C→B→C→A, while request 2 traces the path A→B→A. In this simple scenario, identifying the relative ratio of request 1 and request 2 in a client workload can suggest how to split a given set of computing resources between microservices A, B, and C. For example, if the ratio of request 1 to request 2 is 2:1, a simple initial resource allocation ratio for the microservices A:B:C is 3:3:2, reflecting the total number of requests passing through them. Of course, this information alone is insufficient to judge the optimality of the resource allocation, as the immediate next question would be — What are the relative resource requirements for request 1 and request 2? An equally valid question would be — How do these request ratios change over time?

- **Enabling Adaptivity**: Concerning the question of service adaptivity, microservices expose a significantly larger search space due to the presence of several independent components and their associated tunable "knobs". There are two challenges to enabling adaptivity: (1) navigating this search space can be expensive, and (2) *correctly* implementing adaptivity may involve the orchestration and consensus of several different distributed services. Further, implementing adaptivity of high performance microservices involves the following additional challenges: (1) the clients in the former case (MPI applications) are not stateless entities and may thus require checkpointing to store and retrieve state between adaptations, and (2) modeling the cost-benefits of adaptivity is strictly time-bound because high performance services are transient. At the same time, it is worth noting that microservice architectures' highly modular nature opens up opportunities for fine-grained control of "knobs" affecting overall service performance.

3.2 High Performance Microservices: A Background

Although the questions on performance observability and the solutions proposed thereof are widely applicable to any high performance microservice stack, the high performance microservices that this dissertation considers are built on top of the Mochi [6] software stack. This section presents an overview of the core components that enable Mochi microservices, the RPC execution model, and examples of production-ready Mochi microservices.

3.2.1 Mochi: A Background. Storage systems on today's high-performance computing (HPC) platforms are complex and rapidly evolving because of the continuous adoption of new technologies in storage hardware, networking infrastructure, memory, and compute resources. On the application front, the traditional MPI-based parallelism is increasingly supplemented by large-scale task parallelism [23, 174]. The heterogeneity in hardware, diversified application mix, and execution environments coupled with increasing on-node parallelism complicates the

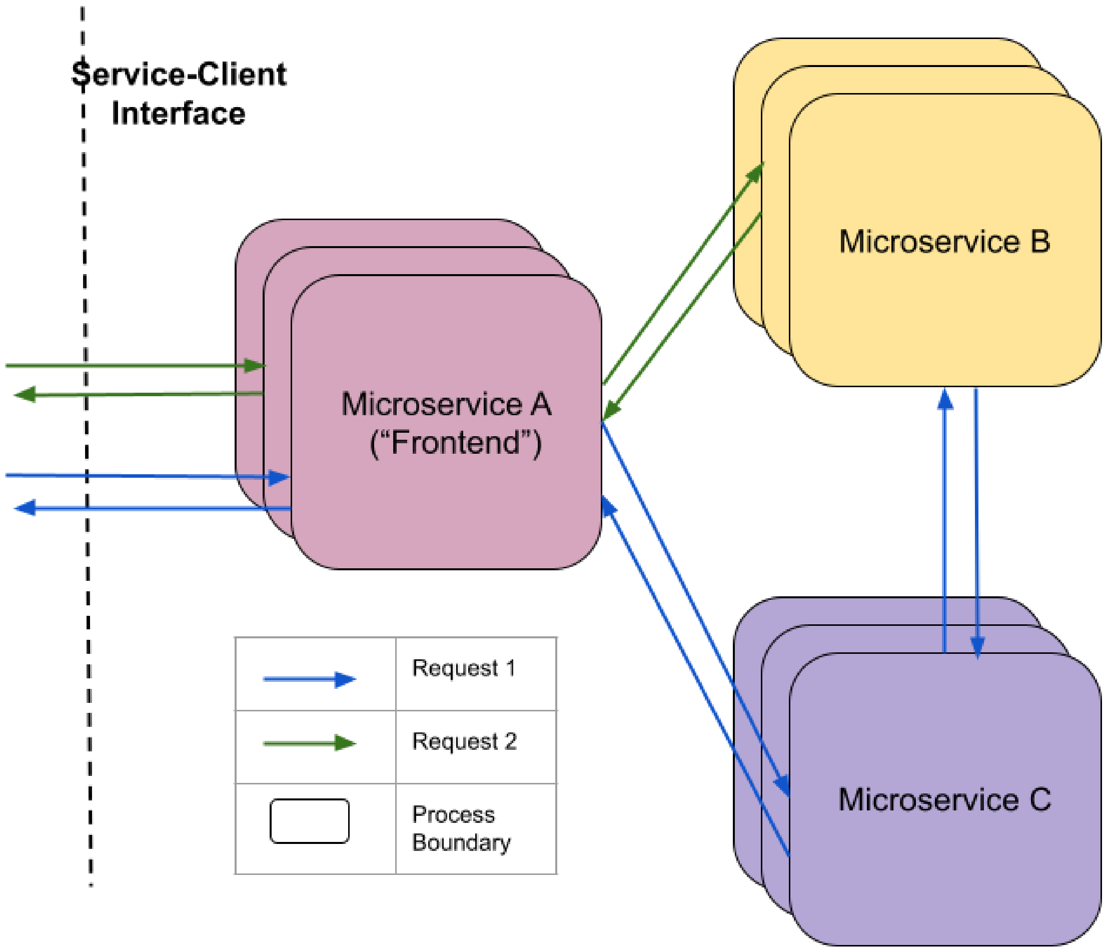


Figure 6. Microservices: Interactions Resulting From Requests Through the Service

task of managing and optimizing I/O performance and meeting the application's data needs.

Composability is a valuable development paradigm for this complex environment; it allows distributed services to be incrementally developed and improved in a modular fashion. The Mochi project [6] is an example of an HPC framework that embodies this principle. It structures and catalyzes the development of customized HPC data services through microservices that can be rapidly composed to meet application requirements. The Mochi project seeks to enable this rapid development of customized data services by providing an array of out-of-the-box microservices for various standard data storage and analysis operations. Examples of such microservices are the SDS key-value store (SDSKV), the BAKE blob object store to store large data objects and the REMI resource migration microservice. These microservices, in turn, provide a uniform client API to access a variety of data storage "backends", effectively serving as an abstraction layer to access and integrate existing and new storage technologies in an incremental and modular fashion.

Users provision a set of nodes on an HPC cluster through the batch system and split the allocation between the various workflow constituents to use Mochi services. The workflow could involve a mix of MPI applications, ML-based tasks, analytics routines, and visualization modules. Note that the Mochi services are *coupled* with the main workflow components in a transient manner, implying that the service runs in userspace for the duration of the batch job. Further, it is up to the user to identify *how* to split the allocation between the workflow entities.

Concerning the development of custom Mochi composed services, the methodology inspired from Ross et al. [6], involves four steps: (1) gathering the user requirements, (2) gathering the service requirements, (3) identifying the Mochi microservices to compose, and (4) instantiation of the Mochi services. Specifically, the key idea is to speed up step (3) through a set of already available, out-of-the-box, high performance microservices that can be composed in a “plug-and-play” fashion. Step (4) allows the user significant flexibility in deciding how to deploy the composed service. Microservices can be shared amongst multiple composed services that are a part of the workflow.

3.2.2 Mochi: Core Components.

Mochi data services are built and composed by using the RPC as the fundamental communication method between processing elements, whether they are local or remote. A Mochi client (referred to interchangeably as an *origin* entity) contacts a service provider (referred to as a *target* entity). The Mochi ideology is to provide the tools and environment necessary to enable the rapid development of HPC data services. This goal is achieved by *composing* microservices to build higher-level, customized functionality. This section describes the core components that enable Mochi microservices.

Mochi’s core components include Argobots [121], Mercury [120], Margo/Thallium, and SSG (Scalable Service Groups). For our study, we focus on the first three.

3.2.2.1 Argobots.

Developed outside the scope of the Mochi project, Argobots is a user-level threading library designed for highly concurrent systems. Argobots decouples the work (user-level threads, or ULTs) from the hardware resources that perform that work (execution streams, or ESs). Argobots was created to provide *lightweight* threading support on modern CPUs that support high degrees of concurrency. Specifically, Argobots strives to implement this lightweight threading support through lightweight notifications, fast thread-switching mechanisms, and efficient data movement and mapping strategies.

Concerning Mochi services, the three fundamental Argobots abstractions are execution streams (ESs), work pools, and work units (WUs). ESs process instructions sequentially and are bound to hardware resources (CPU cores). A concrete implementation for an ES is an OS thread bound to a CPU core. The ES can be associated with a dedicated work pool from which it pulls WUs for execution. Further, the ES can be associated with one or more schedulers that manage the execution of WUs on the ES.

Argobots WUs are of two types — user-level threads (ULTs) and tasklets. While both these WUs represent independent sets of instructions, ULTs are unique in two ways: (1) they are non-preemptable entities and will yield control to the scheduler either when explicitly programmed to do so, or when their execution completes, and (2) ULTs have their persistent stack, while tasklets borrow the stack of the ES's scheduler. This decoupling strategy between WUs and ESs allows the migration of WUs between different schedulers of ESs and between different ESs.

3.2.2.2 Mercury. Mercury is an RPC framework designed for HPC environments. Mercury takes advantage of RDMA-enabled HPC networks for large data transfers and a callback-driven completion model for concurrency. Notably, Mercury supports several different network implementations such as OFI, UCX, TCP, and shared-memory (SM) as plugins underneath a common *network abstraction* layer. A detailed description of Mercury's RPC execution model is presented in Section 3.2.3.

3.2.2.3 Margo/Thallium. As illustrated in Figure 7, Margo is the common underlying layer for Mochi services to interact with RPCs and RPC handlers. Margo eases the burden of Mercury callback programming and Argobots concurrency management and presents a unified model that leverages both technologies. Margo operates in two modes: client and server. When used in server mode, Margo allows the registration of one or more *providers.* Essentially, a provider is a *uniquely identifiable network object* that can receive and make RPC calls. A Mochi provider implements a given microservice API through one or more backend targets. Further, a provider can be assigned dedicated Argobots work pools and ESs on which to schedule the execution of RPCs.

Thallium is a `C++` wrapper for Margo and exposes an object-oriented interface for making RPC calls. Notably, Margo/Thallium supports the execution of blocking and non-blocking calls through the same interface. When a blocking call is made, the client (origin) thread blocks (waits) for a response to be received before the call completes. When a non-blocking call is made, Margo/Thallium returns a handle to the client as soon as the (send) data buffer is safe to be reused. The client can check the status (completion) of the RPC through a separate API call. The progress semantics for the RPC is determined by the network implementation and the configuration of the service.

3.2.3 Mochi: RPC Execution Model.

This section describes the events during the generation and execution of a Mochi RPC call.

3.2.3.1 Service Discovery.

Before an RPC call is made, Mochi service target providers must make their RPC addresses publicly known, usually accomplished through an *address file.* Inside a workflow, the general operation model assumes that the service processes are launched and initialized first, before the other workflow components such as MPI applications. When a client entity initializes, it is expected to look up the address file, identify the target provider, and use this information to create a Margo client object. At this point, the client is ready to use the service.

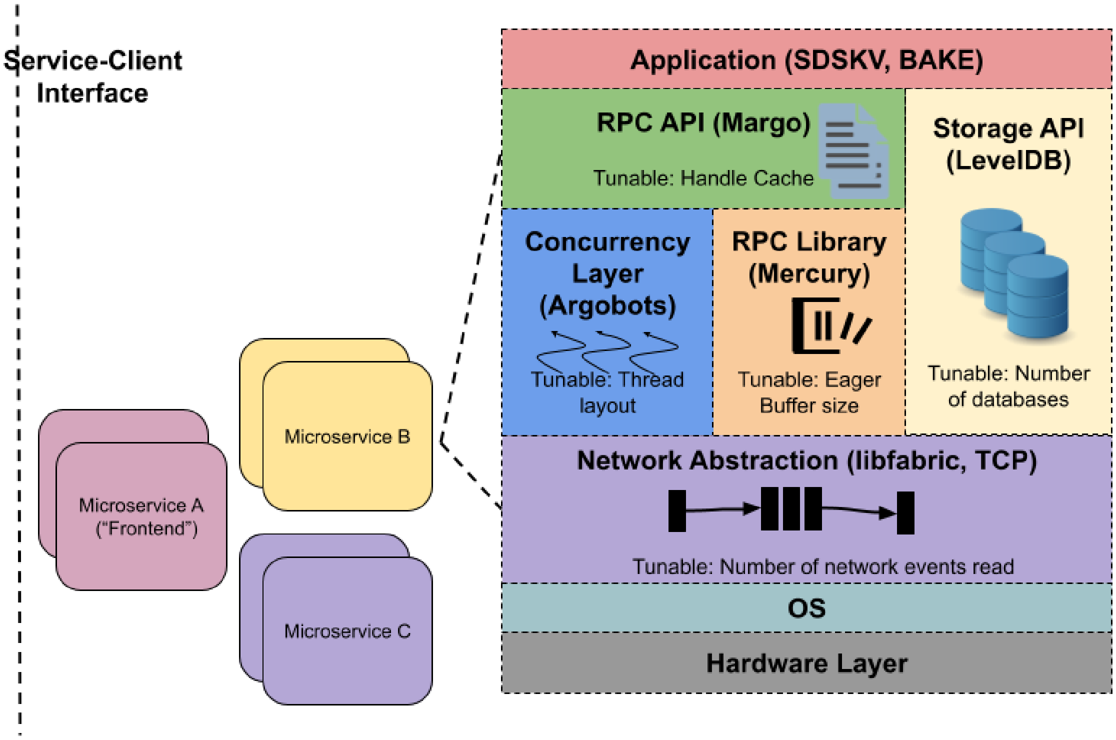

Service-Client Interface
Microservice A ("Frontend")
Microservice B
Microservice C
Application (SDSKV, BAKE)
RPC API (Margo)
Tunable: Handle Cache
Storage API (LevelDB)
Concurrency Layer (Argobots)
Tunable: Thread layout
RPC Library (Mercury)
Tunable: Eager Buffer size
Tunable: Number of databases
Network Abstraction (libfabric, TCP)
Tunable: Number of network events read
OS
Hardware Layer

Figure 7. Mochi: Interaction between Distributed Components and Software Stack

3.2.3.2 Request Generation. After the target provider address has been acquired, the origin entity generates an RPC request. The RPC request metadata is serialized inside Mercury and eagerly sent over to the target. In the case that the eager buffer overflows, Mercury employs an internal RDMA call to send the additional request metadata. Margo installs a callback with Mercury invoked when the response is available. These actions correspond to steps $\mathbf{t_1}$ to $\mathbf{t_3}$ in Figure 8.

3.2.3.3 Execution of the Request. When a target provider receives an RPC request, the main service provider execution stream (progress ES) creates a *new* ULT to service the request ($\mathbf{t_4}$). This request enters a pool of tasks waiting to run on the next available ES. When the ULT is assigned an ES to run on, it begins executing ($\mathbf{t_5}$) by first deserializing the input metadata ($\mathbf{t_6}$ to $\mathbf{t_7}$). The number of ESs available to the service provider is specified during the initialization phase. These ESs

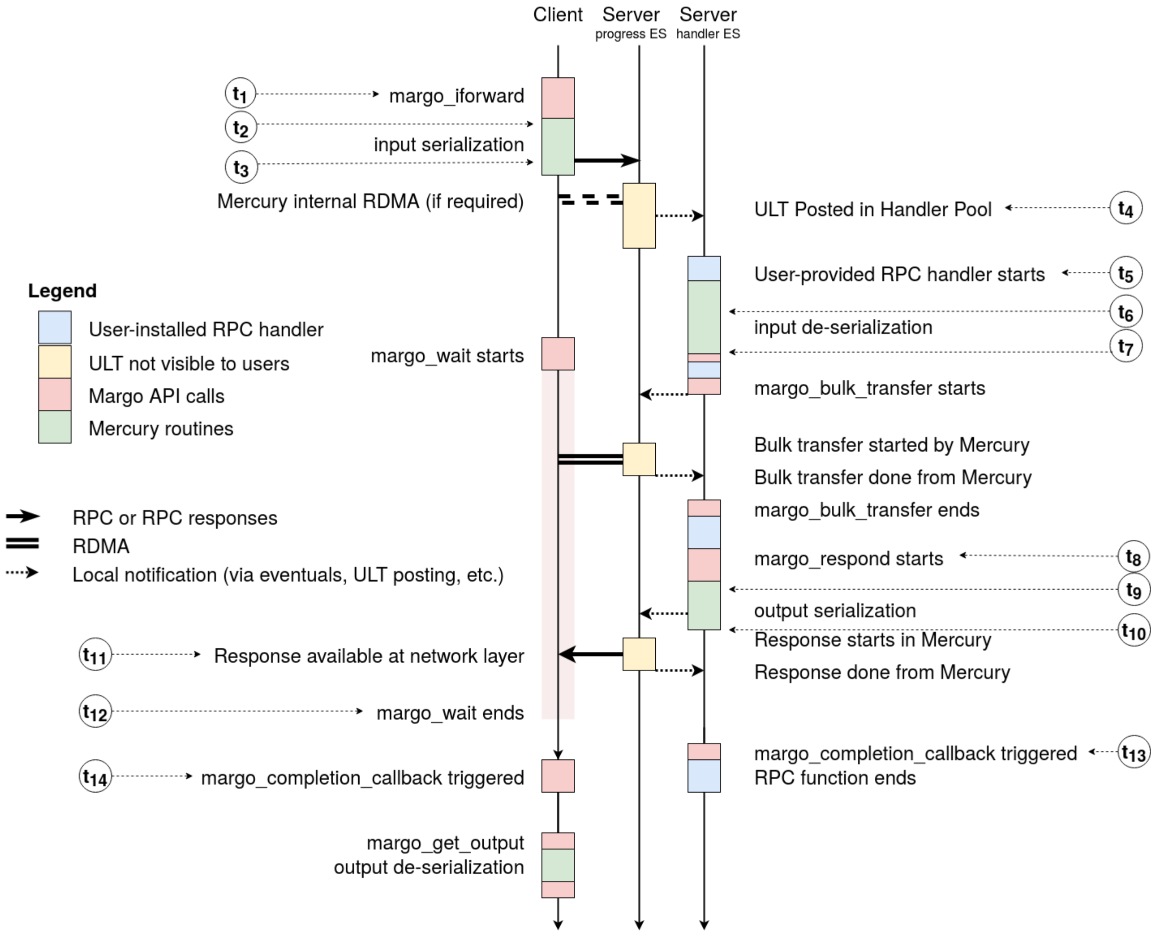

Client
Server
progress ES
Server
handler ES
t1
margo_iforward
t2
input serialization
t3
Mercury internal RDMA (if required)
ULT Posted in Handler Pool
t4
User-provided RPC handler starts
t5
t6
input de-serialization
t7
margo_wait starts
margo_bulk_transfer starts
Legend
User-installed RPC handler
ULT not visible to users
Margo API calls
Mercury routines
Bulk transfer started by Mercury
Bulk transfer done from Mercury
margo_bulk_transfer ends
RPC or RPC responses
RDMA
Local notification (via eventuals, ULT posting, etc.)
margo_respond starts
t8
t9
output serialization
t10
Response starts in Mercury
t11
Response available at network layer
Response done from Mercury
t12
margo_wait ends
margo_completion_callback triggered
t13
RPC function ends
t14
margo_completion_callback triggered
margo_get_output
output de-serialization

Figure 8. Mochi RPC Execution Model

constantly dequeue ULTs from the various registered pools and execute them as they arrive. If there are no ULTs to execute, the ESs remain idle. The service provider transfers data from the origin through Mercury's bulk interface.

3.2.3.4 Issuance of a Response. The target provider generates a response ($\mathbf{t_8}$), and the output gets serialized inside Mercury ($\mathbf{t_9}$ to $\mathbf{t_{10}}$). The Margo library on the target registers a callback handler for the response. Mercury triggers this callback handler ($\mathbf{t_{13}}$) when the response has been sent to the origin.

3.2.3.5 Receipt of a Response. Once the response is available at the network layer on the origin ($\mathbf{t_{11}}$), at some later point in time the Mercury progress engine adds the completion callback for this request to the completion queue ($\mathbf{t_{12}}$). Then the callback for this request is triggered ($\mathbf{t_{14}}$).

3.2.4 Mochi: Microservices. Mochi microservices have two components — the client library and the service library. The client library is a *stub* that serializes input arguments and deserializes the output using Mercury. The server library implements the RPC API. After deserializing the input and performing any necessary RDMA operations to pull in the bulk arguments, the server invokes the API of the backend target during execution. For example, the SDSKV Mochi microservice supports three backend targets — std::map, LevelDB, and BerkeleyDB. Likewise, the BAKE microservice provider supports two backends — a file backend and a persistent memory (PMEM) backend. Abstracting the backend implementation behind a common API allows Mochi to be portable across platforms and targets without code modifications. Presently, Mochi supports the following non-exhaustive list of out-of-the-box microservices:

- **SDSKV** (YOKAN) key-value microservices

- **BAKE** object store

- **REMI** resource migration microservice
- **BEDROCK** bootstrapping microservice
- **COLZA** in situ visualization and analysis microservice
- **POESIE** microservice for executing scripts inside interpreters (Python)
- **SONATA** document store
- **SSG** microservice for group membership operations

A complete list of services supported by Mochi can be found on GitHub (https://github.com/mochi-hpc).

3.2.5 Mochi: Composed Services. The Mochi project has resulted in the development of several composed services across several institutions and laboratories. HEPnOS [6] is a data service for a high-energy physics application that uses the SDSKV microservice. Flamestore [6] is a storage service designed to store the results of the training of ML models on HPC systems. Mobject [6] is a distributed object-storage service that implements a subset of the RADOS [175] API. As Figure 9 illustrates, Mobject is itself composed of three microservices — the SDSKV key-value microservice (for storing metadata), the BAKE microservice for storing object data, and the Mobject "sequencer" microservice to translate a client request into a sequence of RPC calls to the appropriate SDSKV and BAKE service providers.

Mochi components have also been used to design user-level filesystems. UnifyFS [113] provides seamless access to burst-buffer storage on a compute node. The DeltaFS [176] is a user-level filesystem designed to overcome the file metadata bottlenecks encountered by HPC applications that read and write many small files.

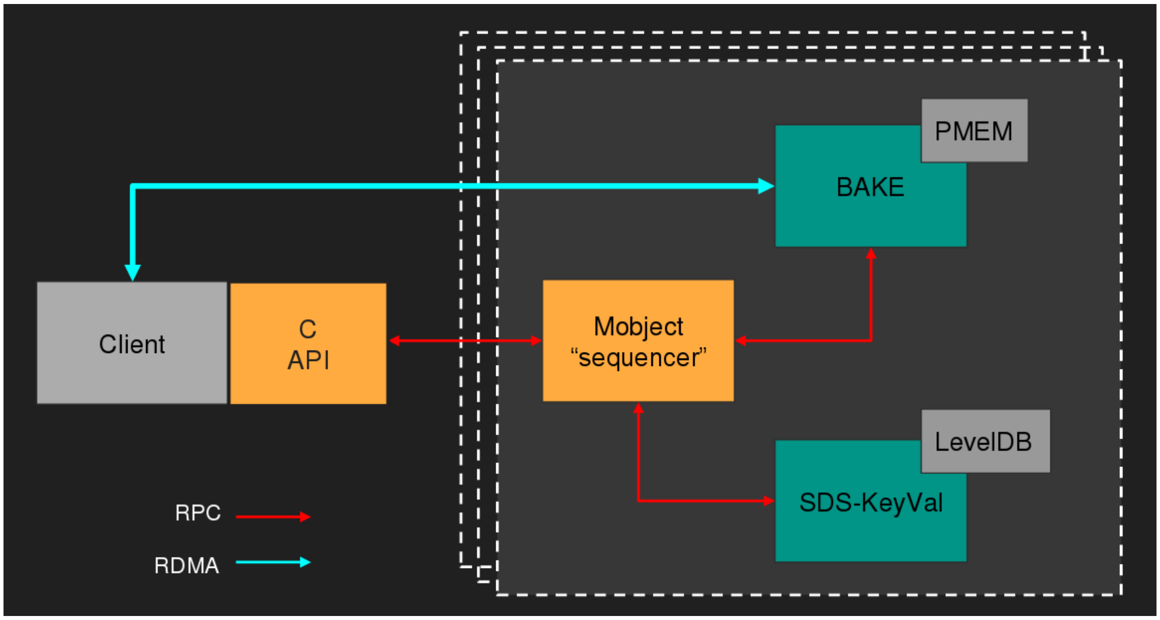


Figure 9. Mobject: An Illustration (Image Credits: Dr. Matthieu Dorier, Argonne National Laboratory)

3.3 The Goal of Performance Observability

Performance observability is defined by Malony [177] as the “ability to accurately capture, analyze, and present information about the performance of a computer system”. This definition has broad applicability, and in the context of high performance microservices, observability has the following goals:

- Generate insight into the interactions amongst the microservice components when coupled with other in situ workflow components,
- Aid in forming a performance model or understanding of the execution of these services that closely reflects their execution model,
- Pinpoint sources of inefficiency by measuring the saturation levels of various hardware and software resources, and
- Ultimately aid in the generation of a better performing service configuration.

Performance monitoring, on the other hand, implies the storage, export, and *online* analysis of some or all of the captured observations by an external entity (human user or another program). Therefore, while performance observability and monitoring are related concepts, the former is primarily concerned with making visible the system's "invisible" performance aspects. The latter deals directly with how to present information for online adaptivity and decision-making while the system is executing. Put another way; performance observability is a necessary first step to be established before monitoring is even possible.

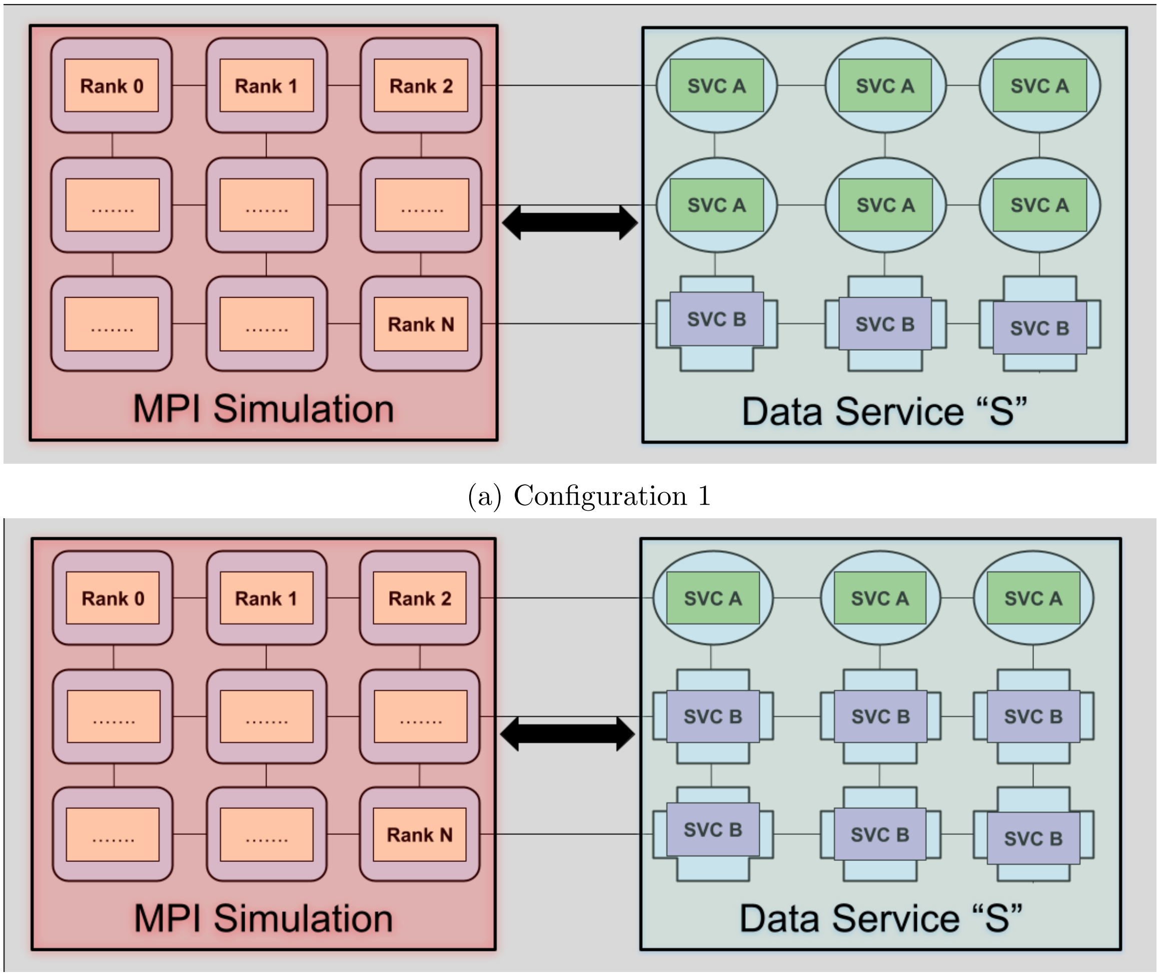
To understand what makes performance observability important for high performance microservices, consider the two service configurations presented in Figure 10. Both configurations involve an in situ workflow that couples an MPI application (workflow component) with a Mochi data service "S". Data service "S" is composed of two types of microservices: (1) microservice A, denoted by the green boxes, and (2) microservice B, denoted by the purple boxes. Configuration 1 involves three instances of microservice A coupled with six instances of microservice B, while Configuration 2 involves six instances of microservice A and three instances of microservice B. Note that individual instances of each microservice type are *identical copies* of each other. Therefore, from the perspective of the service client (MPI application), Configuration 1 and Configuration 2 are *functionally equivalent* but can have *vastly* different performance characteristics, resulting in up to an order of magnitude or more of a performance difference for the overall workflow.

To understand why, consider how the client is expected to use the data service "S". Let microservice A represent a key-value store, and microservice B represent an analysis service. During execution, a client process (MPI rank) may want to store a piece of data in the service. Given that all instances of microservice A are identical, this process generates a hash of the object to identify which instance of microservice A to store this piece of data. Hashing is one way to implement the load balance of data within the service. Another client rank may want to perform analysis on a piece of data for which it contacts an instance of microservice B. Note that the client-server coupling in this example scenario is two-way, meaning the performance of the workflow depends not only on the scalability of the MPI application but also on the optimality of the data service configuration. A haphazard configuration of the service without the knowledge of the precise mix of the operations on the critical path of the workflow can result in an overall poor performance for the workflow.

Therefore, performance observability is the first step in addressing the main question this dissertation attempts to answer: **How to enable and use performance insight to improve service configuration when the service is a part of a coupled, HPC in situ workflow?** Note that deciding on "an improved service configuration" involves a broader range of contributing factors than just identifying the optimal number of microservices of each type. Specifically, these additional factors include the number of Argobots ESs (threads) to assign to each microservice, the software knobs for each framework component, and the client-side batching strategy for RPCs. Two observations concerning these factors affecting performance make observability challenging:

- Overall service performance is often a result of the interplay between these factors, i.e., these contributing factors cannot be analyzed independently, and
- These factors affect different portions of the microservice software stack.

These observations suggest that an *integrated* performance analysis strategy would likely prove most effective.



(a) Configuration 1

(b) Configuration 2

Figure 10. Functionally Equivalent Service Configurations

Table 11. Improving Service Performance By Addressing a Set of Performance-related Queries

Query Number	Query
Query 1	What combinations of dependent microservice operations have the greatest impact on performance?
Query 2	Is there load imbalance in the service?
Query 3	How to gain insight into the individual request structure?
Query 4	How do the individual microservice operations map to resource usage and time for RPC events?
Query 5	What hardware and software resources are being saturated?
Query 6	How to gain insight into the performance of application-level APIs?
Query 7	Is there a better service configuration?

3.4 Elements of Observability

This dissertation proposes to break down the broader question of generating an improved service configuration into a set of specific performance-related queries for the observability infrastructure to target. These queries are listed in Table 11.

The following sections present a detailed overview of each of these performance-related queries and discusses how answering these queries can address the main research question.

3.4.1 Query 1: Identifying Dominant Microservice Operations. Conceptually, the dependencies among the microservices are represented as distributed callpaths through the system. As opposed to monolithic architectures where callpaths are local to a process, generating callpaths for microservices is inherently tricky because these callpaths can span across multiple processes on different nodes. Microservices that make up a composed service are loosely coupled, work on potentially different scales, operate in a heterogeneous execution environment, and are configured in myriad ways. Figure 6 depicts a scenario where three microservices (A, B, and C) interact to generate two distinct callpaths in the system: A⟶C⟶B (shown in blue) and A⟶B (shown in purple). The microservices A, B, and C can be located on the same process, on different processes within a node, or on entirely separate nodes depending on how the service is configured. A summary of the various distributed callpaths in the system sorted based on the total call time or call counts can help identify: (1) the dominant (resource-intensive) callpaths that account for a majority of the execution time of the service and (2) the candidate microservices that could lie on the critical path for the workflow.

3.4.2 Query 2: Detecting Load Imbalance.

Analyzing the distribution of the call times (or call counts) for the dominant callpaths can help the developer quickly track down the load distribution for these callpaths and identify if there is a load imbalance. In a data service, for example, load imbalance can signal that the hashing scheme to distribute the data among the constituent microservices does not yield a uniform distribution.

3.4.3 Query 3: Eliciting Individual Request Structure.

Callpath summaries are a good first step in identifying the dominant microservice interactions, but these summaries alone are insufficient to *explain* why, for example, a certain request took longer than expected to complete. Consider the service deployment configuration described in Figure 6. In an alternate execution, Request 1 can take the following path: A→C→A→C→A→C→B→C→A. The callpath summary for this request would result in the generation of two callpaths: A→C→B and A→C. The red nodes indicate an RPC response. Essentially, the individual call sequences within a request are lost while summarizing the callpaths. However, if information about the entire request structure were available, it would help answer why, for example, the latency for Request 2 was several times higher than the average latency for this specific callpath.

3.4.4 Query 4: Mapping RPC Resource Usage and Time.

Once the microservice dependencies have been identified, it is imperative to understand the relative contributions of various software components and events that make up the remote procedure call (RPC) on the client and the server. Importantly, these events must closely reflect the vital performance-related aspects of the RPC execution model. To tease out these details, we need a way to integrate various sources of performance data and timers gathered from different levels in the stack and fuse that data with the distributed callpaths as common reference points.

3.4.5 Query 5: Detecting Hardware and Software Resource Saturation. Identifying a poorly performing service configuration involves the ability to detect resource saturation. Resource saturation can occur on a hardware level or software level. For example, tasking frameworks (such as Argobots) that manage the concurrency on the server place newly spawned tasks in internal queues. Observing a backlog of tasks on these queues is an indication that the tasking system is starved of compute resources. Correlating these resource saturation metrics with higher-level RPC callpath information can help narrow down the cause of the task pileup. Presenting a "hardware-centric" view of the service execution can identify when hardware resources are being saturated. This hardware-centric view can either be presented in a "software-agnostic" manner, separate from the callpath information, or in conjunction with the callpath information to narrow down the software component or service interaction *causing* the hardware saturation.

3.4.6 Query 6: Application-level API Performance. In a highly concurrent service environment, delays and inefficiencies can also happen inside the microservice API (application-level) and need to be identified through appropriate instrumentation. For example, a service exposing access to a database can experience request pileups if: (1) the access to the database is serialized, or (2) the service is configured with too few database instances for the client workload. Without appropriate instrumentation and knowledge about the characteristics of the client workload, it is not easy to estimate the optimal number of these database instances.

3.4.7 Query 7: Identifying Better Service Configurations.

Ultimately, the goal of enabling performance observability is to aid in the generation of a better, more optimal service configuration if it exists. If resource saturation is detected, the performance data must sufficiently indicate what parameters could be changed to improve performance.

3.5 Summary

Chapter III presented an overview of microservices, touching upon the key benefits and performance-related challenges associated with this highly modular architecture, particularly in the context of HPC microservices. Chapter III also presented an introduction to the Mochi software framework [6] to describe the Mochi RPC execution model and the critical aspects of service operation requiring performance observability. While the ultimate stated goal for the dissertation is to help generate optimal HPC service configurations, Chapter III breaks down this goal into a set of concrete performance-related queries for the observability infrastructure to target. Chapter III sets the stage for discussing the tool solutions to address these performance queries, presented in Chapter IV.

CHAPTER IV

SELECTING AND COMBINING SOURCES OF PERFORMANCE DATA

This chapter contains previously published material with co-authorship. All of the presented research in this chapter was conducted as a collaboration between the University of Oregon and Argonne National Laboratory. The research work in this chapter was presented at IPDPS 2021 [13] and HiPC 2021 [16]. I was the first author of the IPDPS 2021 and HiPC 2021 papers. While working on the SYMBIOSYS tool, I received regular guidance from Dr. Philip Carns and Dr. Robert Ross. Dr. Philip Carns put in place the initial version of the distributed callpath profiling support in Margo that I took forward. While working on SYMBIOMON, I received guidance from Dr. Matthieu Dorier, Dr. Robert Ross, Dr. Philip Carns, and Dr. Jerome Soumagne. For both of these publications, I did all the software development, conducted the experiments, writing, and data collection, with suggestions and edits from Dr. Philip Carns and Dr. Robert Ross. For both of these publications, I received guidance from Dr. Allen Malony. Other co-authors also helped in the proofreading. This chapter is formulated by gathering my contributions from these two publications.

4.1 Introduction

The difficulty with enabling the analyses raised in Chapter III lies in understanding how to combine various instrumentation and measurement techniques to present an integrated analysis and profile. Observing distributed callpaths, for example, involves tracking and forwarding RPC call ancestry across distributed microservices by employing some form of request metadata propagation. Attributing resource usage to individual steps within a microservice operation involves exchanging performance data across the software stack and orienting it around appropriate reference points.

Identifying resource saturation requires the ability to correlate low-level performance metrics with high-level callpath information.

This chapter describes a solution to the queries raised in Chapter III and in the process, addresses the research question **RQ1**: How to enable performance observability (insight) of services built by composing high performance microservices? The solution presented in this chapter consists of a set of techniques packaged as two separate tools — SYMBIOSYS and SYMBIOMON. While SYMBIOSYS is designed to generate and analyze performance observations offline, SYMBIOMON is a flexible tool that allows for both offline analysis of time-series metrics (event traces) and online monitoring and analysis of this metric data. This chapter is concerned primarily with using SYMBIOMON's instrumentation capabilities to analyze performance observations offline. The online monitoring aspects of SYMBIOMON are presented in Chapter V.

4.2 Related Work

This section presents a brief overview of the analysis activities that is central to HPC microservices and the effectiveness of different classes of tools to address these requirements.

4.2.1 HPC Performance Tools.

HPC performance tools excel at the performance analysis of applications based on the distributed-memory parallel programming model. Typically, they build on the presence of an MPI programming model to capture distributed performance information. State-of-the-art tools such as TAU [29], ScoreP [157], CALIPER [31], and HPCToolkit [30] employ sophisticated sampling, automatic compiler instrumentation, manual instrumentation, and library interposition techniques to gather insights into application and communication library performance. These tools implicitly assume that control is not passed *between* applications. As a result, HPC performance tools cannot be directly utilized to observe distributed microservice callpaths. While some HPC tools are capable of working with user-level tasking frameworks such as Argobots (e.g., APEX [171]), almost all are constrained to measuring code performance within a node, with limited application in the generation of distributed callpaths.

4.2.2 Cloud-Based Tools for Microservices. Several efforts have been made within the general distributed systems community and industry to design performance tools for microservice-based distributed services. They employ some form of metadata propagation to stitch together request trace events across processes to form a complete picture. Distributed request tracing is effective in detecting structural and empirical anomalies [178]. A comprehensive survey of the variety of tools available for distributed tracing can be found in [179]. Dapper [14] from Google, OpenZipkin [2], and Jaeger [163] are the notable industry efforts at tracking requests and associated metadata through a hyper-scale distributed setup. Our distributed tracing implementation is compatible with the trace format of these tools. Unlike these tools, however, SYMBIOSYS does not require additional processes on the node for staging performance data. Further, we find the need to extend their data model to support generating and capturing a wide variety of performance data from across the stack.

The SYMBIOMON metric data model is inspired by time-series monitoring databases used in the cloud industry, such as Prometheus [19], Graphite [170], and InfluxDB [180]. Cloud-based monitoring frameworks are typically employed to extract data over coarse-grained time intervals (seconds). The services they monitor are long-running, are spread over a large geographical region, and run on top of a commodity hardware and software stack. Services in the cloud are written in various languages, and thus cloud-based monitoring frameworks offer rich, multilanguage client instrumentation support. Importantly, these monitoring frameworks are not set up to directly enable fast, dynamic service reconfiguration. Instead, they rely on alerting mechanisms complemented with powerful, human-friendly remote querying capabilities. HPC data services are transient, highly concurrent services that run on high-performance hardware. Thus, while the cloud-based time-series data model has application in monitoring HPC services, the specialized hardware and software stack necessitates a high-performance monitoring service implementation.

4.2.3 Tools That Integrate Data Sources. A growing body of research employs techniques to exchange vital performance data between software layers. Notably, within the MPI community, there are ongoing efforts [15, 181, 182] to expose internal MPI counters and events to gain a deeper insight into the distributed communication performance. The OpenMP community is also pursuing similar efforts [51, 183] to associate library-level performance data with higher-level tasks. The PAPI software-defined events [184] approach aims to standardize the exchange of software-level performance metrics across layers through *accessor functions*. Our performance data strategy is inspired by the efforts in the MPI community, whereby tool support is directly available in the communication library as opposed to employing an external component.

4.3 SYMBIOSYS: Distributed Callpath Profiling

4.3.1 Instrumentation. SYMBIOSYS tracks RPC callpath ancestries to present a callpath profile summary. This summary contains information about the total amount of time spent along different callpaths (or *callchains*) in the system. Each microservice instance keeps track of its callpath ancestry and forwards this information along the request path. This callpath information is maintained separately on the origin and target entities. Further, for every callpath, each origin entity making the call and each target entity servicing the call are uniquely identified in the profile. The RPC call name is hashed into a 64-bit value and sent along with the RPC request during the RPC call. This 64-bit value denotes the *callpath ancestry* for the chain of RPCs. The Margo instance invoking the RPC stores this hash value inside Mercury at $\mathbf{t_1}$ (see Figure 11) and retrieves it from a callback argument at $\mathbf{t_{14}}$. At this point, the Margo instance measures the time it took for the RPC target to service the call. This measurement is referred to as the *origin execution time.*

The delay between the receipt of the RPC call at $\mathbf{t_3}$ and the execution of the corresponding ULT at $\mathbf{t_4}$ is denoted as the *target ULT handler time* and is stored in a ULT-local key. The Margo instance receiving the RPC call at $\mathbf{t_3}$ unpacks the incoming RPC request and stores the 64-bit hash value in another key local to the ULT servicing that request. Doing so is important because this ULT can make another RPC call as a side effect of the original one. If that is the case, the ULT needs to pass the callpath ancestry to downstream operations to maintain the correct chain of operations. The ULT first performs a 16-bit left shift of the 64-bit value representing callpath ancestry. It then hashes the name of the downstream call RPC and performs a logical `OR` operation such that the name of the downstream RPC call occupies the lowest 16 bits of the 64-bit value. The ULT then proceeds with making the RPC request. Currently, Margo can store RPC callpath lengths of up to four in the 64-bit hash value. When the ULT servicing the request on the target completes, it measures the time to service this request at $\mathbf{t_8}$. This is denoted as the *target ULT execution time.* The delay between the issuance of a response at $\mathbf{t_8}$ and the triggering of the corresponding completion callback at $\mathbf{t_{13}}$ is stored in a ULT-local key as the *target completion callback time.* In this manner, the Margo instance on each process (origin and target) keeps a track of the total call time and call count for *all* the callpaths that pass through the instance.

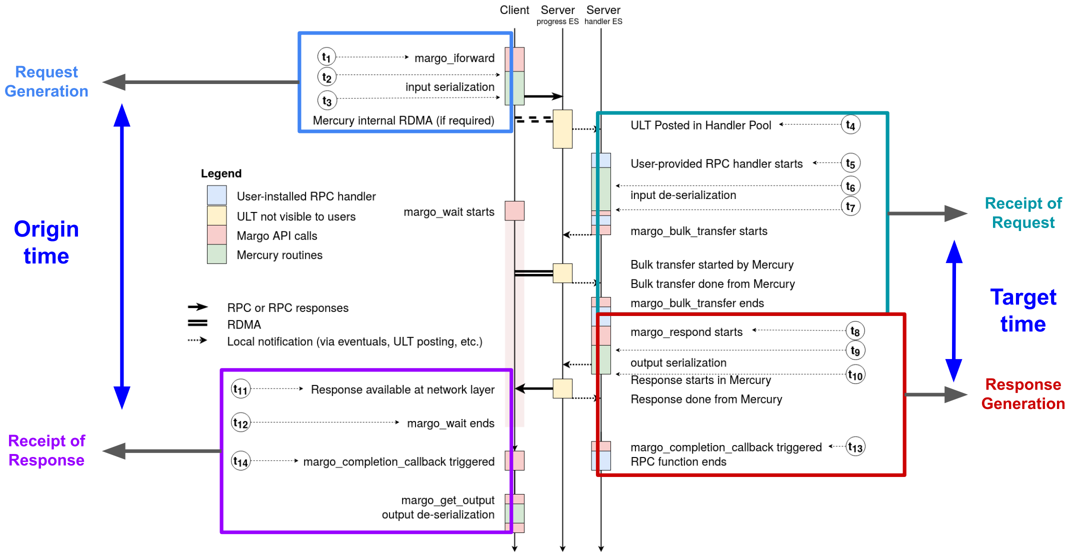


Figure 11. Annotated Mochi RPC Execution Model

4.3.2 Analysis and Visualization. The goal of distributed callpath profiling is to enable insight into the dominant microservice operations that account for most of the service execution time and resource usage. Therefore, the analysis and visualization modules were designed with this goal in mind. The SYMBIOSYS callpath analysis and visualization module perform the following analysis:

- At the end of the execution, each Margo instance in the workflow (origin and target) writes its local callpath profiles into separate files.

- The callpath analysis module reads these files and summarizes the origin and target profiles for every callpath observed. This summary includes the total execution time and call counts as seen by the origin and the target, the distribution of these call times and call counts, and other basic statistics.

- These callpath profiles contain a wealth of information, including all the system's microservice interactions; the callpath analysis module identifies the dominant callpaths by sorting the summarized profiles based on the call time and the call count.
- The visualization module ingests this data and generates a plot of these sorted, summarized profiles.

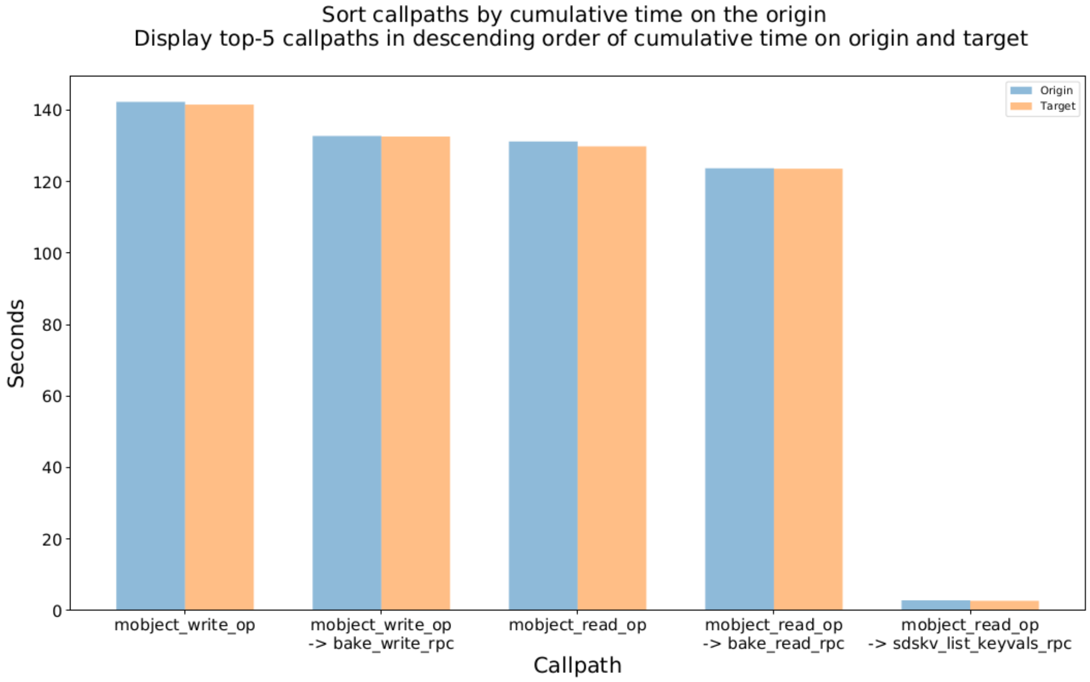


Figure 12. Mobject + ior: Dominant Callpaths by Call Time

Figure 12 illustrates the SYMBIOSYS callpath profile resulting from an execution of an ior benchmark [185] with the Mobject Mochi data service. The callpath profile depicts the top-5 callpaths sorted by cumulative call time. For this execution, the `mobject_write_op` accounts for over 50% of the total service call time (as seen by the ior client). Within this larger API call, the callpath `mobject_write_op`→`bake_write_rpc` is the dominant interaction, accounting for over 90% of the `mobject_write_op` execution time.

4.4 SYMBIOSYS: Distributed Request Tracing

4.4.1 Instrumentation. While callpath profiling helps gather a quick summary of service performance, information about individual requests is lost. Traces can span multiple nodes (and processes) and contain rich performance information that allows for correlations of various performance metrics with time to be performed. The key idea lies in propagating request metadata (typically a unique request ID) through the system and then having a post-processing system collect and stitch the individual trace events after completion. Distributed tracing involves the generation of trace events at $\mathbf{t_1}$, $\mathbf{t_{14}}$ on the origin and $\mathbf{t_5}$, $\mathbf{t_8}$ on the target. The end-client (typically an MPI application) generates a globally unique *request ID* and propagates this ID along with a counter representing the *order* of the event in the individual trace. We implement Lamport's algorithm [186] to mitigate clock skew in the system. For every trace event generated, the current timestamp is stored along with a wide variety of performance data gathered from the RPC API, RPC library, and concurrency control layers. Section 4.5 describes the methodology for retrieving and storing this rich performance data.

4.4.2 Analysis and Visualization. While the locations for instrumentation between distributed callpath profiling and distributed request tracing are common, the analysis and visualization strategy differs. The goal for distributed request tracing is to provide a detailed, *request-centric* view for the execution and enable insight into the structure of the individual requests and allow for the correlation of RPC request events with other performance data gathered from across the Mochi software stack.

The SYMBIOSYS request tracing module performs the following operations:

- When the execution of a service workflow completes, every instrumented Margo process in the workflow writes the locally-collected request traces into its local trace event file.

- Because the event traces that constitute a client request can be spread across multiple microservices (processes), the trace "pieces" belonging to each process need to *assembled* in memory, effectively stitching them together to form a coherent request timeline as it passes through various microservices.

- At this point, the processed requests are ready for analysis and visualizations. Concerning the analysis of the request events, a *temporal* view of the key aspects of service execution is presented, along with correlations of the occurrence of request events with resource usage such as memory and the Argobots pending queue sizes. For example, Figure 13 presents a timeline view of the execution of the `mobject_write_op` RPC for the ior + Mobject workflow described in Section 4.3. The first plot depicts the overall request latencies as a function of time. Interestingly, these latencies fall into two broad "ranges". The second plot depicts the maximum memory usage for any event in the request as a function of time. The third and the fourth plots capture information about the state of the Argobots RPC work queues on the target.

- Concerning the visualization of the request data, request tracing, provides valuable insight into the request structure of individual requests. The standard methodology for visualizing microservice call structures is a *gantt chart.* Instead of building a custom visualizer for this purpose, SYMBIOSYS leverages the standard JSON-based data model exposed by the existing, state-of-the-art Zipkin [2] tool. SYMBIOSYS implements a data converter module that maps the SYMBIOSYS traces to the Zipkin trace format to enable Zipkin-based trace visualization.

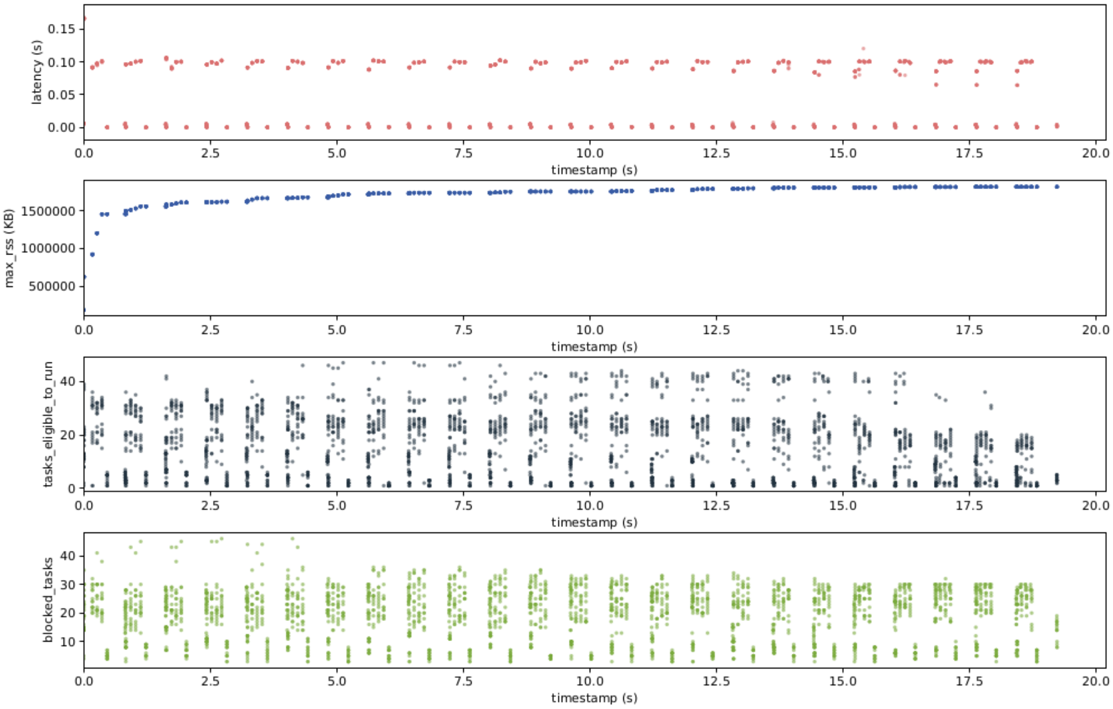


Figure 13. Mobject + ior: Trace Analysis for the mobject_write_op RPC

4.5 SYMBIOSYS: Performance Data Exchange

Associating higher-level callpaths with events from the RPC library and concurrency control layers is critical to forming a complete picture of RPC performance. Typically, each software item in the RPC stack behaves like a black box, preventing the exchange of vital performance data that can aid in understanding performance and can present optimization opportunities. Figure 14 presents a pictorial representation of the problem. The distributed callpath profiling and tracing techniques operate at the RPC API layer (Margo). While they are useful in eliciting high-level microservice interactions, they have limited visibility into the operation of the core RPC (Mercury) and the concurrency layers. The MPI community has attempted to standardize the exchange of performance data through the MPI Tools Information Interface [15]. The SYMBIOSYS architecture for performance data exchange with the RPC library takes inspiration from these efforts.

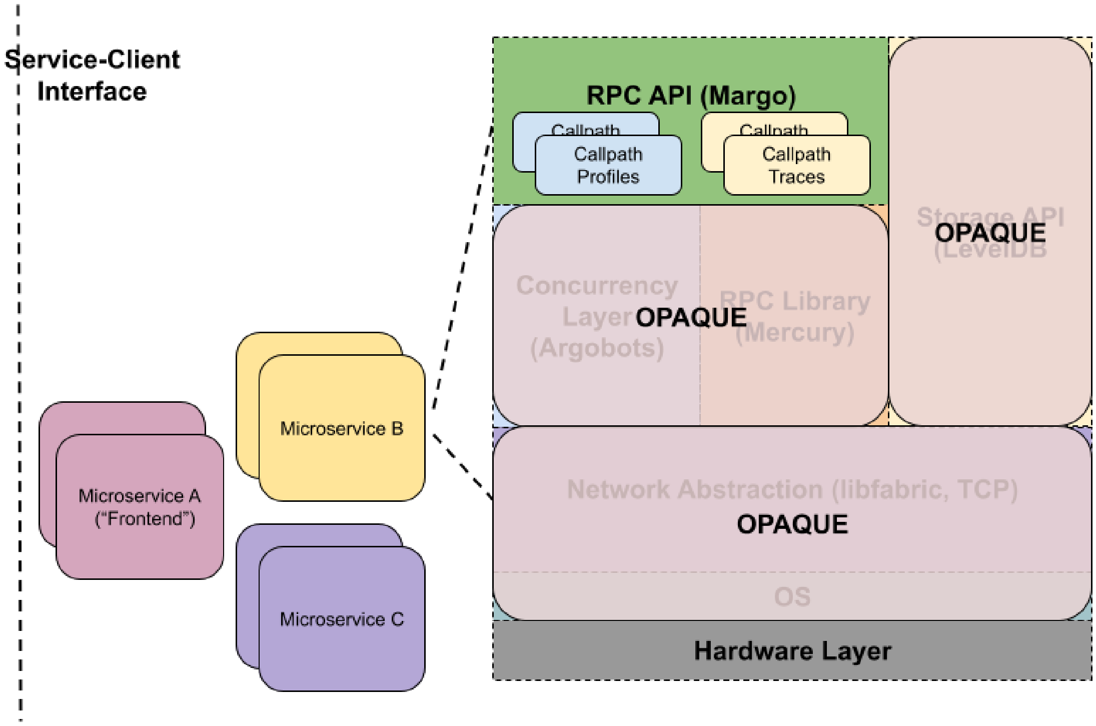


Figure 14. Mochi: Opaqueness of RPC Events to Callpath Profiling and Tracing

4.5.1 Performance Variables.

From the viewpoint of performance, several important events occur inside the Mercury communication library. We identify and implement several key performance variables (PVARs) in Mercury that capture these events. We introduce the concept of *PVAR classes* to represent the variety in the types of PVARs that can exist. For example, the PVAR class `STATE` is used to represent the current state of a particular Mercury resource or metric. Table 12 presents a list of PVAR classes currently available, and Table 13 lists some of the various PVARs that are currently implemented. The PVAR `num_posted_handles` represents a PVAR of the `STATE` class. Similarly, the PVAR class `COUNTER` represents a monotonically

Table 12. Performance Variable Classes

PVAR Class	Description
STATE	Represents any one of a set of discrete states
COUNTER	Monotonically increasing value
TIMER	Interval event timer
LEVEL	Represents the utilization level of a resource
SIZE	Represents the size of a resource
HIGHWATERMARK	Highest recorded value
LOWWATERMARK	Lowest recorded value

increasing value, and the PVAR classes `HIGHWATERMARK` and `LOWWATERMARK` denote the highest or lowest values recorded for a particular metric.

The other key concept we introduce is the notion of *PVAR bindings*. Many PVARs have a “global” scope and represent a counter or metric with a broad temporal and spatial presence across the Mercury library. Such PVARs have a binding type `NO_OBJECT`. An example of a PVAR of this type is the `completion_queue_count` representing the current length of the Mercury completion queue. Other PVARs are short-lived and have a much narrower scope. We introduce the binding type `HANDLE` to represent PVARs bound to internal Mercury handles. Every RPC call is internally associated with a Mercury handle object. Once the particular RPC has been completed, these PVARs go out of scope, and their values are lost forever. Examples of such PVARs include the timers for the input and output serialization and deserialization times on the origin and the target.

4.5.2 Performance Tool Interface. We introduce a PVAR interface in Mercury to externally sample these Mercury PVARs. Briefly, the steps taken by an external tool to access and sample the PVARs are as follows:

Table 13. List of Available Performance Variables

PVAR Name	Description	PVAR Class	PVAR Binding
num_posted_handles	Number of currently posted RPC handles	LEVEL	NO_OBJECT
completion_queue_size	Number of events in Mercury's completion queue	STATE	NO_OBJECT
num_ofi_events_read	Number of OFI completion events last read	LEVEL	NO_OBJECT
num_rpcs_invoked	Number of RPCs invoked by instance	COUNTER	NO_OBJECT
internal_rdma_transfer_time	Time taken to transfer additional RPC metadata through RDMA	TIMER	HANDLE
input_serializaton_time	Time taken to serialize input on origin	TIMER	HANDLE
input_deserializaton_time	Time taken to de-serialize input on target	TIMER	HANDLE
origin_completion_callback_time	Delay between the arrival of RPC response and invocation of completion callback	TIMER	HANDLE

- Initialize a PVAR session: Each tool querying the Mercury PVAR interface is assigned a unique `session_handle`.

- Query the interface to gather a list of the supported PVARs: Once a session is initialized, the external tool queries the interface to gather information about the number, type, binding, and count of all the PVARs exported.

- Allocate handles for PVARs: Once the relevant PVARs have been identified, the tool must allocate `pvar_handles` for the PVARs it wishes to read. The interface provides an API call for this purpose.

- Sample PVARs: After the handle has been allocated, the external tool can sample (read) the value of the PVAR by providing the `pvar_handle` as input to the sampling API. If this PVAR is bound to a Mercury handle, the tool must provide the Mercury handle as input.

- Finalize the PVAR session: When the external tool is done sampling PVARs, it can free the allocated `pvar_handles` and finalize the PVAR session.

4.6 SYMBIOSYS: Orienting Performance Data to Generate Observability

The microservice callpath measurements are used to orient and integrate performance data from the communication library. The Margo RPC API layer initializes a PVAR session with Mercury inside its initialization routine. At the

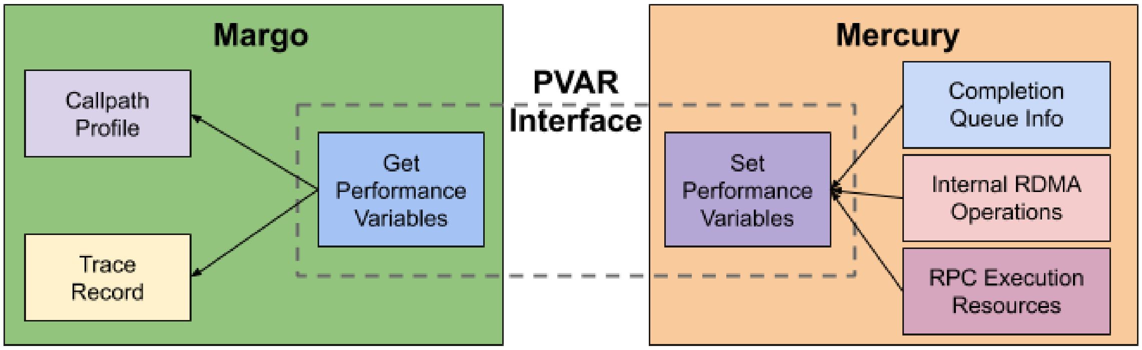


Figure 15. PVAR Interface Between Margo and Mercury

same time, it also initializes all necessary PVAR handles. Figure 15 represents this interaction between the two layers of the Mochi stack. Margo samples the Argobots layer for the number of blocked and runnable tasks when generating a trace event. These instrumentation points also serve as helpful to sample memory usage and CPU utilization from the OS layer.

Although the OpenFabrics Network Interface [187] specification does not allow us to read the instantaneous number of events in its completion queue, we can gather a sense of the size of the completion queue by sampling the *number of actual events last read* in the form of the `num_ofi_events_read` Mercury PVAR at $\mathbf{t_{14}}$. When Mercury PVAR profiling is enabled, it samples the `num_ofi_events_read` PVAR and adds this data to the trace record. At the origin, Margo reads the PVARs holding the *origin callback completion time* and *input serialization time* when measuring at $\mathbf{t_{14}}$. Similarly, at the target, the PVARs representing the *target internal RDMA transfer time*, *input deserialization time*, and *output serialization time* for the particular RPC call are sampled when measuring at $\mathbf{t_{13}}$. These trace events and profile data are consolidated, aggregated, and presented for visualization at the end of the execution.

Table 14. Combining Instrumentation Strategies

Interval Name	Interval Start	Interval End	Instrumentation Strategy
Origin Execution Time	$\mathbf{t_1}$	$\mathbf{t_{14}}$	ULT-local key
Input Serialization Time	$\mathbf{t_2}$	$\mathbf{t_3}$	Mercury PVAR
Target Internal RDMA Transfer Time	$\mathbf{t_3}$	$\mathbf{t_4}$	Mercury PVAR
Target ULT Handler Time	$\mathbf{t_4}$	$\mathbf{t_5}$	ULT-local key
Input Deserialization Time	$\mathbf{t_6}$	$\mathbf{t_7}$	Mercury PVAR
Target ULT Execution Time (exclusive)	$\mathbf{t_5}$	$\mathbf{t_8}$	ULT-local key
Output Serialization Time	$\mathbf{t_9}$	$\mathbf{t_{10}}$	Mercury PVAR
Target ULT Completion Callback Time	$\mathbf{t_8}$	$\mathbf{t_{13}}$	ULT-local key
Origin Completion Callback Time	$\mathbf{t_{12}}$	$\mathbf{t_{14}}$	Mercury PVAR

4.7 SYMBIOSYS: Sampling Node Resource Usage

While the callpath profiling, tracing, and performance data exchange strategies enable insight into the operation of the RPC software stack and generate observations of the microservice interactions, an "out-of-band" sampling strategy that captures basic node resource usage statistics such as the memory usage, CPU utilization and load average can be helpful for performance monitoring. SYMBIOSYS implements this sampling strategy by creating an Argobots user-level thread (ULT) inside Margo during initialization. This ULT is scheduled to execute periodically on the primary ES. The ULT captures these node resource statistics and appends this information to an internal record buffer when it runs. When the Margo instance is finalized, this record buffer is written out as a profile and analyzed offline. The node information is appended as additional metadata for the analysis to ingest.

4.8 SYMBIOMON: Time-series Metrics

While SYMBIOSYS enables performance analysis of the critical aspects of the RPC software stack and captures the overall execution time for the RPC on the origin and the target, we find this instrumentation strategy is insufficient to explain the *cause* for the delays occurring inside the microservice API. Gaining insight into these delays requires additional instrumentation to be placed inside the microservice API.

Specifically, in a microservice that is expected to operate efficiently under concurrent access, time-series metric analysis (a form of event tracing) proves beneficial. Time-series metrics are commonly employed for performance monitoring of cloud services. Time-series data can capture trends, allow for “windowed analysis” on the time-series data, detect anomalies in function execution times, and correlate measurements across different metrics to narrow down the root cause of performance inefficiencies. This section describes the time-series metric interface and the implementation of the COLLECTOR microservice component developed as a part of the SYMBIOMON monitoring system.

4.8.1 COLLECTOR Microservice. The COLLECTOR microservice exports the metric collection API that any service component or MPI process can invoke to create and update arbitrary metrics. Notably, the COLLECTOR is the only SYMBIOMON component interacting directly with a metric API client. COLLECTOR microservice instances run within the address space of the metric API client. In other words, the COLLECTOR operates like a library-based microservice, and all COLLECTOR API calls are implemented as regular function calls. This microservice is unique because the origin entity and the target provider are located within the same process, and they communicate through regular function calls (as opposed to using RPCs). This design choice was made recognizing that clients interact with their local COLLECTOR provider only for the most frequently used operations, such as metric creation and update. Implementing the COLLECTOR as a library ensures that these frequently used operations finish quickly and introduce minimal unnecessary overhead. During metric creation, the COLLECTOR associates an *internal buffer* of a fixed size with the metric and returns a *metric handle* to the client. The client subsequently uses the metric handle to update and destroy the

Table 15. SYMBIOMON Metric API

Microservice API	Key Input Arguments
COLLECTOR_taglist_create(destroy)	taglist
COLLECTOR_metric_create(destroy)	name, namespace, type, taglist
COLLECTOR_metric_update(destroy)	value, sample_id
COLLECTOR_remote_metric_fetch	collector_address, metric_id, num_samples
COLLECTOR_output_raw_metric_data	filename, metric_id
COLLECTOR_metric_reduce(all)	metric_id, reduction_operator
AGGREGATOR_aggregate_metric	metric_id, aggregator_address
REDUCER_global_metric_reduce(all)	metric_id, reduction_operator, reducer_address

metric. Our current prototype limits metric sample updates to a single `DOUBLE` value along with an optional sample ID field. By default, the COLLECTOR uses the Argobots thread ID of the caller for the sample ID field.

4.8.2 Data Model and Metric API. SYMBIOMON employs a time-series data model to store and export performance metrics and exposes the corresponding API through the COLLECTOR microservice. A time series is a collection of *samples*. Table 28 depicts the key metric APIs available to the user for metric creation, aggregation, and reduction.

4.8.2.1 Metric Creation. During metric creation, the user must supply a *namespace* and a *metric name*. Since SYMBIOMON can simultaneously monitor multiple workflow components, namespaces are essential to differentiate metrics with the same name but originating from different sources. The other required field is a *metric type*. The metric types currently supported, along with a brief description of each, are presented in Table 16. These types are reflective of common practice in time-series monitoring systems.

The user can optionally "decorate" a metric with an arbitrary-length *taglist* of strings. For example, the unique microservice provider ID within a cohort is a good starting point for generating a taglist. Taglists are necessary when the user wants

Table 16. SYMBIOMON Metric Types

Metric Type	Description
COUNTER	Monotonically increasing value
TIMER	Monotonically increasing value representing a timer
GAUGE	Metric values that can increase or decrease over time

Table 17. SYMBIOMON Metric Reduction Operators

Reduction Operator	Description
SUM	Sum of all metric samples
MIN	Low-watermark value
MAX	High-watermark value
AVG	Mean value of all metric samples
ANOMALY	Metric samples that deviate from the mean by three σ

to aggregate metric data globally. The other optional metric field is a *reduction operator*. Table 17 presents a list of reduction operators that are currently supported. Reduction operators are meaningful only when the COLLECTOR is composed with an AGGREGATOR and a REDUCER microservice.

4.8.2.2 Metric Update. Traditional time-series databases employ a tuple of a timestamp and a value to denote each metric sample. As depicted in Table 28, we find that it is helpful to have an optional "sample ID" field (e.g., representing the thread's ID) for performing the metric update. The metric update is a local, fast operation; no RPC calls are made during the metric update operation.

A new metric value is inserted at the end of the buffer holding the metric data during the metric update. The user supplies the metric value and an optional sample ID when updating a metric through the COLLECTOR metric API.

4.8.2.3 Management of Metric Buffer Size. The COLLECTOR provides three options for the user to manage the buffer associated with a metric. First, one can allocate more space for the metric data when the metric buffer is full.

This strategy is the default behavior that we choose to evaluate this work. Second, the COLLECTOR will reset the metric buffer index to zero and overwrite the existing data. Third, the COLLECTOR will reset the metric buffer index to zero a reduction operation is invoked.

4.9 Case Studies Addressing the Performance Queries

This section presents case studies addressing each performance query raised in Chapter III. The background of the operation of specific Mochi microservices is presented before the case studies are discussed.

4.9.1 Mobject Composed Service.

Mobject [6] is a distributed object storage service that exposes a subset of the RADOS [175] API to support concurrent, noncontiguous writes of objects. Each Mobject *provider node* (service provider process) hosts three types of providers—a Mobject sequencer provider, a BAKE provider, and an SDSKV provider. The Mobject sequencer provider translates the RADOS operations into the underlying BAKE and SDSKV operations. BAKE stores object data through RDMA transfers between BAKE and client memory. SDSKV is used to store metadata information. We note that the Mobject provider is the client-facing provider, and control always goes back to the Mobject provider after the BAKE and SDSKV operations are complete. Figure 16 depicts this structure.

4.9.1.1 Query 1: Identifying Dominant Microservice Dependencies.

Identifying dominant microservice dependencies or callpaths is a crucial first step in performance analysis. It isolates resource-intensive portions of the workload at a high level and helps determine where to focus attention for further analysis and optimization. Recall that RPC callpaths can cross process boundaries. The SYMBIOSYS profile summary script ingests all the profiles and performs a global analysis to identify origin-target pairs for each callpath. The script summarizes

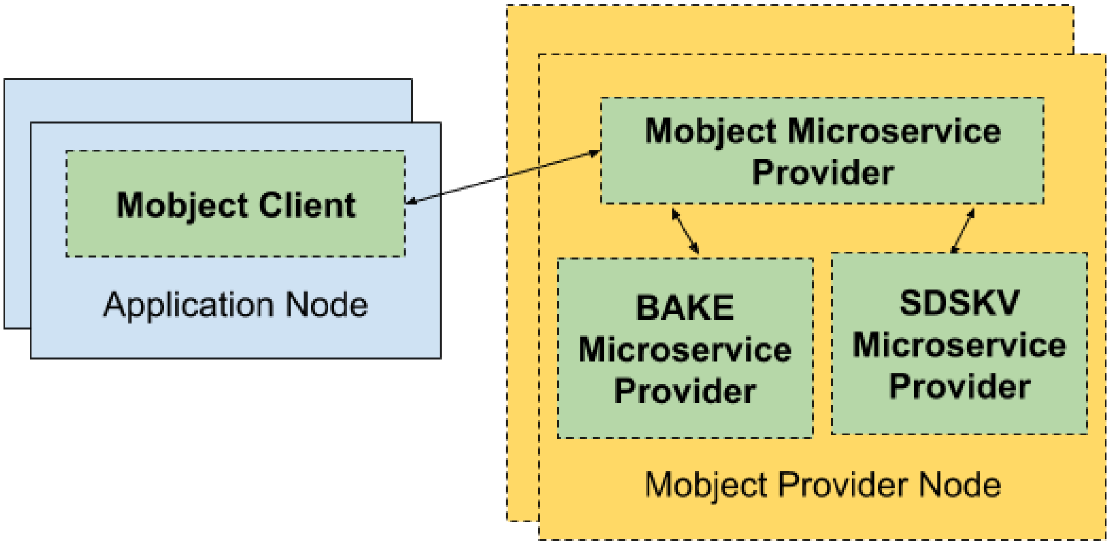

Mobject Client
Application Node
Mobject Microservice Provider
BAKE Microservice Provider
SDSKV Microservice Provider
Mobject Provider Node

Figure 16. Mobject Illustration

and sorts callpaths by cumulative end-to-end request latency to determine the most dominant ones. For each of these dominant callpaths, the SYMBIOSYS profile summary script generates call count distributions for all the participating origin and target entities. These distributed callpaths are used as a pivot around which lower-level communication library data and tasking library queue information is oriented. The results from this profile summary can be used as a starting point for a more detailed performance study.

We employ a single Mobject service provider node and ten ior [185] clients colocated on the same physical node. The ior benchmark has been modified to use Mobject for reading and writing objects. For this setup, Figure 17 depicts the top 5 most dominant callpaths by cumulative end-to-end request latency. mobject_read_op is the most expensive Mobject API operation overall. The profile suggests that the mobject_read_op⟶sdskv_list_keyvals_rpc is the dominant component of the top-level mobject_read_op API call. Note that for each of these callpaths, the breakdown of the individual steps for each callpath, such as the *input serialization time*, *internal*

RDMA transfer time, and *target handler time* is shown. For this application setup, these individual steps occupy a negligible time compared to the time taken to execute the request on the target.

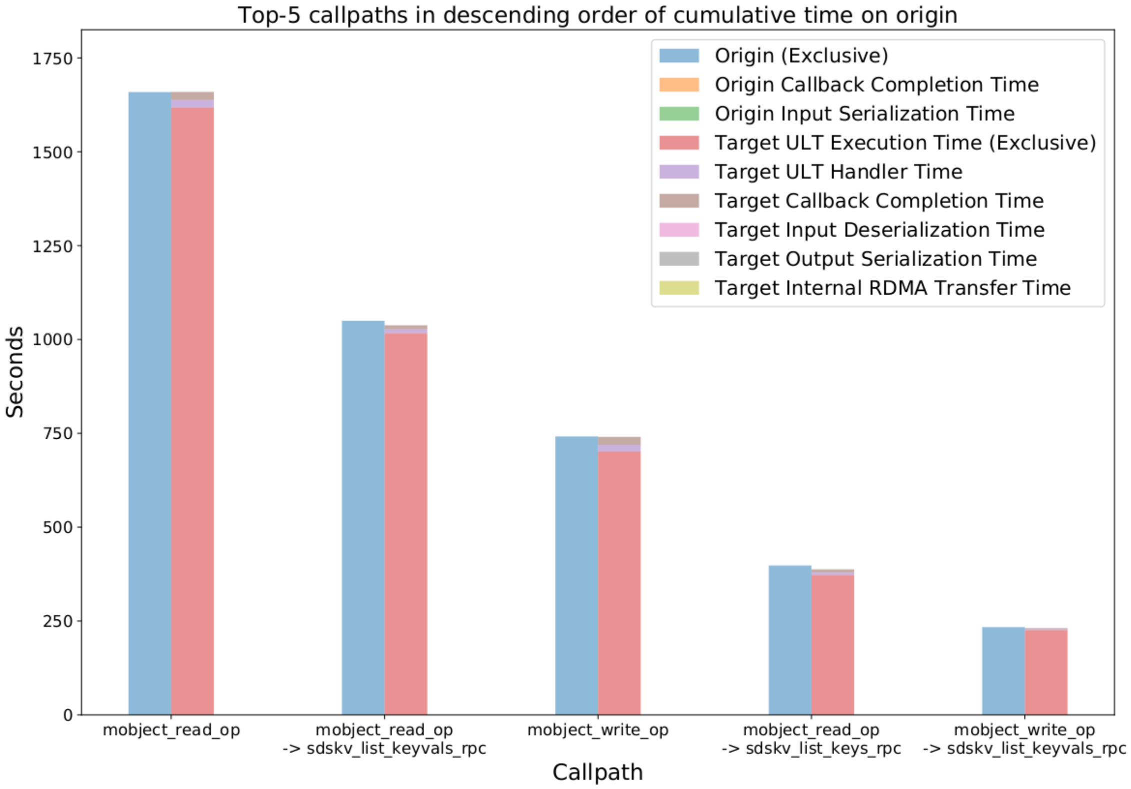


Figure 17. ior + Mobject: Identifying the Dominant Callpaths

4.9.1.2 Query 2: Detecting Load Imbalance.

While Figure 17 depicts the cumulative time for the top-5 callpaths resulting from the ior + Mobject execution, Figure 18 depicts the distribution of the call time across the origin entities for the same execution. Significant load imbalance is observed for the `mobject_read_op` RPC call time across the origin (MPI) processes. Note that this observation, although helpful in explaining the observed performance, does not directly yield additional information about the *cause* for the load imbalance.

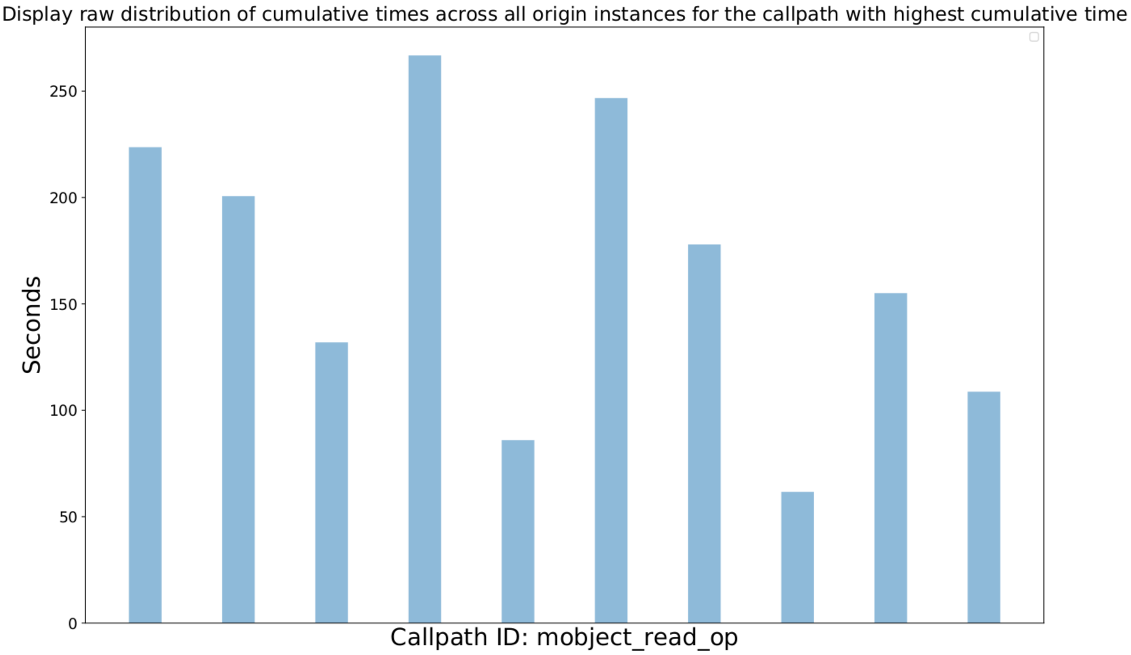
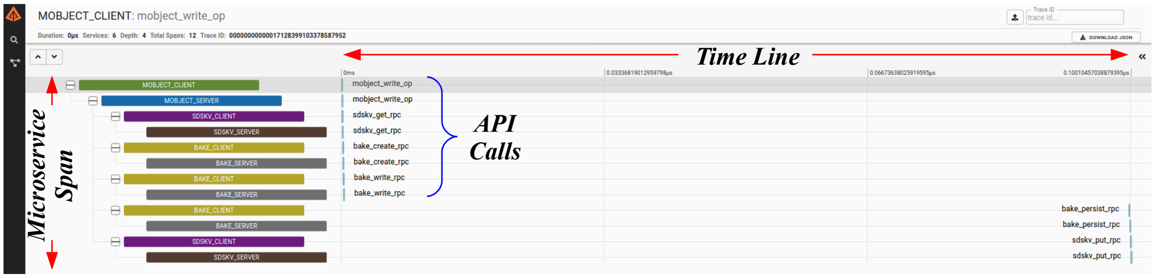


Figure 18. ior + Mobject: Raw Cumulative Distribution of Calltimes Among Origin Entities

4.9.1.3 Query 3: Discovering Individual Request Structure. Once the dominant callpaths have been identified, developers may be interested in tracing the path of individual requests to pinpoint the exact microservice operations getting invoked due to a higher-level operation. This sort of analysis is beneficial for identifying root causes for performance anomalies resulting from structural abnormalities in service requests.

For the same ior and Mobject setup described previously, Figure 19 represents the trace visualization for a single invocation of the mobject_write_op callpath. It discovers 12 discrete SDSKV and BAKE microservice calls (e.g., mobject_write_op⟶sdskv_get_rpc, mobject_write_op⟶bake_persist_rpc) that make up the higher-level mobject_write_op request. Each of these 12 discrete microservice calls has its profiling data, so the user can break down where time is spent and reason



MOBJECT_CLIENT: mobject_write_op

Duration: 0μs Services: 6 Depth: 4 Total Spans: 12 Trace ID: 00000000000017128399103378587952

Figure 19. ior + Mobject: OpenZipkin [2] Trace Visualization Depicting Discrete Steps for a Single mobject_write_op Request

about request performance. Without this trace data, the internal structure of the request is completely opaque to the user. SYMBIOSYS enables this Gantt chart visualization through an adapter module that "stitches" the events with a common requestID from different processes into a Zipkin [2] JSON trace file.

4.9.2 SONATA Microservice. Sonata is a microservice for remotely accessing and storing JSON objects. It is based on an UnQLite [188] database and offers the ability to run analysis remotely on the stored JSON objects through Jx9 scripts. While the BAKE microservice is optimized for large blobs of unstructured data, and the SDKSV microservice is optimized for small key-value pairs, Sonata is instead optimized for document storage, especially if there is a need to perform complex, in-place queries on these documents.

4.9.2.1 Query 4: Mapping RPC Resource Usage and Time. The JSON document to be stored is transferred as RPC metadata. However, if Mercury's *eager buffer* overflows, the additional RPC metadata is transferred through an internal Mercury RDMA operation (between $\mathbf{t_3}$ and $\mathbf{t_4}$). With a large RPC metadata transfer, it is imperative to understand the contributions of (de)serialization and internal RDMA transfer operations to the RPC execution time. We execute a simple Sonata benchmark with one target and one origin entity on separate compute nodes. The benchmark repeatedly invokes the `sonata_store_multi_json` API call to store a

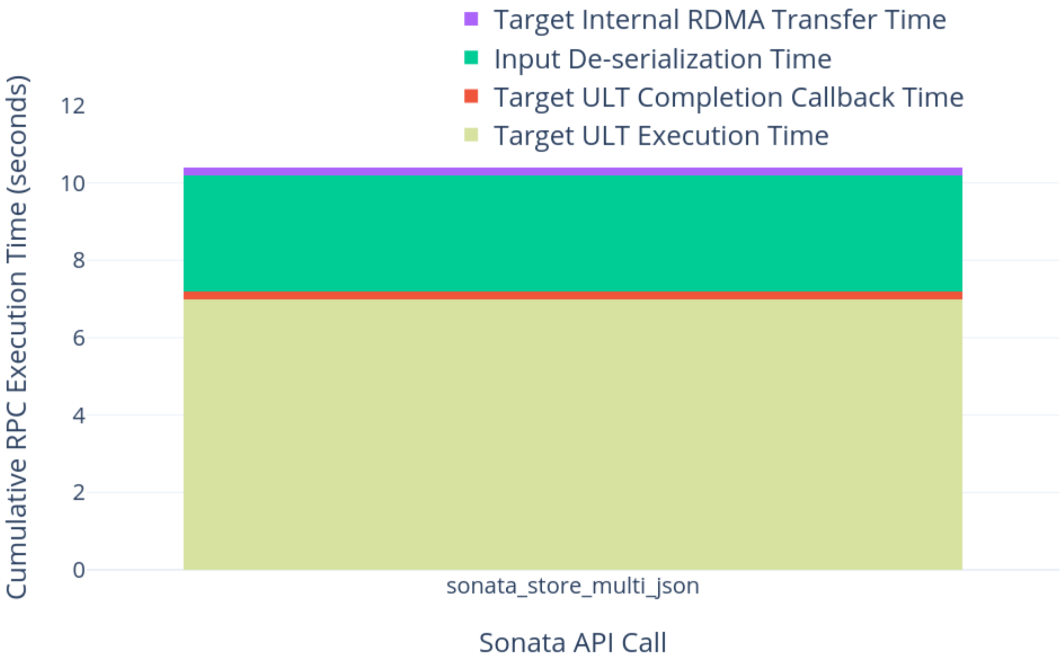


Figure 20. Sonata: Mapping Execution Time to Individual Steps

fixed-length JSON record array in a set of batches. The *batch size* benchmark parameter determines the size of RPC metadata and the total number of RPC calls. Figure 20 depicts the breakdown of the cumulative RPC execution time on the target for a JSON record array of 50,000 entries and a batch size of 5,000. While the *target internal RDMA transfer time* is relatively low, the time to de-serialize the input accounts for 27% of the overall execution time on the target.

4.9.2.2 HEPnOS Composed Service. HEPnOS [6] is a Mochi storage service designed for high-energy physics experiments and simulations at Fermilab. Data in HEPnOS is arranged in a hierarchy of datasets, runs, sub-runs, and events. Events correspond to serialized C++ data objects. HEPnOS distributes both object data and metadata. Each HEPnOS service provider node hosts several SDSKV providers to store event and product data. Figure 21 depicts the structure of the

HEPnOS service. Client processes (physics simulation) contact the SDSKV providers directly through a C++ client API.

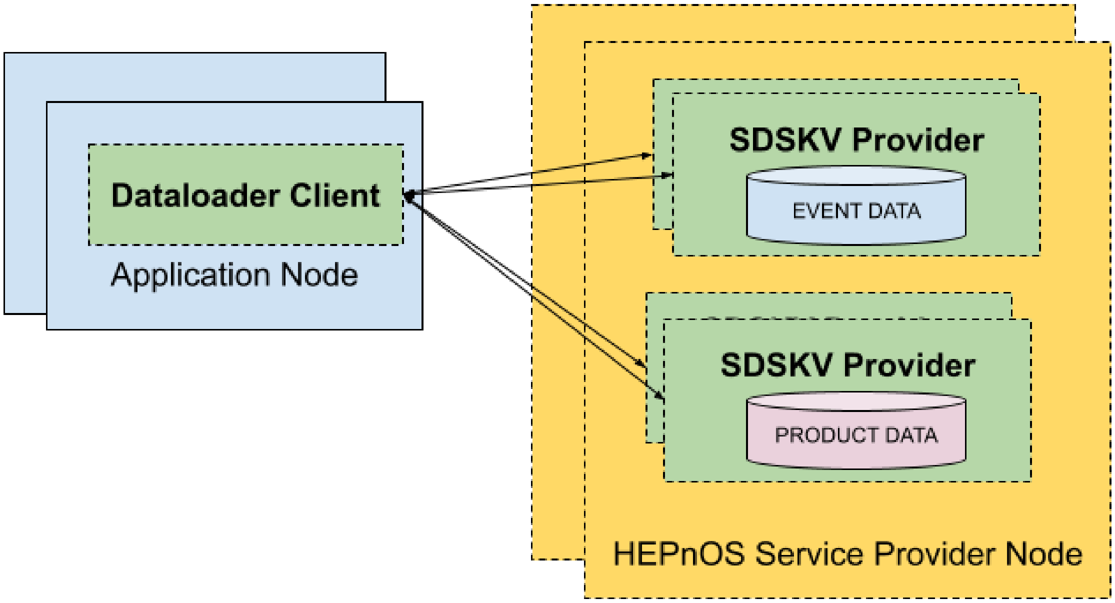


Figure 21. HEPnOS Illustration

The production HEPnOS client application is a workflow comprising multiple steps. This study focuses on the “data-loader” step in which particle event data is loaded into the system. We chose this step for evaluation since it is the most mature and has the fewest external dependencies. Client processes (MPI-based physics analysis tasks) contact the SDSKV providers directly using a C++ API. To minimize the RPC data transfer overheads, the data-loader client *batches* the event data using the `sdskv_put_packed` API. Specifically, event data is transferred to the service providers as a set of packed, key-value pairs.

The data-loader client application and the HEPnOS service are *tightly coupled* entities. Thus, the overall runtime of the data-loader is determined primarily by the configuration of the HEPnOS data service. The key high-level, tunable parameters affecting HEPnOS performance are as follows.

Table 18. HEPnOS: Service Configurations

Configuration	Total Clients; Clients Per Node	Total Servers; Servers Per Node	Batch Size	Threads (ESs)	Databases	Client Progress Thread?	OFI_max_events
$\mathbf{C_1}$	32; 16	4; 2	1024	5	32	✗	16
$\mathbf{C_2}$	32; 16	4; 2	1024	20	32	✗	16
$\mathbf{C_3}$	32; 16	4; 2	1024	20	8	✗	16
$\mathbf{C_4}$	2; 1	4; 2	1024	16	8	✗	16
$\mathbf{C_5}$	2; 1	4; 2	1	16	8	✗	16
$\mathbf{C_6}$	2; 1	4; 2	1	16	8	✗	64
$\mathbf{C_7}$	2; 1	4; 2	1	16	8	✓	64

- Batch Size: Amount of work passed through the system. Clients batch RPC requests to improve throughput. A low value of batch size is expected to affect performance negatively. Note that batching is performed at the HEPnOS application level and is not a feature offered natively by the Mochi stack.

- Threads: Number of Argobots execution streams per service provider node.

- Databases: Number of databases to store events and products per service provider node. Increasing the number of databases does not always lead to improved performance. The event data is spread over many databases, potentially resulting in smaller effective batch size.

- Service Topology: Number of server processes deployed on a single computing node.

4.9.3 Query 5, 7: Observing Resource Saturation and Identifying a Better Service Configuration. SYMBIOSYS is employed to study the root causes of poorly performing HEPnOS configurations and to determine better service configurations to improve performance. Table 18 enlists the various service configurations that are a part of this study.

4.9.3.1 Too Few Execution Streams. It is difficult to estimate the optimal number of target Argobots execution streams (ESs) required for a given workload. However, by observing delays in the execution pathway of the RPC call, one can determine when these resources saturate, thereby identifying a poorly performing service configuration. Recall that on the target, a new ULT is spawned at $\mathbf{t_4}$ for every incoming RPC request. We define the delay between events $\mathbf{t_4}$ and $\mathbf{t_5}$ in Figure 8 as the *target handler time*. A newly spawned ULT spends this portion of time in the Argobots handler pool before an ES can pick it up for execution. When the target is overloaded with RPC requests and lacks the execution resources to dispatch the corresponding ULTs promptly, the *target handler time* can contribute significantly to the overall request latency and worsen performance.

Figure 22 demonstrates that $\mathbf{C_1}$ suffers from a lack of execution resources on the target. Avoidable delays inside the Argobots handler pool (target handler time) account for 26.6% of the total RPC execution time.

A Better Service Configuration: $\mathbf{C_2}$ remediates this by adding 15 additional execution streams (threads). Overall cumulative RPC execution time improves by 53.3%, with the target handler time contributing 14% to the overall time.

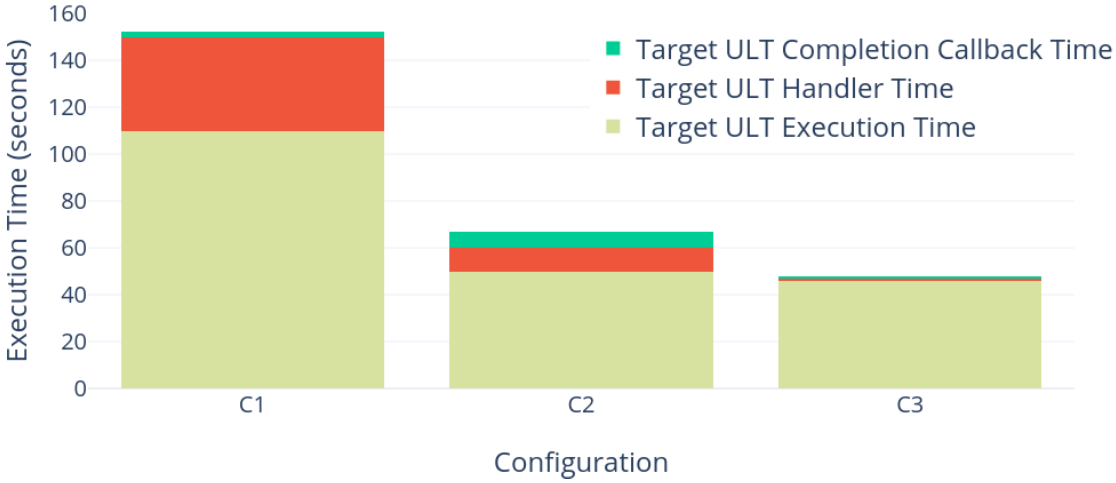
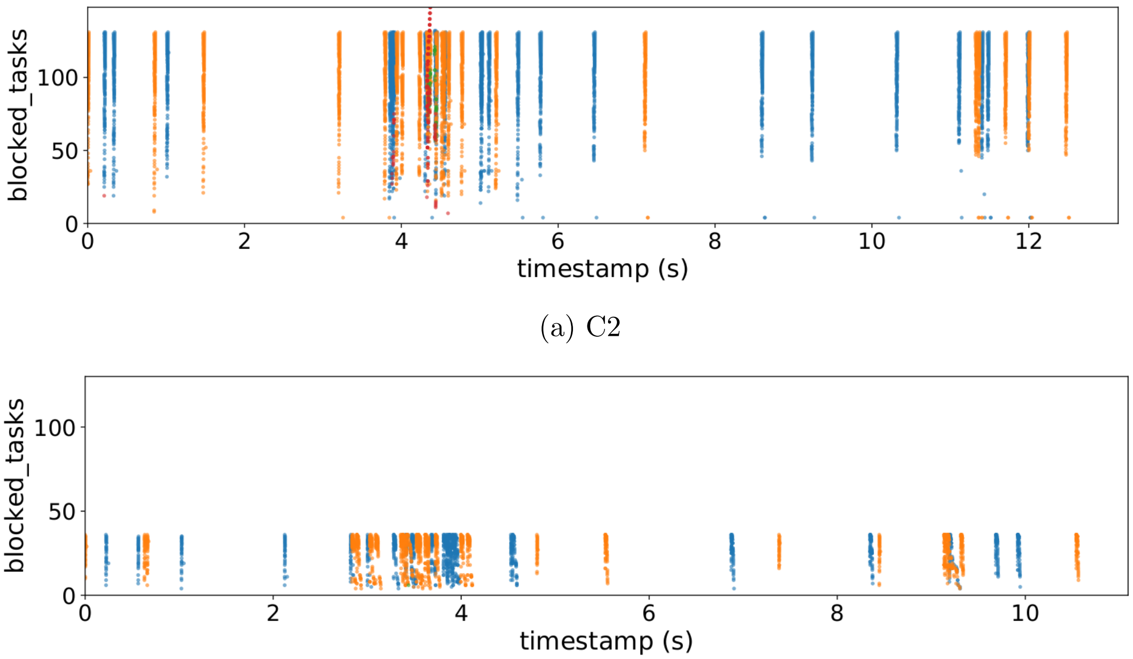


Figure 22. HEPnOS: Cumulative Target RPC Execution Time for sdskv_put_packed

4.9.3.2 Too Many Databases. Each target provider node employs several databases to parallelize the writing of HEPnOS event data. Specifically, in this study, the target utilizes a `map` backend. For the sdskv_put_packed RPC, the origin implements a hashing scheme using the key and the total number of databases to identify the target database ID. All else being equal, the greater the number of target databases, the greater the number of RPCs generated. Since the `map` backend is not capable of parallel insertions, employing too many target databases can create a flood of RPCs and result in write serialization during bursty behavior. Configuration $\mathbf{C_2}$ suffers from this problem. The x-axis in Figure 23 denotes the timestamp of when the request began execution on the target at $\mathbf{t_4}$ (Figure 8), and the y-axis represents the total number of blocked ULTs sampled from Argobots at this time. Each colored dot represents a single request. Different colors represent requests executed at different targets. Figure 23a depicts this serialization problem during bursty behavior with configuration $\mathbf{C_2}$. This pattern of vertical lines generated by requests that arrived simultaneously but finished in quick succession (as opposed to simultaneously) indicates serialization on a backend resource.

A Better Service Configuration: Counterintuitively, RPC performance in this situation improves when reducing the number of databases. RPC performance in $\mathbf{C_3}$ is better than $\mathbf{C_2}$ by 28.5%. Figure 23b also demonstrates that the severity of serialization in $\mathbf{C_3}$ is much reduced as compared with $\mathbf{C_2}$. The reduced number of RPCs generated with $\mathbf{C_3}$ also has the effect of lowering the target handler time and the target completion callback time—the ULTs are being processed quickly without introducing unwanted delays.

4.9.3.3 Effect of a Low Batch Size on Client Progress. HEPnOS clients batch key-value pairs containing HEPnOS event data to improve RPC throughput when generating an sdskv_put_packed request. A batch size of 1,024 ($\mathbf{C_4}$) is roughly 475 times more performant than a batch size of 1 ($\mathbf{C_5}$). Figure 24 suggests that instrumentation from the RPC API and RPC library layers is insufficient



(a) C2

(b) C3

Figure 23. HEPnOS: Sampling Blocked Tasks from Argobots for sdskv_put_packed

to capture all components of the cumulative RPC execution time for $\mathbf{C_5}$ (the unaccounted portion is depicted by the blue color in Figure 24). We consider the question of identifying this gap in instrumentation. We also seek to improve RPC performance when low batch size is an inherent property of the application setup.

The HEPnOS data-loader client employs a Mercury *progress ULT* to progress RPC communications within the client process. This progress ULT has two important tasks: (1) read the OFI events containing notifications of RPC responses from the network abstraction layer and add the corresponding completion callbacks to the completion callback queue and (2) trigger completion callbacks from the completion callback queue.

To prevent context switching overheads, this progress ULT is executed within the context of the main Argobots execution stream that also runs the ULTs issuing

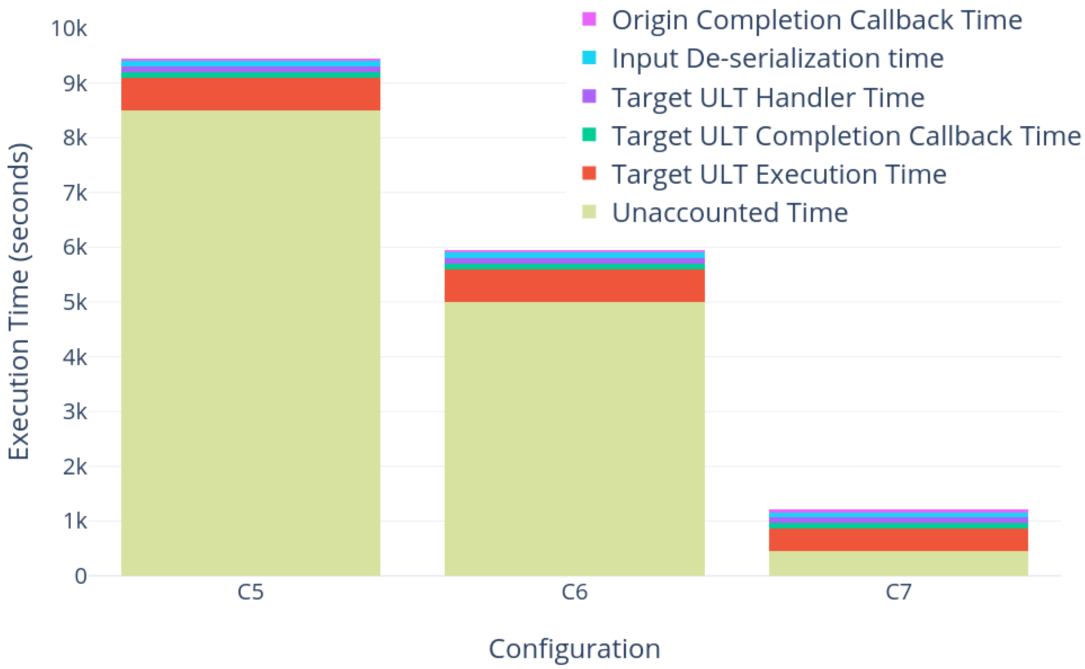
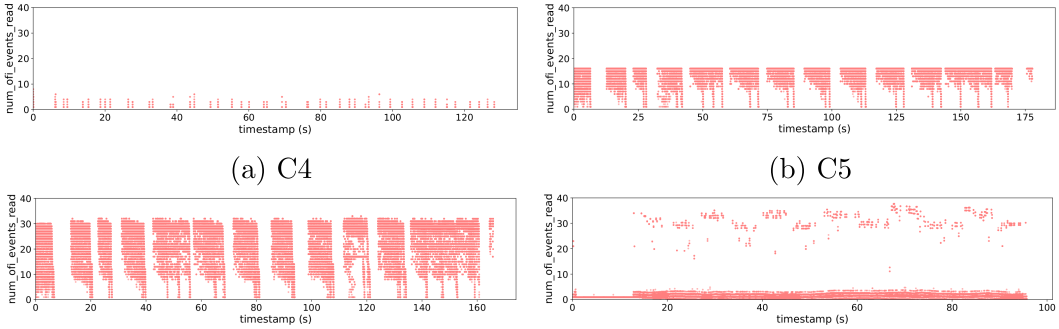


Figure 24. HEPnOS: Unaccounted Component of RPC Execution

RPC requests. When the batch size is small, the progress ULT can compete for CPU resources with the ULTs issuing RPC requests. As a result, the completion callback queue or the OFI event queue can clog up and introduce unwanted delays. Every time it is scheduled to execute, the progress ULT reads up to a maximum of `OFI_max_events` events from the OFI interface. `OFI_max_events` has a default user-defined value of 16, set inside the Mercury library. Figure 25a depicts a sample of the `num_ofi_events_read` PVAR in configuration $\mathbf{C_4}$ where the batch size is optimal. In this configuration, the `OFI_max_events` threshold is never breached, implying that the OFI completion queue is emptied at regular intervals. Figure 25b depicts a sample of the same PVAR in configuration $\mathbf{C_5}$ where the batch size is low. The number of OFI events read consistently breaches the threshold value of 16, suggesting that the completion queue is backed up.

A Better Service Configuration: Increasing the `OFI_max_events` threshold from 16 to 64 with $\mathbf{C_6}$ improves RPC performance by over 40% while reducing the unaccounted time by 47%. The performance data gathered from various layers in the software stack indicate that employing a separate, dedicated execution stream for the client's progress thread is likely to improve performance. Figure 24 confirms that RPC performance for $\mathbf{C_7}$ improves by a further 75% and the unaccounted time reduces by a further 90% as compared with $\mathbf{C_6}$. From Figure 25d, we conclude that the OFI event queue is no longer backed up.



(a) C4

(b) C5

(c) C6

(d) C7

Figure 25. HEPnOS: Sampling OFI Events Read from Network Abstraction Layer for sdskv_put_packed

4.9.3.4 Query 6, 7: Improving Key-value Store Performance Under Concurrency.

It is difficult to estimate the correct number of HEPnOS service providers and database instances per service provider without knowing the client workload characteristics. Recall that a provider spawns a new Argobots ULT to service an incoming RPC. When the provider is servicing multiple requests simultaneously, the ULTs can contend for database access. When an SDSKV provider, for example, is provisioned with a less-than-optimal number of database resources, the ULTs requesting concurrent access can get queued up, leading to delays and an overall poor data service performance.

This problem is especially pronounced when the database access is serialized, that is when only one ULT gets exclusive access to the database at any point in time. Here we use SYMBIOMON to understand how concurrency affects SDSKV database performance. Specifically, we focus on the `sdskv_put_packed` API under concurrent access while keeping all the other performance-related factors fixed. The database write operation delineates a *candidate critical section* around it. Note that the actual database backend implementation underneath determines whether multiple threads are allowed simultaneous access to the candidate critical section. SDSKV is instrumented with the COLLECTOR metric API to generate a time-series for the following GAUGES.

– num_entrants: This metric denotes the number of ULTs currently inside a candidate critical section. When a ULT enters the section, this metric is incremented. When the ULT exits, this metric is decremented.

– latency: This metric stores the latency of the database operation.

– data_size: This metric captures the total amount of data written.

– batch_size: This metric denotes the batch size (number of keys) written as part of the put_packed operation.

Figure 26 depicts a trace of the SDSKV num_entrants metric during a typical execution of the data-loader client with the HEPnOS service. There are bursts of time when the database is busy (num_entrants is more than zero), and there are periods when the database is free (indicated by a zero value for the num_entrants metric). In this context we define a *concurrency region* as the period of execution between two such zeros. We are particularly interested in studying the database performance inside these concurrency regions.

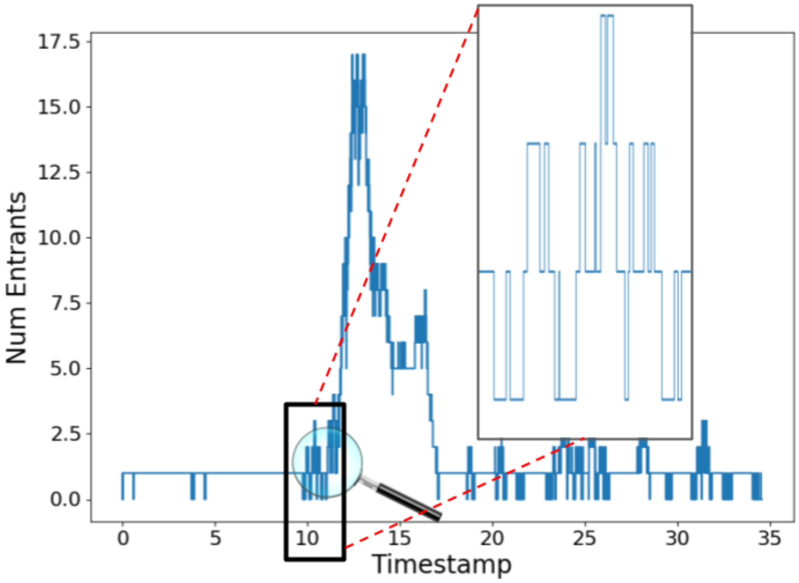


Figure 26. SDSKV: "num_entrants" Metric and Concurrency Regions

4.9.3.5 Establishing a Performance Baseline. A baseline performance model must be established before time-series data is analyzed to generate a better service configuration. We use this data to generate a multiple linear regression model of serial performance. A single-client, single SDSKV server setup is employed to observe the latency for the `sdskv_put_packed` operation for various data sizes and batch sizes.

The fundamental premise for our analysis is that the performance of a `sdskv_put_packed` call under concurrency must be *at least as good as*, if not better than, the baseline serial performance. For database backends that do not support concurrent write access (such as `std::map`), performance under concurrent access can be *no better* than serial performance. Regardless of the type of database backend employed, we aim to generate a better service configuration that is at least as good as serial performance, if not optimal.

4.9.3.6 Identifying Performance Bottlenecks. We fix all the other input parameters and identify a better service configuration by resolving performance issues related to concurrent database access. Algorithm 1 describes this process. The algorithm inputs the time-series data for the four metrics of interest described previously and the regression model of serial performance. It generates the *average latency dilation ratio* (ALDR) as the output. The ALDR metric measures the factor by which the average request is dilated because of concurrency. When performance is at least as good as serial performance, the ALDR should be close to one. A higher value for the ALDR implies a relatively poor service configuration.

Algorithm 1 ingests the time-series data and generates a list of concurrency regions in the execution. A list of threads executing inside each concurrency region is generated. Note that the sample ID field enables this within the metric sample. Recall that the Argobots thread ID is employed as the sample ID by default. The data size and the batch size of the `sdskv_put_packed` operation associated with each thread are used to predict the baseline latency value (using the regression model). The actual latency is divided by this baseline latency value to generate the dilation factor for the thread. In this manner, the ALDR value is calculated and returned as the output.

4.9.3.7 A Better Service Configuration. The ALDR metric value is directly employed to generate a more optimal database configuration as a simple heuristic. For example, suppose the ALDR is two. In that case, the user is recommended a new service configuration in which the total number of databases in the system is approximate twice the previous value. There are two ways to increase the number of databases in the system. The first method is to increase the number of SDSKV databases per HEPnOS service provider. The second is to increase the

Algorithm 1 Latency Dilation Due to Concurrency

```
Input: Time series data (all service providers), regression model
Output: Average latency dilation ratio for the entire execution
procedure LatencyDilationRatio
    $C \leftarrow$ SetOfConcurrencyRegions
    $TT \leftarrow$ TotalNumberOfThreads
    $TT \leftarrow 0$
    $TDR \leftarrow$ Total Latency Dilation Ratio
    $TDR \leftarrow 0$
    $ALDR \leftarrow$ Average Latency Dilation Ratio
    $ALDR \leftarrow 1$
    for each $c$ in $C$
        $T \leftarrow$ ThreadsInConcurrencyRegion
        $TT \leftarrow TT + Size(T)$
        for each $t$ in $T$
            $l \leftarrow operation\ latency$
            $b \leftarrow batch\ size$
            $d \leftarrow data\ size$
            $baseline\_latency \leftarrow regression\_model.predict(b,\ d)$
            $ldr \leftarrow l/baseline\_latency$
            $TDR \leftarrow += ldr$
    $ALDR \leftarrow TDR/TT$
return $ALDR$
```

number of HEPnOS service provider instances by running more providers on the same set of resources. We choose to use the latter method in our experiments. Note that the number of threads for each service provider process was set to 16, allowing for a maximum of 4 service provider processes per KNL node. Algorithm 1 is repeatedly invoked until the ALDR ratio approaches the value of one. Figure 27 depicts this process. While this algorithm can be employed to reconfigure the service online, we use an offline analysis of the time-series data for our case study.

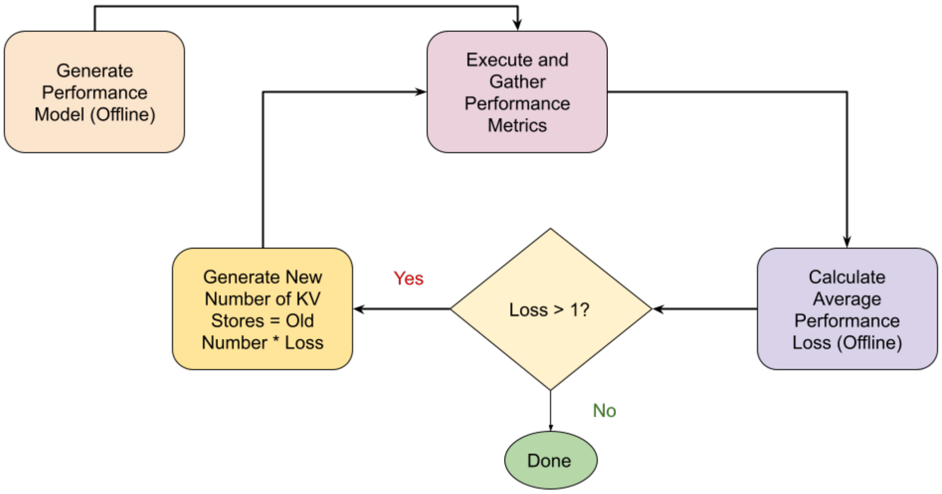

Generate Performance Model (Offline)
Execute and Gather Performance Metrics
Generate New Number of KV Stores = Old Number * Loss
Yes
Loss > 1?
Calculate Average Performance Loss (Offline)
No
Done

Figure 27. HEPnOS: Iteratively Improving Data Service Performance Under Concurrency

Algorithm 1 is applied to a large-scale HEPnOS setup on the Theta[1] machine at the Argonne Leadership Computing Facility (ALCF). Theta is a CrayXC40 system that hosts 4,393 Intel Knights Landing compute nodes, each of which supports 64 CPU cores. Execution time of the data-loader client application is employed as a measure of performance. The performance improvements from applying Algorithm 1 to a set

[1] https://www.alcf.anl.gov/alcf-resources/theta

Table 19. HEPnOS: Generating Better Service Configurations

Configuration	#Clients / Client Nodes	#Databases / Databases Per Service Provider	ALDR	Execution Time (s)
$\mathbf{C_{1,\,0}}$	112 / 1	64 / 16	1.616	336
$\mathbf{C_{1,\,1}}$	112 / 1	96 / 16	0.751	123
$\mathbf{C_{2,\,0}}$	224 / 2	128 / 16	1.836	195
$\mathbf{C_{2,\,1}}$	224 / 2	256 / 16	0.876	139
$\mathbf{C_{3,\,0}}$	448 / 4	256 / 16	2.063	114
$\mathbf{C_{3,\,1}}$	448 / 4	512 / 16	0.808	67

of service and client configurations are described in Table 19. $\mathbf{C_{i,\,j}}$ represents the j^{th} iteration of Algorithm 1 on configuration $\mathbf{C_i}$. $\mathbf{C_{i,\,0}}$ is the default configuration. Note that for a given $\mathbf{C_i}$, the total number of clients (TC) remains the same. At the same time, the algorithm attempts to identify the optimal total number of database service providers (TD) for the client workload generated by TC. In addition, the number of database service providers per node (DSP) remains constant across configurations, implying that a higher TD value is associated with a larger node allocation for service providers.

The service and client processes are spread across 128 compute nodes on the HPC machine. Note that the batch size (8192), the number of threads (16), and the number of SDSKV databases (with `std::map` backend) per HEPnOS service provider (16) are kept fixed across all the configurations. Performance improves by up to 65% between the default and optimized service configurations with identical client workloads. An astute reader would note that the ALDR value is less than one, given that `std::map` does not allow concurrent writes. We attribute this to "holes" in the training data used to generate the regression model and to performance variation in the system. Future work will aim to employ additional training data to create a model of baseline performance.

4.10 Overhead Analysis

This section presents the overheads in employing the SYMBIOSYS and SYMBIOMON tools for performance measurement and analysis of the HEPnOS storage service. We pay special attention to the process of separating the overheads of adding instrumentation, making the measurement, and analyzing the generated performance data.

4.10.1 Setup.

4.10.1.1 Hardware. All the experiments were conducted on the Theta [2] system at the Argonne Leadership Computing Facility (ALCF). Theta, a CrayXC40 system, hosts 4,393 Intel KNL compute nodes, each of which hosts 64 processing cores. We used the Intel KNL processors for all our experiments.

4.10.1.2 Software. Our experiments were conducted using the HEPnOS storage service and a data-loader client application setup. The Mochi components were installed using the Spack [189] package manager. We employed 128 nodes for our large-scale study.

4.10.2 SYMBIOSYS Overhead Study. We used 32 HEPnOS service provider processes spread evenly over 16 nodes. Each service provider process was assigned 30 threads and 16 databases for storing HEPnOS events. We employed 224 data-loader clients spread over 112 nodes. The batch size was set to 8,192, and a separate client progress thread was not used for our experiments. We measured the execution time of the data-loader application as the metric to compare the instrumentation and measurement overheads in SYMBIOSYS. We used the following terms to denote the various stages of the process:

[2] https://www.alcf.anl.gov/support-center/theta

- Baseline: This is the baseline execution time with instrumentation and measurement disabled.

- Stage 1: This is the execution time with instrumentation turned on while no measurements are made. In SYMBIOSYS, this corresponds to the addition of RPC callpath and trace ID information in the RPC request.

- Stage 2: Callpath profiling, tracing, and system statistic sampling are enabled, but Mercury PVAR collection is disabled.

- Full Support: Callpath profiling, tracing, and system statistic sampling are enabled. Mercury PVAR collection is turned on, and the PVAR data is integrated on the fly with the callpath profiles.

Figure 28 depicts the overheads involved in enabling various stages of performance measurement using the HEPnOS setup. Each entry in the table is the average of 5 execution times.

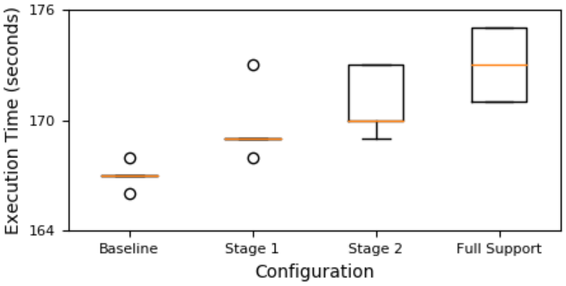


Figure 28. HEPnOS: SYMBIOSYS Measurement Overheads

The SYMBIOSYS tracing system collected 1 million samples on a large scale. Even at this scale, enabling profiling and tracing led to minimal overheads of less than 4% of the total workflow execution time. Table 20 presents the time taken to analyze the collected performance data and generate visual plots. The profile and

Table 20. HEPnOS: SYMBIOSYS Analysis Overheads

Profile Summary (s)	Trace Summary (s)	System Statistics Summary (s)
35.1	481.1	73.4

Table 21. HEPnOS: SYMBIOMON Measurement Overheads

Configuraton	Minimum (s)	Maximum(s)	Average (s)
Without COLLECTOR	111	118	116
With COLLECTOR	106	118	116

system summary analysis scripts took a short amount of time to complete, while the trace summary script took a long time to run when applied to the large-scale performance data. The trace summary overhead can be reduced by employing the sampling support within the trace summary module.

4.10.3 SYMBIOMON Overhead Study. We present a study of the overhead of employing the SYMBIOMON COLLECTOR component to instrument the HEPnOS data service at scale. The HEPnOS storage service is instrumented to extract the data size, batch size, latency, and num_entrants for the `sdskv_put_packed` API.

We employed 128 nodes for our overhead study. We used 224 data loader clients, spread evenly across 112 nodes, and 32 HEPnOS service providers, spread evenly across the remaining 16 nodes (a known good configuration). We assigned 32 threads to each HEPnOS service provider and fixed the batch size at 8,192. The execution time of the data-loader application was utilized as a measure of performance. Table 21 depicts the execution time for the setup with and without the use of COLLECTOR instrumentation. The results suggest that the COLLECTOR introduces no noticeable overhead.

4.11 Limitations

While SYMBIOSYS represents a significant first step toward generating optimal HPC service configurations, it is worthwhile to discuss the limitations of this approach:

- Performance Observation in a Shared-service Setting: The case studies presented in this chapter involve the coupling of an MPI application (workflow component) with a *dedicated* service instance. The flexibility resulting from a microservice architecture allows multiple clients to share access to microservices. The Mochi framework indeed provides the support for a shared-service model to be configured and deployed. In this scenario, the performance observation framework would be required to discern the performance of the service components as they relate to the specific client accessing the service. If employed in a shared-service setting in its current state, SYMBIOSYS would generate an aggregated profile that would not be able to map the performance observations to the individual clients making the requests. To rectify this, an additional *client identifier* would need to be appended and passed along with the existing RPC ancestry information in the request. The offline analysis module would also need to be appropriately modified to reflect this change.

- Performance Data Exchange with Argobots Concurrency Layer: The performance data exchange strategy presented in Section 4.5 primarily deals with eliciting the critical events from within the Mercury RPC layer. Presently, the Argobots API is queried for basic information about the state of the RPC queues and the number of blocked tasks. Extending the performance data exchange layer to support Argobots would enable a deeper insight into the lifecycle of the Argobots ULT servicing the RPC request and the status of the locks and other synchronization objects it acquires and releases during its execution.

4.12 Summary

To address research question **RQ1**, Chapter IV presented the SYMBIOSYS integrated performance measurement and analysis infrastructure designed for HPC data services. SYMBIOSYS enables effective and portable profiling and analysis of HPC microservices by tracking the RPC callpath ancestry. By integrating data from the RPC communication library through a data-exchange strategy, SYMBIOSYS correctly attributes low-level events and resource usage levels with higher-level interactions between service entities. This chapter presented the COLLECTOR microservice, which helped improve the microservice operation under concurrency. The analysis of the performance observations generated by the SYMBIOSYS and SYMBIOMON tools enables the identification of a better data service configuration. This analysis helped improve the coupled application-service workflow's performance (execution time) by up to 3x while operating with a maximum runtime overhead of 4% of the total workflow execution time. Figure 29 summarizes the key takeaways from Chapter IV:

- A combination of performance data from different techniques yields performance observability of HPC microservices.
- The solutions corresponding to individual techniques may already be proposed in some form by different communities — the challenge involves understanding how to adapt and apply these techniques for HPC microservices.
- Statistical modeling generates an understanding of the expected performance under ideal operating conditions. This dissertation finds the combination of

statistical modeling with the performance data useful for generating better-performing service configurations offline.

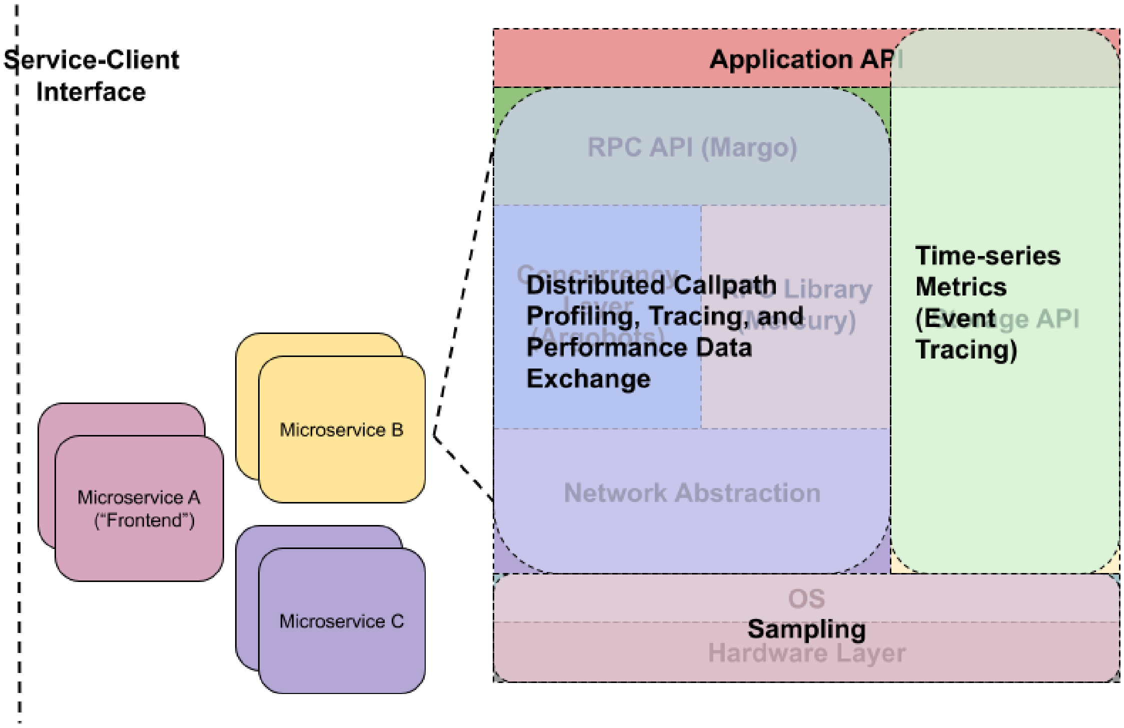

Service-Client Interface
Application API
RPC API (Margo)
Concurrency Layer (Argobots)
RPC Library (Mercury)
Distributed Callpath Profiling, Tracing, and Performance Data Exchange
Time-series Metrics (Event Tracing)
Storage API
Network Abstraction
OS
Sampling
Hardware Layer
Microservice A ("Frontend")
Microservice B
Microservice C

Figure 29. Selecting and Combining Instrumentation Techniques

While SYMBIOSYS enables a better service configuration through offline analysis, dynamic (online) service reconfiguration requires performance data for analysis and control to be accessible quickly from anywhere in the system and available to a remote user or entity for external, interactive analysis. Further, since these HPC services operate inside a workflow constituting other components such as ML tasks or MPI applications, a ubiquitously applicable performance monitoring solution is required. Chapter V strives to present a solution to this problem.

CHAPTER V

UBIQUITOUS MONITORING OF SERVICES AND IN SITU WORKFLOW COMPONENTS

This chapter contains previously published and unpublished material with co-authorship. Section 5.2, Section 5.4.1, Section 5.4.2, and Section 5.5.2 represent research conducted as a collaboration between the University of Oregon and Argonne National Laboratory. This research was published as a full paper at HiPC 2021 [16], a poster at HiPC 2021 [17], and a poster at SC 2021 [18]. For each of these publications, I was the first author and implemented the software, conducted the experiments, and wrote all the sections for the papers and posters, with suggestions and text edits by Dr. Philip Carns and Dr. Robert Ross. All the co-authors helped with proofreading.

Section 5.3 and Section 5.5.1 represent research that was conducted as a collaboration between Dr. Allen Malony, Dr. Sameer Shende, Dr. Kevin Huck, Dr. Nick Chaimov, and me. This research was published as a full paper at ICPP 2019 [21], where I was the second author. I implemented the plugin system in TAU with guidance from Dr. Sameer Shende and Dr. Kevin Huck. I implemented all the plugins described in the paper and conducted all the experiments except those involving the SOS plugin developed by Dr. Kevin Huck. Dr. Allen Malony wrote the introduction, related work, and background sections for the ICPP paper, while I wrote the plugin implementation, usage scenarios, and evaluation sections. The plugin design section was written by both Dr. Allen Malony and me (equal contribution). All the authors helped in proofreading the paper.

Section 5.4.3 represents unpublished work resulting from a collaboration between the University of Oregon, Brookhaven National Laboratory, and RUTGERS University. I implemented the software, conducted the experiments, and wrote this section. While performing this research, I received regular guidance from Dr. Matteo Turilli, Dr. Shantenu Jha, Dr. Tan Li, Dr. Mikhail Titov, and Dr. Allen Malony.

5.1 Introduction

HPC services do not operate in isolation — instead, they execute as a part of an in situ workflow consisting of other distributed components that include MPI applications, visualization routines, and ML tasks. Therefore, their performance must also be presented and analyzed in the context of the other workflow components. Further, these user-level services are transient entities — they are launched as a part of the batch job, execute for the duration of the workflow and are shut down once the workflow completes. While the SYMBIOSYS tool presented in Chapter IV enables offline analysis of service performance and is certainly useful to generate better service configurations for a static client workload, enabling the online adaptivity of these transient services requires a performance monitoring solution that can export, analyze, and make available some or all of these observations. Chapter V presents a solution to the challenges posed by these requirements:

- **Challenge**: Managing Large Data Volumes: Performance monitoring of many independent entities can result in trace volumes that can quickly grow to be intractable. The SYMBIOMON monitoring service, presented in Section 5.2 addresses this problem by exposing a set of reduction operators for the trace data. SYMBIOMON *decouples* the instrumentation and measurement components from the components responsible for the storage, analysis, and remote monitoring of this data. Specifically, these components are high performance microservices built using the Mochi framework. This methodology of building a monitoring service out of high performance microservice components has four distinct advantages: (1) it allows easy integration for monitoring Mochi data services, (2) SYMBIOMON can directly leverage performance improvements made to the core Mochi software stack, (3) SYMBIOMON's composable design enables its constituent components to be "toggled" as necessary, allowing for various modalities to be realized, and (4) it is arguably more maintainable than monitoring services built in an ad-hoc manner.

- **Challenge**: Ubiquitous Monitoring: The goal of ubiquitous monitoring involves using the *same* components to monitor a plethora of different types of workflow entities. The challenge is to seamlessly integrate monitoring capabilities while simultaneously leveraging existing tools that perform instrumentation and measurement. This dissertation proposes to use a plugin approach to address this challenge. Plugins, as described in Section 5.3 are attractive because they serve as a gateway between the source of the performance data (measurement tool) and the destination for this data (monitoring system). Further, plugins can host the "glue" code hosting the connection logic to the service entities, thereby enabling seamless integration of monitoring capabilities without requiring a significant code change to the application.

In addressing these challenges, Chapter V answers the research question **RQ2**: How to enable the ubiquitous monitoring of HPC applications, services, and workflows alike?

5.2 SYMBIOMON: A Composable Service for Monitoring and Analysis

A systematic approach to generating a better service configuration involves the ability to characterize a client workload, observe the performance of the service under the workload, decide whether a better service configuration exists, and implement the service reconfiguration through a control system. Dynamic (online) service reconfiguration requires performance data for analysis and control to be accessible quickly from anywhere in the system and be available to a remote user for external, interactive analysis.

To this end, we propose SYMBIOMON, a metric-monitoring service built by composing high-performance microservices. The monitoring service consists of three core microservice components—a COLLECTOR microservice that exposes the metric collection API, an AGGREGATOR key-value store for aggregation and storage, and a REDUCER microservice to implement global reduction operations. While SYMBIOMON is intended primarily to self-monitor Mochi data services, its utility as a general-purpose, scalable monitoring service for traditional HPC applications is also demonstrated. Time-series databases are beneficial in situations that require *changes to be tracked over time.* Time-series data is utilized in cloud-based service monitoring systems to perform forecasting, variability, and trend analysis. We argue that the monitoring of HPC applications can also benefit from this approach. For example, detecting anomalies in function execution times requires storing and analyzing snapshots of pertinent performance data over a time interval. SYMBIOMON implements a flexible time-series data model that allows users to export metrics with custom taglists to "decorate" the metric data.

5.2.1 Related Work. The applications and services to be monitored on HPC systems execute in the context of ephemeral batch jobs. As a result, monitoring services must be spun up quickly without any elevated privileges. Further, the monitoring service is expected to yield high-resolution time-series data and run efficiently in a highly concurrent environment. This high-resolution data also needs to be managed because storing every single data point at scale is not feasible. This section presents our contributions in the context of existing monitoring approaches.

5.2.1.1 Cloud-Based Service Monitoring Tools. The SYMBIOMON metric data model is inspired by time-series monitoring databases used in the cloud industry, such as Prometheus [19], Graphite [170], and InfluxDB [180]. Cloud-based monitoring frameworks typically are employed to extract data over coarse-grained time intervals (order of seconds). The services they monitor are long-running, are spread over a large geographical region, and run on top of a commodity hardware and software stack. Services in the cloud are written in various languages, and thus cloud-based monitoring frameworks offer rich, multilanguage client instrumentation support. Importantly, these monitoring frameworks are not set up to directly enable fast, dynamic service reconfiguration. Instead, they rely on alerting mechanisms that are complemented with powerful, human-friendly remote querying capabilities. HPC data services are transient, highly concurrent services that run on high-performance hardware. Thus, while the cloud-based time-series data model has application in monitoring HPC services, we find that the specialized hardware and software stack necessitates a high-performance monitoring service implementation.

5.2.1.2 HPC Monitoring and Analysis Tools. The SYMBIOMON design shares a lot in common with the state-of-the-art HPC monitoring services and aggregation frameworks, such as SOS [165], LDMS [166, 190], MRNET [167], Ganglia [168], and Nagios [191]. All of these tools implement distributed data aggregation and optionally allow the ability to store the aggregated data for offline or remote analysis. Tools such as WOWMON [164] and SOS are closest to SYMBIOMON in terms of design and target usage. They all export a client instrumentation library and allow the ability to extract and aggregate performance data from multiple sources. However, they do not explicitly track time-series data, nor do they offer the ability to analyze or reduce performance data at the source. Instead, analysis is often the last step in the data collection pipeline.

Time-series monitoring is similar in approach to HPC event tracing. While tools such as Vampir [192] and Jumpshot [193] offer sophisticated support for offline event trace analysis, fewer tools exist that can perform online analysis of trace events. Online event tracing in the HPC application context is known to be prohibitively expensive in terms of memory and storage costs [194]. Thus, there is a need to reduce, compress, or down-sample the trace data before it becomes a problem. In situ analysis tools such as Chimbuko [169] choose to analyze the trace data for performance anomalies while also generating provenance information. Seer [125] is an in situ analysis and simulation steering infrastructure intended explicitly for human user interaction. SYMBIOMON chooses to build upon a high-performance framework, reuse existing components, and extend them only where necessary, instead of building everything from scratch. This design principle ensures that the service components are more maintainable than ad hoc counterparts.

5.2.2 Design. SYMBIOMON is a high-performance, *composable* metric-monitoring service. SYMBIOMON can capture, store, analyze, and export performance metrics from any distributed component running on the HPC platform. Its composable architecture enables a user to turn features on and off easily. SYMBIOMON consists of three core microservice components—a COLLECTOR that exports the metric collection API, an AGGREGATOR that aggregates and stores metric data from distributed COLLECTOR instances, and a REDUCER that performs a global reduction of the metric data. The COLLECTOR component and the SYMBIOMON metric API have been described in Chapter IV. Here we present the AGGREGATOR and REDUCER components and describe their composition model in detail.

5.2.2.1 AGGREGATOR. SYMBIOMON depends on a two-stage reduction scheme to manage the volume of time-series data being transferred between different microservice components. During the first stage, the COLLECTOR implements a *local* reduction of the time-series data using one of the operators described in Table 17, and it stores the locally reduced results in one of the AGGREGATOR instances. The AGGREGATOR service consists of a configurable number of service provider processes that run on dedicated computing resources. A *cohort* is defined as a set of processing elements within a workflow that share a similar goal. Figure 30, for example, depicts two cohorts—the MPI application and the data service. When SYMBIOMON is employed to monitor multiple cohorts in a coupled workflow simultaneously, these cohorts can share a single set of AGGREGATOR instances.

5.2.2.2 REDUCER. The REDUCER component implements the second stage—a *global* reduction operation. When a client invokes the REDUCER's microservice API to perform the global reduction of a metric, the REDUCER contacts the appropriate AGGREGATOR instance to ingest the locally reduced values using RPC calls. Then, it applies the appropriate reduction operator on the set of locally reduced values and stores the resulting globally reduced value(s) as a regular COLLECTOR metric. Both reduction stages employ the *same* reduction operator for a particular metric; that is, a composition of different operators for the two stages is not yet supported. Each cohort is expected to use its dedicated REDUCER microservice. As depicted in Figure 30, the COLLECTOR metric representing a globally reduced result lives inside the REDUCER process. It can be exported via

RPC to an external remote monitoring client, a visualization module, or a control infrastructure that runs within the same node allocation.

5.2.2.3 Py-COLLECTOR: Python Client for Remote Monitoring.

The COLLECTOR exposes a Python client that can list the available metrics and return metric samples from a COLLECTOR provider running inside the HPC workflow. Specifically, this enables remote-monitoring scenarios that involve human interaction with the HPC workflow (for example, through a Jupyter notebook). Although the SYMBIOMON framework allows the querying of any COLLECTOR provider running in the system directly, we expect the typical user to query the COLLECTOR running alongside the REDUCER provider only for metric data that has undergone a global reduction.

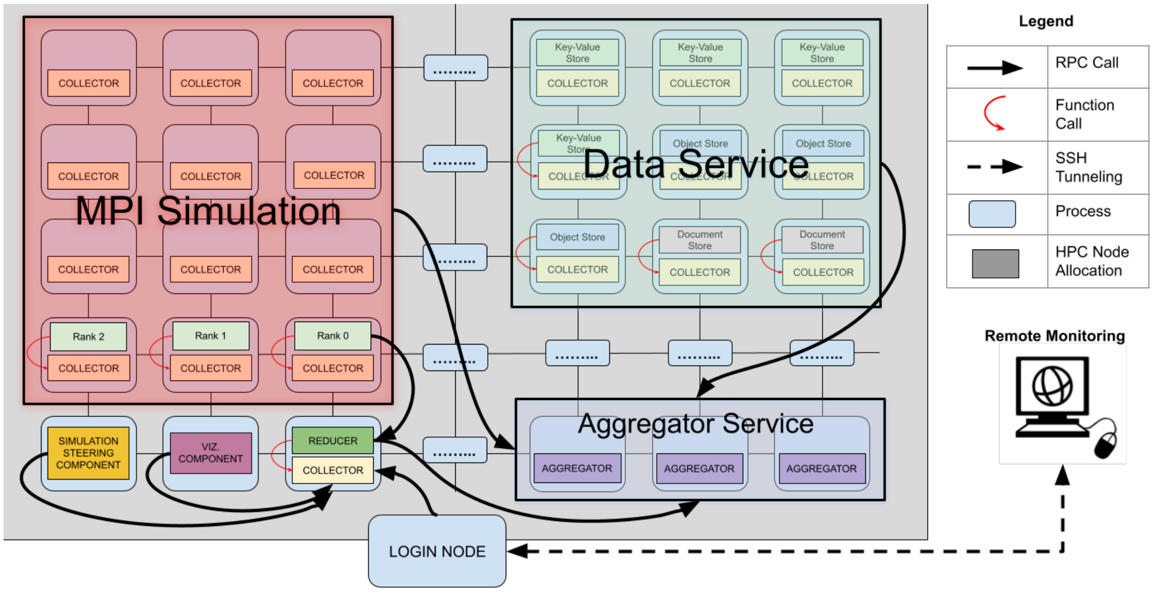


Figure 30. SYMBIOMON Conceptual Illustration

5.2.2.4 Flexible Integration. One of the key benefits of a composable design for the monitoring system is the resulting flexibility in its integration. Figure 31 depicts the various modalities for using SYMBIOMON. Further, switching between these modalities involves little to no client code modification. Once the service components are built, configured, and deployed, switching between the modalities in SYMBIOMON requires only an update to a set of environment variables describing the execution. Chapter IV explores the offline analysis of time-series data captured during the execution of a data service. At the same time, this chapter employs the monitoring and analysis components and describes their integration with HPC applications and ensembles.

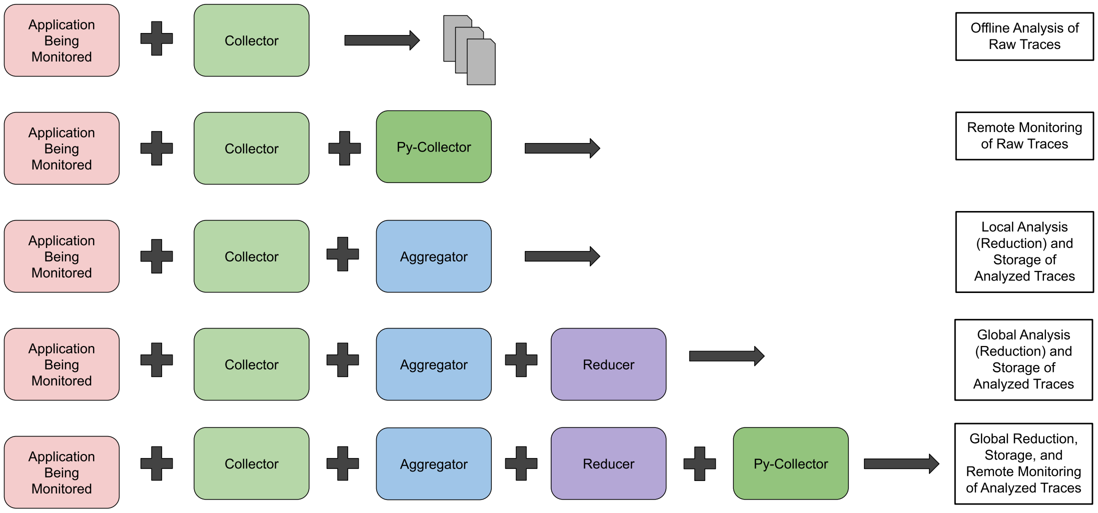

Application Being Monitored
Collector
Offline Analysis of Raw Traces
Application Being Monitored
Collector
Py-Collector
Remote Monitoring of Raw Traces
Application Being Monitored
Collector
Aggregator
Local Analysis (Reduction) and Storage of Analyzed Traces
Application Being Monitored
Collector
Aggregator
Reducer
Global Analysis (Reduction) and Storage of Analyzed Traces
Application Being Monitored
Collector
Aggregator
Reducer
Py-Collector
Global Reduction, Storage, and Remote Monitoring of Analyzed Traces

Figure 31. SYMBIOMON: Flexibility in Integrating Monitoring and Analysis Capabilities

5.2.3 Implementation.

Every SYMBIOMON component is implemented as a Mochi microservice. As a result, SYMBIOMON can take direct advantage of the high-performance RPC library software stack and all the tools that the Mochi framework provides. Here we describe the implementation of each SYMBIOMON component in the context of the Mochi framework. The metric capture through the COLLECTOR component has been described in Chapter IV. Here, we describe the implementation and composition of the AGGREGATOR and REDUCER microservices.

5.2.3.1 AGGREGATOR. The AGGREGATOR service is implemented as a collection of a configurable number of distributed SDSKV microservice instances. SDSKV is an existing Mochi microservice that provides general key-value store capabilities with a choice of multiple backends—an `std::map`, a LevelDB database, or a BerkeleyDB[195] database. When an application invokes the local reduction operation on a metric, the COLLECTOR invokes an `sdskv_put` operation (one per metric) to store the reduced value as a key-value pair. The key is generated by using the metric name, the metric namespace, and the reduction operator. The COLLECTOR passes this key through a hash function to identify the SDSKV instance to store the reduced result. The hash function attempts to evenly distribute keys across the total number of available SDSKV instances. Note that the metric taglist is not used to generate the key for aggregation. Doing so ensures that all the metrics with a common name and namespace are stored within the same SDSKV instance.

5.2.3.2 REDUCER. As Figure 30 depicts, the REDUCER microservice is invoked by the leader of a cohort to perform a global reduction operation. The REDUCER exposes a single API that takes the metric name, the metric namespace, and the aggregation operator as input arguments to the RPC. It employs the same hashing scheme as the AGGREGATOR to identify the SDSKV instance holding the key-value pairs representing locally reduced results. Then, it makes a single `sdskv_list_keyvals_with_prefix` RPC call to gather all the locally reduced values belonging to a cohort. Here the REDUCER makes a critical assumption: the taglist (if any) used to create the metric is consistent across all the cohort members. Once the locally reduced values are available, the REDUCER employs the aggregation operator to perform a global reduction of the particular metric. The REDUCER stores the resulting value as a COLLECTOR metric with a `GLOBAL` suffix added to the metric name. This metric can be queried by a visualization module or a control system implementation within the HPC workflow to identify performance trends.

5.2.3.3 BEDROCK Integration. BEDROCK functions as a bootstrapping and configuration system for Mochi microservices. Specifically, a user can employ BEDROCK daemons to dynamically compose, configure, and resolve service dependencies using a JSON configuration file. The user can also query the existing configuration of a service using BEDROCK. By implementing a BEDROCK module for the COLLECTOR microservice, any composed Mochi data service configured using BEDROCK can take seamless advantage of monitoring capabilities.

5.2.3.4 Service Discovery and Composition. Our prototype implements the AGGREGATOR service inside a separate MPI application; that is, the AGGREGATOR service instances are launched on MPI processes. Further, each service instance (one per process) is assigned a dedicated set of threads for executing RPCs. AGGREGATOR service instances collectively write out their addresses to a file subsequently read by the COLLECTOR instances during the service discovery phase. The REDUCER is launched as a standalone process and discovers the AGGREGATOR instances in the system using the same file. The REDUCER makes its address public through another file. After service discovery is complete, all subsequent interactions between distributed SYMBIOMON components are performed by using RPCs.

5.2.3.5 Metric Reduction. When aggregation is enabled, the COLLECTOR exposes an API to perform local reduction on a metric—aggregation results in the invocation of RPCs. Among the reduction operators currently supported, the `ANOMALY` is the only operator that can result in more than one value when reduction is invoked. When global reduction support is enabled, the COLLECTOR exposes an API to contact a REDUCER that performs the global reduction on behalf of the caller. It is expected that only the cohort leader (rank 0 of an MPI cohort, for example) invokes the global reduction operation.

5.3 Plugin Architecture for Ubiquitous Monitoring

The advances in high-performance computing (HPC) hardware and systems have made it possible to develop parallel applications of greater sophistication and power for purposes of achieving more ambitious objectives in computational and data science domains. With the potential for scalable parallelism, heterogeneous execution, massive concurrency, and low-latency/high-bandwidth interconnection, the challenge for applications is how to maximize the advantage these advances bring. Clearly, the evolution of HPC technology and integration have increased the complexity of this challenge. While new features in parallel languages, programming tools, and runtime system environments can help to transform existing applications or to develop new ones in ways that leverage HPC's strengths, they can also introduce complexities of their own. At the end of the day, the goal of gaining high performance is paramount, but productivity and performance portability concerns are important as well.

Throughout the history of parallel computing, it has been important to characterize and understand the performance of HPC systems and the applications that run on them. For this purpose, *parallel performance systems* have been developed to empirically analyze real applications on real machines. Several robust performance systems have been created for parallel systems with the ability to observe diverse aspects of application execution on the different underlying hardware. A key objective is to support measurement methods that are efficient, portable, and scalable. For these reasons, the performance measurement infrastructure is tightly embedded with the application code and runtime execution environment. Unfortunately, the performance system is not immune to the changes in HPC system environment and parallel programming methodologies. Researchers in the parallel tools' community know very well the constant attention needed to massage and extend a performance system to support the rich space of possible parallel operation that it is supposed to observe.

Indeed, it is remarkable that these parallel performance systems can do what they do on leading HPC platforms and for the variety of applications run on them today. However, as HPC systems and parallel software evolve, especially towards more heterogeneous, asynchronous, and dynamic operation, it is to be expected that the requirements for performance observation and awareness will change. For example, there is a growing interest in interacting with the performance infrastructure for in situ analytics and policy-based control. The problem is that the performance systems may need to react to new demands for performance observation that might go beyond the core features of the performance system. More seriously, the performance system architecture could be constrained in its ability to evolve to meet these new requirements.

Here we report our research efforts to address these type of evolutionary concerns in the context of the *TAU Performance System*. Given the goals of TAU to support current and next-generation high-performance parallel applications, it is necessary to create mechanisms to enable advanced performance measurement and analysis solutions. Specifically, we consider the use of a powerful plugin model to both enhance TAU's existing capabilities, and to extend its functionality in ways it was not necessarily conceived originally. The TAU plugin architecture supports three types of plugin paradigms: *EVENT*, *TRIGGER*, and *AGENT*. We demonstrate how each operates under several different scenarios. Results from large-scale experiments are shown to highlight the fact that efficiency and robustness can be maintained, while new flexibility and programmability can be offered that leverages the power of the core TAU system while allowing significant and compelling extensions to be realized.

5.3.1 Related Work. Our ideas are inspired by research contributions in parallel performance systems, performance interfaces, measurement libraries, and performance monitoring frameworks. There are several remarkable performance systems developed for HPC applications, most notably HPCToolkit [30] Scalasca [196], Vampir [192], Score-P [157], Extrae [197], Open|SpeedShop [198], and TAU [29]. All are robust, scalable, and able to work with applications written in a variety of languages, targeting multiple models of parallelism, and executing in sophisticated runtime environments. It is almost certainly the case that these performance systems face the same challenges as TAU with respect to extending their architecture to extend functionality. Callbacks and plugins are powerful software techniques for adding new capabilities flexibly to systems in general.

It is easy to identify the usage of such techniques in how software and hardware interfaces are made accessible to performance measurement. The PMPI interposition concept in MPI and its extension with $P^N MPI$ [199] make possible the linking and activation of software based on an MPI call. Other interfaces are exposed for performance measurements using callback support, such as OpenMP OMPT [51] and CUDA CUPTI [200]. PAPI [201] implements statistical profiling by installing and emulating arbitrary callbacks on hardware counter overflow. These and other similar mechanisms are used by performance systems to be made aware of events taking place in libraries, runtime systems, and hardware, as well as to provide access to the context in which the events occurred.

The general notion of a producer-consumer model for performance tool interaction maps to examples where callbacks and plugins can be applied. One example is the RCRToolkit [202] which can gather information from various producers and provide a shared memory region for real-time access by consumers. It is used for resource-level observation to better understand the performance interactions in shared-memory systems. Similarly, the APEX [203] autonomic performance system provides asynchronous introspection of performance for policy-driven adaptive control. APEX was developed for the HPX runtime system to support dynamic task scheduling for performance and power optimization. Both RCRToolkit and APEX allow online performance data processing and support separation of concerns between components. While APEX does not implement a plugin model, it is based on an event/listener model (*Observer* design pattern), and functionality can be added to APEX by implementing additional listeners. Paradyn [204] is an early foundational system that captures the abstraction of a composable system with pluggable components addressing measurement and online analysis for performance discovery.

This perspective extends to performance monitors in general. Such is the case with the Periscope Tuning Framework [205]. Periscope enlists a communication and analysis agent framework that runs in a connected way with the application to identify performance problems at runtime. While the access to performance measurements takes place in a "monitor library" called by application processes, the actual analysis takes place in Periscope's analysis agent network running on separate processes. The AutoTune project [206] extended Periscope with plugin support for runtime control.

It has been integrated with several parallel pattern libraries to tune parameters for power and scheduling.

There are two research projects closest to what we are discussing in this paper. The first concerns the extension of Score-P through plugins [207]. As noted above, Score-P [157] is a leading parallel performance system very similar to TAU. (In fact, TAU components can be integrated with Score-P.) It provides both “metric” plugins, which extend the standard timers and hardware counters, and “substrate” plugins, which enables custom processing of event streams, including doing analysis. In contrast, our plugin architecture works differently in that it allows a closer link between “states” and “triggers” while also allowing different plugins to be programmed and registered. We believe it would be possible to provide similar “metric” and “substrate” plugins within the TAU plugin architecture.

The second is the Caliper [208] performance system. Caliper consists of the source-code annotation API, a backend runtime component that manages blackboard buffers and the generalized context tree, add-on support services for control tasks (such as I/O and controlling snapshots), and additional data producer, measurement control, and data consumer services. Caliper utilizes a mixture of techniques identified in the tools, but combined in a novel and effective architecture. Snapshot mechanisms are the basic methods to collect data from data producers and provide it to data consumers. Data producers and consumers interact with Caliper through annotation and control APIs, or by registering callback functions for certain events. The callback interface provides notifications about system actions, measurement states, and event processing. Caliper is a newer performance system architecture than TAU and does not necessarily have the challenges TAU does. That said, there are similarities with respect to plugin support and it might be interesting to try to recreate certain Caliper functionality, such as the blackboards.

Finally, there is the interesting new support in PAPI for software defined events [209] that is related to our approach. The essential idea is that PAPI infrastructure can be extended to support events that come from different software layers (application, library, runtime) while leveraging the core PAPI infrastructures. These events are defined and programmed by the software developer. There is a way to register the events with PAPI, similar to plugin interfaces. Because the data created is all accessible within the PAPI system, performance systems that use PAPI, like TAU, could gain access. It would also appear that TRIGGER plugins could be used to reproduce PAPI software defined events.

5.3.2 Background.

5.3.2.1 TAU Overview. The TAU Performance System® [29] is the product of 25+ years of development to create a robust, flexible, portable, and integrated framework and toolset for performance instrumentation, measurement, analysis, and visualization of large-scale parallel computer systems and applications. TAU supports all major parallel paradigms (shared memory multithreading, distributed memory message passaging, data parallel acceleration), parallel programming models (e.g., MPI, OpenMP, OpenACC), and languages (e.g., C, C++, Fortran, Python). The TAU software is open source and has been ported to many processor architectures and HPC platforms around the world. It has been utilized in many performance analysis and optimization efforts across a wide spectrum of science and engineering applications.

From TAU's perspective, the execution of a program is regarded as a sequence of significant performance *events*. TAU was originally conceived to observe these events through *probes* inserted in the application code. The combination of a flexible event model and multiple instrumentation techniques allowed TAU to effectively morph its observation capabilities to capture events and their semantics that might otherwise be intractable. Over time, TAU expanded its observation approach to include event-based sampling (EBS) methods [210], where the "event" here is an interrupt to the application's execution. TAU's EBS support adds performance observability and detail. Both probes and statistical sampling (i.e., EBS) can be used simultaneously in TAU.

Logically, once "events" are made visible (via probes or sampling) they can be measured. The TAU measurement system *event interface* allows events to be defined, their visibility controlled, and their runtime data structures to be created. Each event has a *type* (*atomic* or *interval*), a *group*, and a unique *event name.* The event name is a character string and is a powerful way to encode event information. At runtime, TAU maps the event name to an efficient *event ID* for use during measurement. Events are created dynamically in TAU by providing the event interface with a unique event name. This makes it possible for runtime context information to be used in forming an event name (*context-based events*), or values of routine parameters to be used to distinguish call variants, (*parameter-based events*). TAU also supports *phase* and *sample* events.

The purpose of event control in TAU is to enable and disable a group of events at a coarse level. This allows the focus of instrumentation to be refined at runtime. All groups can be disabled and any set of groups can be selectively enabled. Similarly, all event groups can be enabled initially and then selectively disabled. It is also possible to individually enable and disable events. TAU uses this support internally to throttle high overhead events during measurement.

The measurement system is the heart and soul of TAU. It has evolved over time to a highly robust, scalable infrastructure portable to all HPC platforms. The instrumentation layer defines which events will be measured and the measurement system selects which performance data metrics to observe. Performance experiments are created by selecting the key events of interest and by configuring measurement modules together to capture desired performance data. TAUś measurement system provides support for portable timing, integration with hardware performance counters (e.g., PAPI [50]), parallel profiling, parallel tracing (with OTF-2 [211]), and runtime monitoring.

TAU's measurement system has two core capabilities. First, the *event management* handles the registration and encoding of events as they are created. New events are represented in an *event table* by instantiating a new *event record*, recording the *event name*, and linking in storage allocated for the event performance data. The event table is used for all atomic and interval events regardless of their complexity. Event type and context information are encoded in the event names. The TAU event management system hashes and maps these names to determine if an event has already occurred or needs to be created. Events are managed for every thread of execution in the application.

Second, a runtime representation called the *event callstack* captures the nesting relationship of interval performance events on each thread. It is a powerful runtime measurement abstraction for managing the TAU performance state for use in both profiling and tracing. In particular, the event callstack is key for managing execution context, allowing TAU to associate this context with the events being measured.

The final component of TAU is analysis tools. TAU includes support for parallel profile data management

(*TAUdb* [212]), analysis(*ParaProf* [213]), and data mining (*PerfExplorer* [214]). It leverages existing trace analysis functionality available in robust external tools, including the Vampir [192], Jumpshot [193], and Expert/CUBE [215, 216].

5.3.2.2 TAU Operation. Given this overview, let us consider how the TAU measurement actually operates. The TAU measurement system is implemented as a library that is loaded with a parallel application and lives within each application process on every node that is allocated to the job. During execution, events occurring within threads of execution on a process are then measured by TAU and stored in thread-specific data structures. That is to say, if a parallel profile measurement is being made, each application thread of execution will have its own profile data for the events that occurred on that thread. Similarly, each application thread will output a separate trace of events from that thread. TAU collects a variety of metrics for events instrumented by probes and/or captured by event-based sampling in parallel profiles and/or traces. All of the TAU performance data is stored in the memory space of each process.

Interval events capture performance metrics that occur between *entry* and *exit* actions. The metrics could include execution time, hardware counters, or software counters. Atomic events are used to capture metrics of interest at a particular point in time or location in the code. Metrics could include hardware counters, message sizes involved in MPI communication, memory allocation, and so on. For every thread of execution within a process, TAU collects data for every interval and atomic event instrumented.

Figure 32 shows the two event types occurring on a thread of execution and TAU's updating of the thread's performance data structure. Figure 33 portrays how TAU operates for multiple application processes. All of the TAU performance data

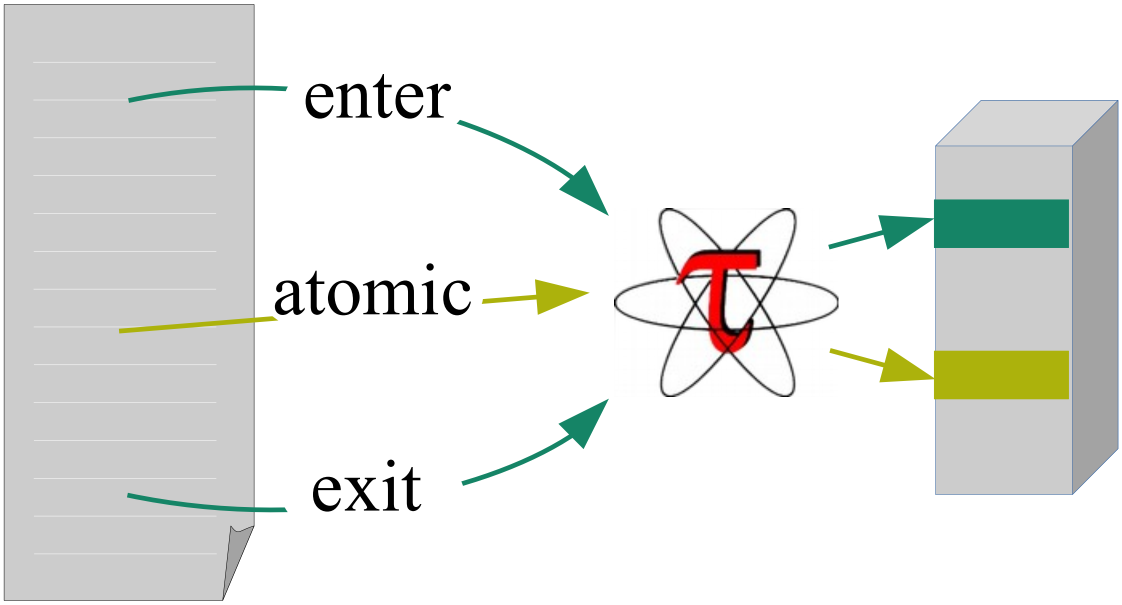


Figure 32. TAU supports *interval* and *atomic* events. Event measurements are made for each event occurring on each thread of execution. The performance data is store in a thread-specific data structure.

measured for each process thread is stored in the process memory space. Because TAU is a library, it will run within the process thread that called it. For this reason, measurement should be made as efficiently as possible so TAU can return to the application quickly. Furthermore, TAU is a thread-safe library and can be running simultaneously on each application thread of execution. In a distributed MPI application, TAU will generate performance data that resides locally within each MPI rank.

TAU measurements are recorded in parallel profiles and/or traces. Figure 34 portrays the profiles kept for all application processes which are spread across the HPC nodes allocated for the execution. TAU collects these profiles at the end of program execution and saves the performance information for offline analyses. TAU provides a routine for a process to access its thread profiles during execution, including taking a snapshot of the entire profile data.

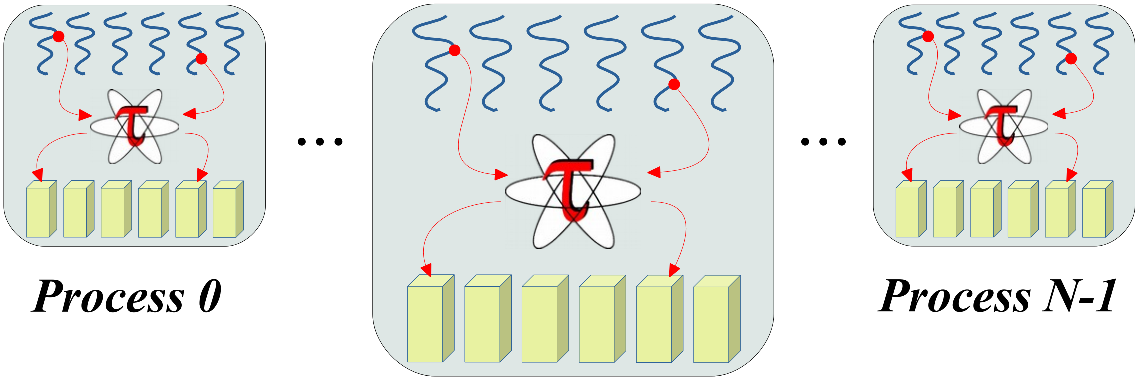


Figure 33. TAU captures performance data for all threads in each application process.

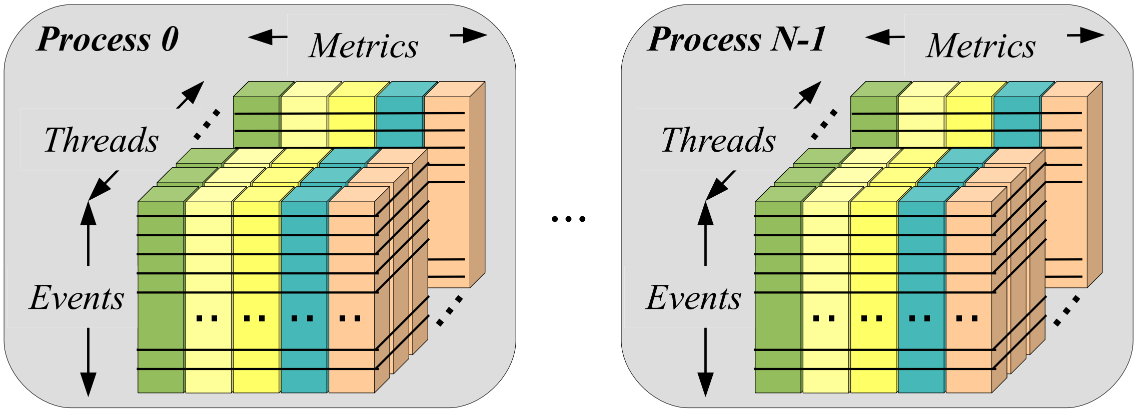


Figure 34. TAU collects parallel profiles for different events across all threads and processes in an application's execution. The "global" parallel profiles are distributed across all nodes of the application.

5.3.2.3 Constraints. While TAU offers powerful measurement capabilities which have been successfully applied over many years, there are aspects of its present operation that will impose constraints to its use in the future. These include:

- All TAU execution takes place on application threads, including event measurements and interrupt handlers. There is no support for TAU-only thread resources to aid in any runtime tasks.

- TAU measures exactly the same metrics for all interval events. It is not possible, for example, to measure only time for one event and only cache misses for another event.

- If TAU is set up for profiling and tracing, all events are profiled and all events are traced. It is not possible to have only a subset of events traced.

- Although TAU offers "user-defined" events (both interval and atomic), the application can not pass TAU any data. The atomic interface has basic support for processing application values, but the functionality is limited.

- TAU resides in each application process and thus executes in a distributed manner, but there is no support for TAU to interact across the other processes except at the end of the program.

- Beyond the EBS support in TAU, there are rudimentary mechanisms for handling signals and asynchronous events.

Our objective is to update and extend the TAU architecture in such a way to address these concerns. We believe that doing so will "open up" TAU development to enhancements with broad interest.

In general, TAU is not dissimilar to other major parallel performance system in its design and operation, including Score-P [157], HPCToolkit [30], Scalasca [196], Vampirtrace [192], Extrae [197], and Open|SpeedShop [198]. It is possible that these performance systems struggle with the same limitations.

5.3.3 TAU Plugin Architecture. Our objective is to overcome the potential future constraints to TAU by updating its system design architecture to allow new functionality to be developed. Specifically, we look to a *plugin architecture* to extend the core measurement and analysis capabilities in TAU. The general idea is that a plugin will register callbacks to respond to certain states of TAU execution. When such a TAU state occurs, any registered plugins are called with data appropriate for the state semantics. In this manner, the plugin operation can be informed of its context and implement additional processing. Below we discuss our design and development approach to a TAU plugin architecture.

Of course, the challenge for a performance system such as TAU with mature capabilities in frequent use is how to "integrate" a plugin architecture into an existing implementation. The tension between retrofitting an existing system and wholesale redesign is always there. Nevertheless, the plugin architecture we hope for should be designed to address as best as possible the following criteria:

- Separation of concern between what action invokes a plugin and what functionality the plugin provides.
- Control over enabling/disabling of plugin invocation versus control of plugin operation.
- Access to TAU's internal operations and performance information.

– Flexible plugin programming methods that are not constrained by necessarily by TAU development rules.

These criteria are better thought of as guidelines for the plugin architecture rather than strict requirements, since they will be addressed at different degrees depending on what is being done.

The following describes the approach taken for our prototype TAU plugin architecture. We begin with a description of the "state classes" distinguished by TAU and what additional access points might be relevant. We then discuss the operational model for plugins at a high level, followed by a description of the plugin prototype implementation.

5.3.3.1 TAU States and STATE Plugins. TAU supports multiple capabilities and is in different states of processing depending on what it is doing at a particular time. For instance, Section §5.3.2.1 describes at a high-level what happens when TAU makes measurements for interval and atomic events, but there are other functionalities that TAU supports that are relevant to identify. The general idea is that plugins could register callbacks to one or more of TAU's "states" of execution that could be "salient" or "interesting" from a perspective of additional plugin processing.

Presently, TAU recognizes several states of execution that effectively correspond to locations inside the TAU performance system where the processing is taking place. Each state will have a state-specific *context* that captures relevant information of importance of the plugin. If the state context identifies the particular instance, we will say that the plugin is *named*, otherwise, it is *generic*. For example, when TAU is in a `FUNCTION_ENTRY` processing state, the name of the function being entered is part of the state context, making it possible for the plugin to be invoked with the function name, to distinguish it from other occurrences of `FUNCTION_ENTRY` processing. Below is a non-exhaustive list of TAU states that are presently available for plugin support.

– `FUNCTION_ENTRY/EXIT`: When TAU measures interval events, the profiling module is invoked at the entry and exit of every instrumented interval event. The `FUNCTION_ENTRY` and the `FUNCTION_EXIT` TAU states occur accordingly. Here, the interval event name, TAU's associated function ID, function group, and timestamp information is context data that can be passed to a registered *named* plugin.

– `PHASE_ENTRY/EXIT`: TAU makes it possible to demarcate phases within the application source code. The `PHASE_ENTRY` and `PHASE_EXIT` TAU states represent the TAU processing states specific to phases. The phase name is a part of the context provided to a registered *named* plugin for invocation with every entry/exit of static and dynamic phases.

– `INTERRUPT_TRIGGER`: TAU installs a signal handler for the `SIGUSR` signal to implement sampling operations. When sampling is turned on and TAU's signal handler is run, TAU is in the `INTERRUPT_TRIGGER` state. Different interrupt intervals are possible. Any registered plugins are invoked with a *generic* context.

– `MPI_T`: When TAU is configured to support the MPI Tools Information Interface (MPI_T [217]), TAU collects performance variable data (PVARs) through the MPI_T interface. Each PVAR has a name and a unique ID associated with it. When TAU is querying a PVAR value, it is in the `MPI_T` state. Registered *named* plugins can be called with the PVAR name and ID.

– `POST_INIT/PRE_END_OF_EXECUTION`: During its initialization phase, TAU performs a number of important internal tasks such as invoking the `init()`

routines of downstream APIs, allocating memory for timers, and so on. TAU is in the `POST_INIT` state then. Similarly, during its finalization phase after profiling has been completed, TAU frees up allocated memory, invokes the `finalize()` routines of APIs, and prepares to shut down operations and write out the profile or trace information that has been collected. TAU is in the `PRE_END_OF_EXECUTION` state then. During these states, registered *generic* plugins can utilize TAU support to perform specialized tasks that rely on parallel programming and profiling libraries to be in a certain state (initialized or finalized).

We have designed *STATE* plugin support to work with TAU states and have developed specific plugins for each of the cases above. Figure 35 (top) graphically portrays how the plugins work for the `FUNCTION_ENTRY/EXIT` and `ATOMIC` TAU states. Because this is a plugin for a *named* state, it is called with an interval event name or atomic event name, plus other context information. The plugin can then process the event with or without specializing its function based on the context. Event-specific plugin processing is indicated by the plugin shading. Notice, the standard TAU performance measurements are still being performed. The plugin processing is whatever the plugin developer implemented.

5.3.3.2 TRIGGER Plugins. TAU states discussed above are all defined by and occur within the TAU performance system. Registering generic and named plugins for TAU states offers a clear separation of concern from the state occurrence and any plugin action associated with it, beyond the standard TAU measurement processing. In essence, it allows for enhanced observation and actuation of extended functionality "triggered" by TAU state. We will refer to these plugins as *TRIGGER* plugins.

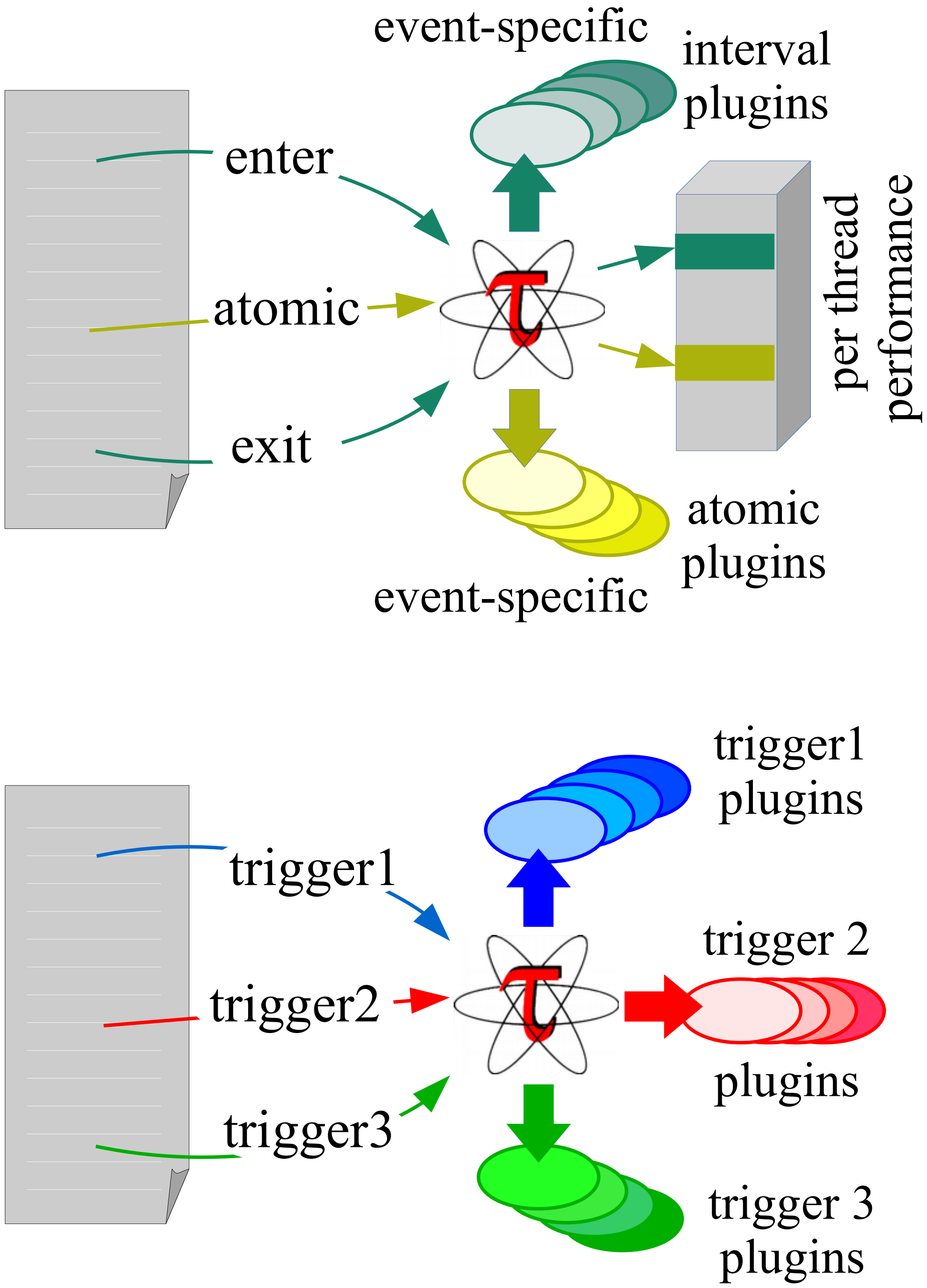


Figure 35. (Top) STATE plugin operation for interval and atomic events. (Bottom) TRIGGER plugin operation.

Should it be the case that only internal TAU processing can trigger plugins? Consider a situation where the application desires an analysis of current TAU performance data at a particular point in its execution. For instance, it might be interested in detecting performance anomalies per iteration of a time-stepped simulation on a large HPC system. One example of a performance anomaly is a sudden increase in communication time that deviates significantly from regular behavior. If such an anomaly is detected, the application could decide to checkpoint certain aspects of its application state, as well as performance state, for offline analysis. The analysis task can be broken down into the following distinct elements:

- The application invokes (somehow) performance processing within TAU at the end of every iteration.
- Communication performance is aggregated and stored in a buffer.
- Anomalies are analyzed with respect to previously stored data.
- Results are returned (somehow) to the application.

How should this functionality be developed? It seems reasonable to want to actuate certain operations for a task like this independent of a particular TAU state. In fact, it should be expected that the "triggering" of these operations could occur outside of TAU altogether, such as from the application itself. However, it is still important that the operations run "inside" the TAU environment so that they can access the TAU performance data.

In keeping with our plugin design concerns, we want to register a plugin with respect to a "trigger" that actuates the plugin when "fired" while keeping plugin functionality isolated within its implementation. To do so, a means to create a trigger, to fire the trigger, and receive a result is required. We have designed the TRIGGER plugin architecture that does just that. Like the STATE plugin, a TRIGGER plugin will run on an application thread of execution in the TAU environment with access to TAU performance data. However, it requires a new API to create and fire the trigger and exchange data across the interface. Figure 35 (bottom) shows how the TRIGGER plugins operate. They are *named* in the sense that a trigger has a unique name and plugins register for specific triggers.

5.3.3.3 AGENT Plugins. Both STATE plugins and TRIGGER plugins are *synchronous* in that they are invoked by TAU run on the same application thread as TAU, and are expected to return to TAU. Interestingly, TAU was originally designed to execute on the same threads of execution as the application. For the most part, it continues to do so now. Suppose that we relax this constraint and allow TAU to create new threads that then run in an *asynchronous* manner to the application. The concept of an *AGENT* plugin is based on exactly this thought. It is a plugin that has its own thread resources to do things that are not otherwise bound by application or TAU processing. Its functionality benefits from running on a thread within the process and hence has full access to TAU information. Like STATE and TRIGGER plugins, in fact, it can allocate memory for its operation. It can interact (for the purpose of exchanging data or control) using thread synchronization mechanisms with the TAU system and with other plugins.

5.3.4 Plugin Implementation. The plugin system is implemented in C++ with C/C++ interfaces. The user can specify the path to the directory containing the plugins using the environment variable `TAU_PLUGINS_PATH`. The user can also specify the plugins to be loaded using the environment variable `TAU_PLUGINS`, separating the plugins by use of a delimiter.

5.3.4.1 ***Plugin Lifecycle.*** The plugin system in TAU operates through the following phases:

– **Initialization**: This is invoked during TAU library initialization. During this phase, TAU's plugin manager reads the environment variables `TAU_PLUGINS_PATH` and `TAU_PLUGINS` and loads the plugins. Internally, each loaded plugin is associated with a unique, unsigned integer ID for the duration that it is in scope. Each plugin must implement a function called `Tau_plugin_init_func`. Inside this function, every STATE plugin registers callbacks for a subset of TAU states.

– **Callback**: When TAU states occur during execution, the plugin manager invokes any registered callbacks for the specific TAU state in the increasing order of their ID's. Each state that is supported has a specific, typed data object associated with it. When the state occurs, this data object is populated and sent as a parameter to the plugin callback. The TRIGGER plugin is special as it serves the purpose of analyzing application-defined data. As a result, a *void** pointer is passed to such plugins from within the application. It is up to the plugin developer (user) to know how to decipher this information from within the plugin.

– **Finalize phase**: When TAU is done processing the performance data measured for an application, the plugins are notified to complete their operation. They are then unloaded, and all the auxiliary memory resources allocated by the plugin manager are freed. Optionally, the plugin may use this phase to write out any performance data for offline analysis at the end of the execution.

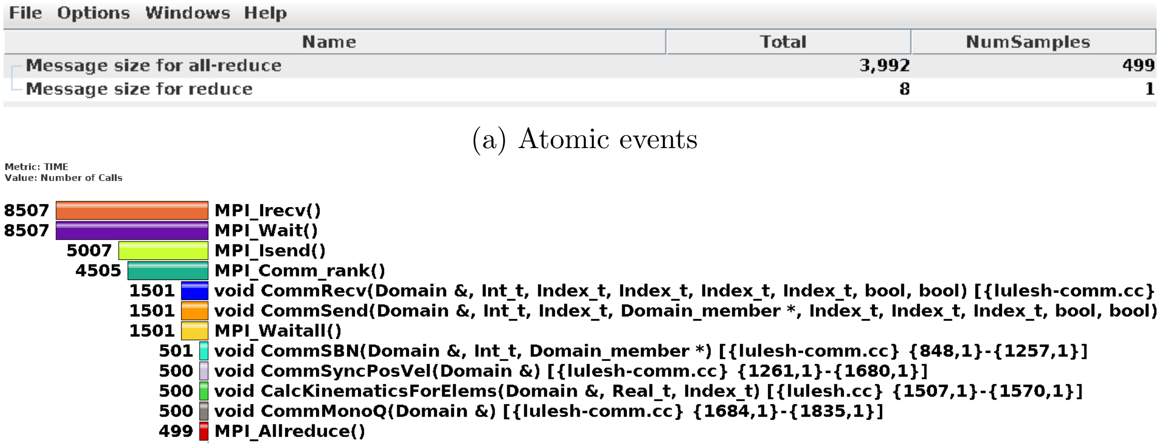
5.3.4.2 Enabling Customizability and Runtime Control. Section 5.3.3.1 introduces the notion of *named* and *generic* states in TAU. For instance, the function name field is used to distinguish between the various `FUNCTION_ENTRY` states. The entry of functions `foo()` and `bar()` represent separate, distinguishable states in our framework.

By default, all registered plugins for a given state are invoked when the state occurs. We define *customizability* as the ability to specify the execution of a subset of all registered plugins on the occurrence of a certain named state. Customizability is valid only for named states. Internally, named states are hashed based on the name field and a map is used to store the list of plugins to execute for the specific named state. TAU allows users to modify this map through the plugin API. Note that the full name of the state or its regular expression can be used to access the map.

We define *runtime control* as the ability to enable or disable specific plugins for specific states (named or generic) at runtime. The TAU plugin API allows users access to such functionality. This API requires the name of the event (or its regular expression) along with the plugin ID. The same functionality is extended to trigger states as well, where the trigger ID is used in place of the name. We envision runtime control to be invoked infrequently in the program, such as when a phase change occurs. Also, we expect both customizability and runtime control to be invoked in a serial portion of the program. These semantics allow for a simpler plugin framework design that involves a significantly reduced need for locking of internal data structures.

5.3.5 Usage Scenarios. In this section, we describe some of the scenarios that motivate and are enabled by the design of our plugin system.

5.3.5.1 Event Counter. In order to broadly demonstrate our plugin system, we designed a plugin that registers callbacks for a variety of supported TAU



File
Options
Windows
Help

Name	Total	NumSamples
Message size for all-reduce	3,992	499
Message size for reduce	8	1

(a) Atomic events

Metric: TIME
Value: Number of Calls
8507 MPI_Irecv()
8507 MPI_Wait()
5007 MPI_Isend()
4505 MPI_Comm_rank()
1501 void CommRecv(Domain &, Int_t, Index_t, Index_t, Index_t, Index_t, bool, bool) [{lulesh-comm.cc}
1501 void CommSend(Domain &, Int_t, Index_t, Domain_member *, Index_t, Index_t, Index_t, bool, bool)
1501 MPI_Waitall()
501 void CommSBN(Domain &, Int_t, Domain_member *) [{lulesh-comm.cc} {848,1}-{1257,1}]
500 void CommSyncPosVel(Domain &) [{lulesh-comm.cc} {1261,1}-{1680,1}]
500 void CalcKinematicsForElems(Domain &, Real_t, Index_t) [{lulesh.cc} {1507,1}-{1570,1}]
500 void CommMonoQ(Domain &) [{lulesh-comm.cc} {1684,1}-{1835,1}]
499 MPI_Allreduce()

(b) Number of function calls

Figure 36. Paraprof: LULESH profile

plugin processing events. The events of primary interest to us are `FUNCTION_ENTRY`, `FUNCTION_EXIT`, and `ATOMIC_EVENT_TRIGGER`. Inside the callback for the plugin event, a thread-level counter keeps track of the number of times that TAU has seen that event.

No filtering of any sort is performed on the processing state: the plugin is invoked for every named and generic processing state encountered. For example, the `FUNCTION_ENTRY` state is invoked at the entry of every function that is instrumented by TAU. The rationale behind designing this plugin is to correlate information collected by the plugin with the information collected by TAU's measurement system. Figure 36a summarizes the number of function calls made by thread 0, MPI rank 0 of a LULESH [218] application that has been instrumented and built with MPI+OpenMP support. Figure 36b depicts the number of atomic events that have been triggered on this thread. Table 22 is the statistics that are output by the event counter plugin for thread 0, MPI rank 0 at the end of execution. It is evident that this correlates well with the profiling information collected by TAU, depicted in Figure 36.

Table 22. LULESH: Processing state counter values on thread 0, MPI rank 0 (other ranks also captured)

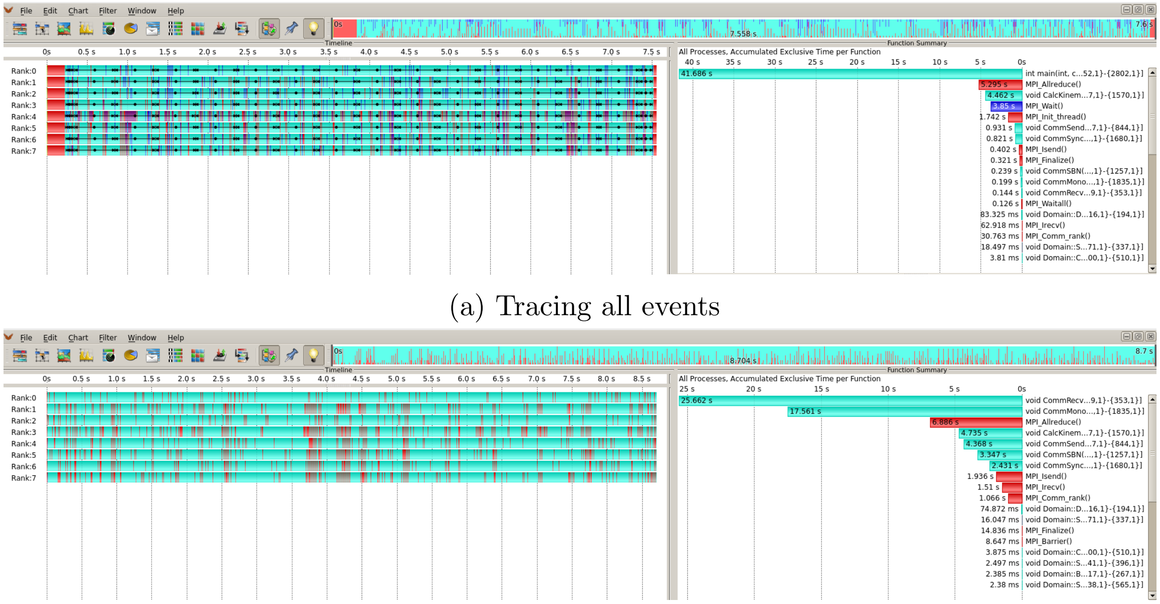
Processing State	Counter Value
FUNCTION_REGISTRATION	29
FUNCTION_ENTRY	23528
FUNCTION_EXIT	23538
ATOMIC_EVENT_REGISTRATION	2
ATOMIC_EVENT_TRIGGER	500

5.3.5.2 Selective Tracing. Section 5.3.2.1 describes the types of events that TAU measures: interval and atomic events. Every event that passes through TAU's measurement system is either profiled and/or traced. As such, there is no mechanism in TAU that allows one to specify separate sets of events to profile and trace. In other words, the action to profile and/or trace is specified at a high level and is applied across the board for all instrumented events.

The plugin system allows one to separate the occurrence of an event from the action associated with it. Particularly, it allows for the capability to profile all events while only tracing a *subset* of those events. This is useful in at least two ways:

- It reduces the size of the trace file generated.
- It reduces the amount of information in the trace file making it easier to visualize only the events that are important.

A plugin that performs selective tracing registers callbacks for the `FUNCTION_ENTRY` and `FUNCTION_EXIT` processing states and performs tracing inside these callbacks. The selection of these named processing states, however, is performed from within the application using TAU's plugin API. The application uses C++ `regex` support to enable/disable selective tracing for classes of functions. In order to demonstrate the usage of this functionality, we consider a LULESH application that been instrumented


File Edit Chart Filter Window Help
Timeline
Function Summary
0s
7.558 s
7.6 s
All Processes, Accumulated Exclusive Time per Function
Rank:0
Rank:1
Rank:2
Rank:3
Rank:4
Rank:5
Rank:6
Rank:7
41.686 s int main(int, c...52,1}-{2802,1}]
5.295 s MPI_Allreduce()
4.462 s void CalcKinem...7,1}-{1570,1}]
3.85 s MPI_Wait()
1.742 s MPI_Init_thread()
0.931 s void CommSend...7,1}-{844,1}]
0.821 s void CommSync...,1}-{1680,1}]
0.402 s MPI_Isend()
0.321 s MPI_Finalize()
0.239 s void CommSBN(...,1}-{1257,1}]
0.199 s void CommMono...,1}-{1835,1}]
0.144 s void CommRecv...9,1}-{353,1}]
0.126 s MPI_Waitall()
83.325 ms void Domain::D...16,1}-{194,1}]
62.918 ms MPI_Irecv()
30.763 ms MPI_Comm_rank()
18.497 ms void Domain::S...71,1}-{337,1}]
3.81 ms void Domain::C...00,1}-{510,1}]

(a) Tracing all events

File Edit Chart Filter Window Help
Timeline
Function Summary
0s
8.704 s
8.7 s
All Processes, Accumulated Exclusive Time per Function
Rank:0
Rank:1
Rank:2
Rank:3
Rank:4
Rank:5
Rank:6
Rank:7
25.662 s void CommRecv...9,1}-{353,1}]
17.561 s void CommMono...,1}-{1835,1}]
6.886 s MPI_Allreduce()
4.735 s void CalcKinem...7,1}-{1570,1}]
4.368 s void CommSend...7,1}-{844,1}]
3.347 s void CommSBN(...,1}-{1257,1}]
2.431 s void CommSync...,1}-{1680,1}]
1.936 s MPI_Isend()
1.51 s MPI_Irecv()
1.066 s MPI_Comm_rank()
74.872 ms void Domain::D...16,1}-{194,1}]
16.047 ms void Domain::S...71,1}-{337,1}]
14.836 ms MPI_Finalize()
8.647 ms MPI_Barrier()
3.875 ms void Domain::C...00,1}-{510,1}]
2.497 ms void Domain::S...41,1}-{396,1}]
2.385 ms void Domain::B...17,1}-{267,1}]
2.38 ms void Domain::S...38,1}-{565,1}]

(b) MPI_Wait events disabled

Figure 37. Vampir: LULESH trace

with the Program Database Toolkit (PDT) and built with MPI+OpenMP support. We trace all events except `MPI_Wait` events. Figure 37a depicts the OTF2 trace visualization generated in Vampir [219] when *all* instrumented events are traced. Figure 37b depicts the trace visualization generated when tracing for `MPI_Wait` events is selectively disabled using TAU's plugin infrastructure. Note that this functionality is distinctly different from using Vampir to "turn off" the uninteresting events. The plugin system prevents such events from being processed in the first place, allowing for a significant reduction in trace sizes.

5.3.5.3 Filter Plugin. When instrumenting an application using compiler-based instrumentation or PDT, TAU uses a *selective instrumentation file* to selectively enable/disable instrumentation for functions and/or files. We developed a filter plugin that allows us to enable filtering *at runtime*. The plugin registers a callback for the `FUNCTION_REGISTRATION_COMPLETE` processing state that is invoked exactly once for every function that is instrumented using TAU. Inside the callback, the plugin checks the selective instrumentation file to see if instrumentation needs to be disabled for the function in question.

This functionality enables selective instrumentation to be performed without the need to recompile the application in order to perform different targeted performance studies. As opposed to the selective tracing plugin, the filter plugin enables specialization of plugin functionality through an external mechanism (selective instrumentation file).

5.3.5.4 TAU SOS Plugin. The Scalable Observation System (SOS) [220] is a framework for aggregating performance data from distributed workflows in support of in-situ analysis and feedback & control. SOS is designed as an aggregation network consisting of client data sources, intermediate data listeners (one per allocation node), and one or more aggregation servers running on additional allocation nodes. Some HPC submission queues and batch systems don't allow launching multiple distributed applications per allocated node, so the SOS plugin follows a particular startup procedure to work around that limitation. The integration description below assumes that TAU is measuring an MPI application.

SOS is integrated into TAU as a plugin that can aggregate the data either as performance profiles or full event traces. In a typical usage scenario, the SOS plugin is configured to initialize the SOS client connection during TAU plugin initialization. It is assumed that one or more SOS aggregation daemons are already running on additional allocation nodes, having been launched by the submission script. If the SOS client library is unable to make a connection to an SOS listener daemon running on its node, the plugin will use the MPI infrastructure to coordinate between all ranks, group them by the nodes they are assigned to and then choose one rank on that node to launch an SOS listener daemon. The chosen rank will `fork` and `exec` the SOS listener process daemon for the SOS clients to connect to.

After the connections are established, the plugin will then optionally spawn a thread to perform asynchronous data aggregation over SOS. If asynchronous data aggregation isn't used, the data can be aggregated using the TAU trigger event. For profiling, the plugin code will iterate over the profile data of the current process, pack and publish it to the local listener. When tracing, the plugin will pack data in the local SOS client for each event and publish either periodically or on a trigger event. Once the listeners have received the data, they will then forward it to the aggregator(s).

5.3.5.5 Trigger: Aggregating Interval Events. The trigger mechanism is special in the sense that it allows an application to access TAU without an accompanying measurement being made. This can be harnessed to aggregate and snapshot performance information at an application-defined point of execution. We developed a plugin that aggregates and summarizes TAU interval and atomic event information across *all* threads of *all* processes inside the callback for the `TRIGGER` event. Note that this aggregation happens *synchronously* on the thread that invokes the trigger-event.

Additionally, the plugin depends on MPI to perform the necessary reduction operations on the TAU performance data. The summarized information is stored inside the plugin as a snapshot and can optionally be written out at the end of the application. The overheads involved in enabling this plugin depends on three factors:

– The number of processes involved in the reduction operation.

– Frequency of invocation of the trigger plugin.

– Number of interval and atomic events to be processed.

5.3.5.6 Agent: Asynchronous Load Tracking. An agent plugin is especially useful in a situation where a background task needs to be performed on a repetitive basis in such a way that the application execution isn't interrupted. Recall that the agent plugin is assigned a thread when the plugin is loaded. This thread can perform arbitrary tasks.

In addition to the SOS plugin that performs asynchronous work, we developed an agent plugin that periodically wakes up to record the current system load and resident set size in an OTF2 trace for every process. Specifically, the plugin creates a single *pthread* during the plugin initialization phase. In an MPI application, each MPI process would create its own *agent* thread. The load information is gathered from `\proc\loadavg` and the resident memory usage is extracted using the `getrusage` POSIX routine. Each of these metrics is internally stored as TAU user events and recorded in an OTF2 trace for offline visualization. The *agent* thread and its resources are cleaned up during the execution of `PRE_END_OF_EXECUTION` plugin state.

5.4 Monitoring of HPC Applications and Ensembles

Extending the SYMBIOMON monitoring service such that it can generally apply to HPC applications and ensembles requires a scheme to control the large volume of data generated. The other aspect to consider is seamlessly integrating these distributed time-series monitoring capabilities into traditional HPC applications. Here we describe an archetype of such a setup using the TAU performance system [29].

5.4.1 TAU Plugin for SYMBIOMON. As depicted by Figure 38, TAU's plugin mechanism is employed to integrate SYMBIOMON instrumentation into any traditional MPI application seamlessly. The SYMBIOMON plugin installs a callback to the `TAU_DUMP` plugin event triggered when the application invokes the `Tau_dump`

API call. We expect the typical application to repeatedly invoke this API at an appropriate location in the code, for example, at the end of every simulation timestep. The plugin uses all the SYMBIOMON components to enable online monitoring and analysis of application performance. The plugin has access to the TAU measurement API, making it a powerful mechanism for building application-monitoring capabilities.

In the callback function for the `TAU_DUMP` event, the plugin gathers a list of all timers and counters currently being tracked by TAU. For each of these timers and counters, the plugin creates a SYMBIOMON metric of an appropriate type. Within a process, TAU timers are expected to increase monotonically. Thus, TAU timers map directly to SYMBIOMON TIMERS. On the other hand, TAU counters are either SYMBIOMON COUNTERS or GAUGES, depending on what they represent. All the metrics are created within the "tau2" metric namespace. The metric name is borrowed directly from TAU. When the plugin is built with MPI support, the "taglist" for all the metrics contains the MPI rank information. Every time the plugin is invoked, the TAU measurement API is queried for the latest value of a metric, and the metric is updated accordingly. Local metric updates, local reduction, and metric aggregation do not require synchronization among the instrumented MPI processes. The global reduction, however, depends on the individual results being readily available with the AGGREGATOR service. The user can optionally request an MPI_Barrier before rank 0 contacts the REDUCER to invoke a global reduction. Doing so guarantees the total correctness of the operation. On the other hand, a lack of synchronization may result in a global reduction being performed on partially available data. The plugin also provides the user with an option to control the frequency of reduction operations. At present, a change in the aggregation operator for timers and counters

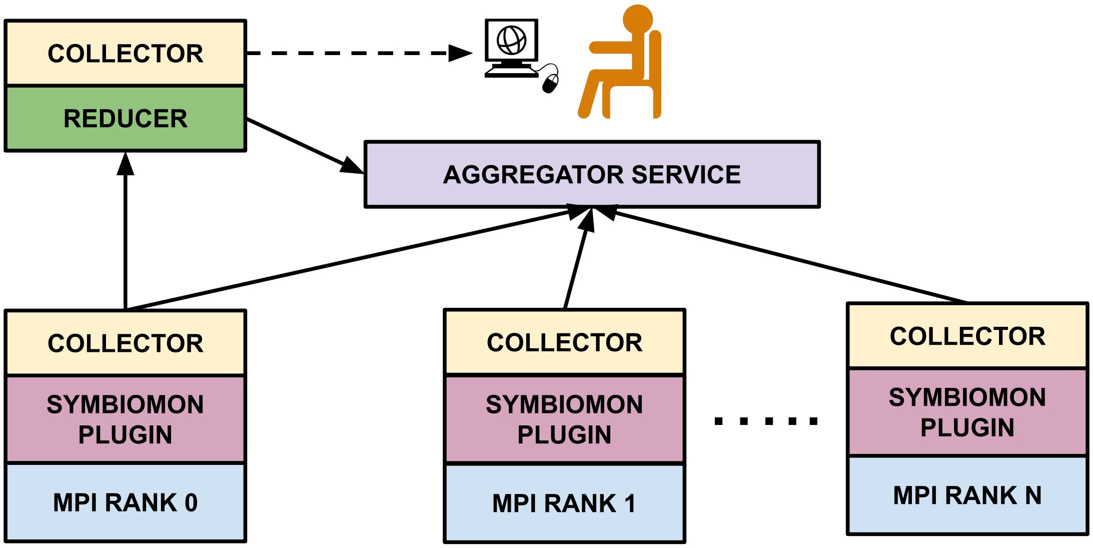


Figure 38. TAU SYMBIOMON Plugin

requires recompilation of the plugin. In the future, we plan to export environment variables to control this behavior.

5.4.2 HPC Application Monitoring: Visualizing LULESH Load Imbalance. We apply the TAU plugin for SYMBIOMON to monitor and visualize LULESH performance data during execution. Specifically, the `tau_exec` program is used to wrap MPI calls made by the LULESH application. The global range of the total exclusive time for MPI_Allreduce is used as a proxy for the degree of load imbalance in the application. LULESH is unchanged except for a single line of code that invokes the `Tau_dump` routine and triggers the plugin module. AGGREGATOR and REDUCER support is enabled. A Jupyter notebook is employed to connect to the COLLECTOR instance running alongside the REDUCER. We extract time-series data for the (instantaneous) global maximum and minimum for the MPI_Allreduce exclusive call time using the COLLECTOR's Python client. Figure 39 depicts a visual representation of this time-series data when the load is balanced between processes (blue and green lines). When load imbalance is induced in the application through the use of configuration options (orange and red lines), the difference between the global maximum and minimum time for MPI_Allreduce is more significant than in the former case.

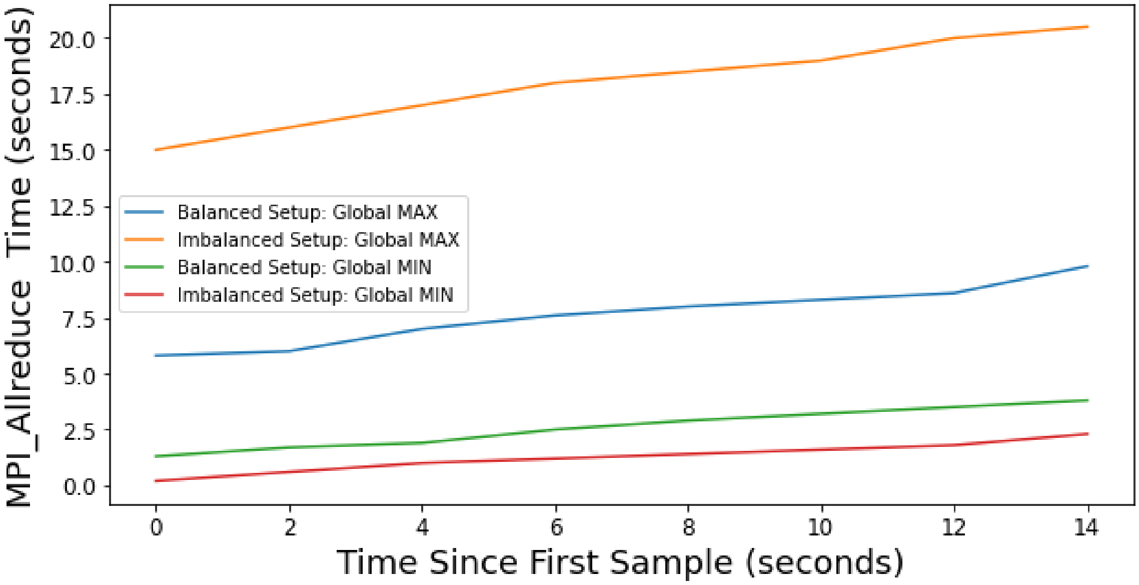
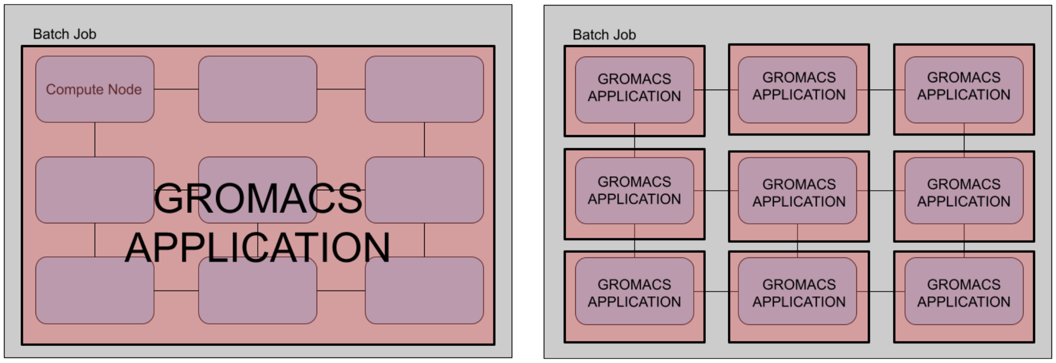


Figure 39. LULESH: Monitoring the Range of MPI_Allreduce Exclusive Time

5.4.3 HPC Ensemble Monitoring: Performance Variation in GROMACS Ensembles. Ensemble computing is an exciting new way of deploying scientific applications on HPC platforms. Ensembles have been used to generate superior scientific results for molecular dynamics applications such as GROMACS [221]. Figure 40 depicts the fundamental difference between how HPC cluster resources are traditionally used to deploy HPC applications and how ensembles are executed. Traditionally, a single, extensive MPI application spans the entire batch job allocation. Over the past three decades, the HPC community has primarily focused on making this single application execute faster and more efficiently on increasingly more significant node counts, thereby accelerating the rate of scientific



(a) Traditional HPC Applications (b) HPC Ensembles

Figure 40. Traditional HPC Applications vs. HPC Ensembles

insight. HPC ensembles take an orthogonal approach, executing many smaller, independent MPI tasks (instances) on the same resource allocation.

The breadth of applications that benefit from an ensemble computing model also spans machine learning applications [23, 222]. Today, ensembles executing on HPC platforms orchestrate 100s to 1000s of individual tasks. Typically, each task is a scientific simulation implemented as a parallel MPI program spanning one or more distributed computing nodes. Adaptive ensembles, depicted in Figure 41 analyze the intermediate results to generate newer generations of ensemble tasks. Intermediate scientific and performance data analysis is employed intelligently, guiding future ensemble workloads.

While adaptive ensembles promise to accelerate science on pre-exascale and exascale platforms, several software challenges need to be addressed before this promise is realized in full effect. One such challenge pertains to enabling robust performance observability and monitoring ensemble components. Performance monitoring entails capturing, exporting, storing, and analyzing all the necessary performance data for different elements in the ensemble system.

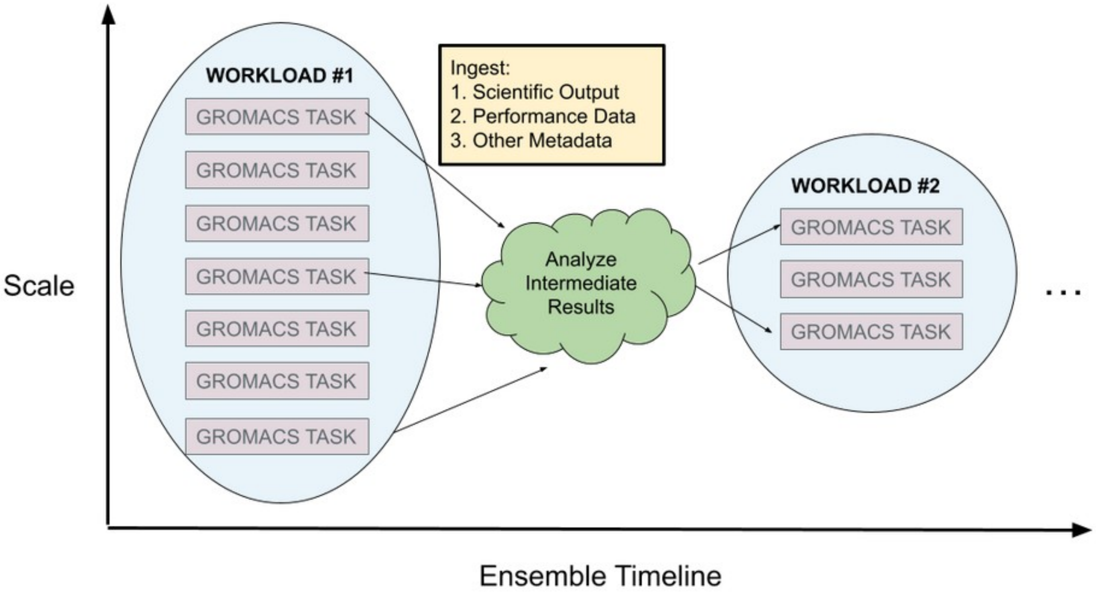


Figure 41. Adaptive Ensembles

5.4.3.1 Performance Variability of Ensemble Tasks. Figure 42 depicts the performance variation resulting from the execution of 2,500 GROMACS MPI tasks on the Theta [1] supercomputer at the Argonne Leadership Computing Facility. The RP ensemble system was set up to execute these GROMACS tasks as a set of ensemble workloads spread across multiple batch jobs. Each MPI task is run on a dedicated Theta KNL node using 64 MPI processes. The key observation from Figure 42 is that these GROMACS tasks experience significant performance variation. Given that these tasks are identical copies operating on the same input, this observation requires further investigation and an appropriate solution to mitigate its effects online.

From the ensemble's viewpoint, it is highly undesirable for there to be a large performance variation between the tasks in the workload for the following reasons:

[1] https://www.alcf.anl.gov/alcf-resources/theta

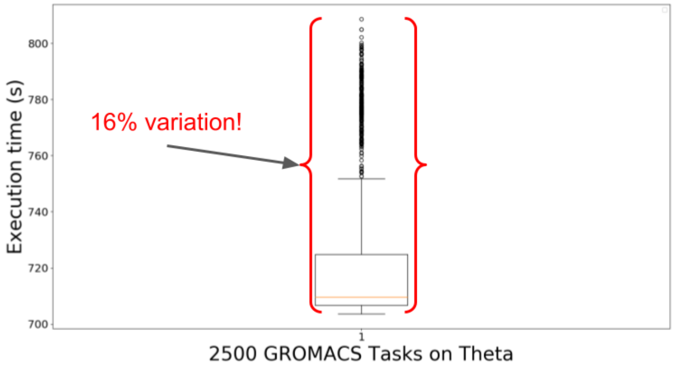


Figure 42. Performance Variation in GROMACS Ensembles

- The overall execution time of the workload is determined by the slowest task in the workload — this could lead to significant wastage of CPU node hours.
- The unpredictability resulting from performance variation makes future task scheduling difficult.
- There is a lack of performance reproducibility of the ensemble's execution.

While understanding the *cause* for the performance variation requires a separate study to be carried out, high-level online performance monitoring and analysis of the task runtimes serve as a first step in mitigating its effects within the ensemble system. The following section describes the design and implementation of an ensemble monitoring solution.

5.4.3.2 Ensemble Monitoring Solution. We integrated the SYMBIOMON monitoring system with the GROMACS ensemble tasks through the TAU plugin described in Section 5.4.1. Figure 43 depicts this integration.

The steps involved in integrating SYMBIOMON with the RADICAL PILOT ensemble are as follows:

- Instrumenting GROMACS: The first step involves gathering the necessary performance data through instrumentation. Here, we are interested in

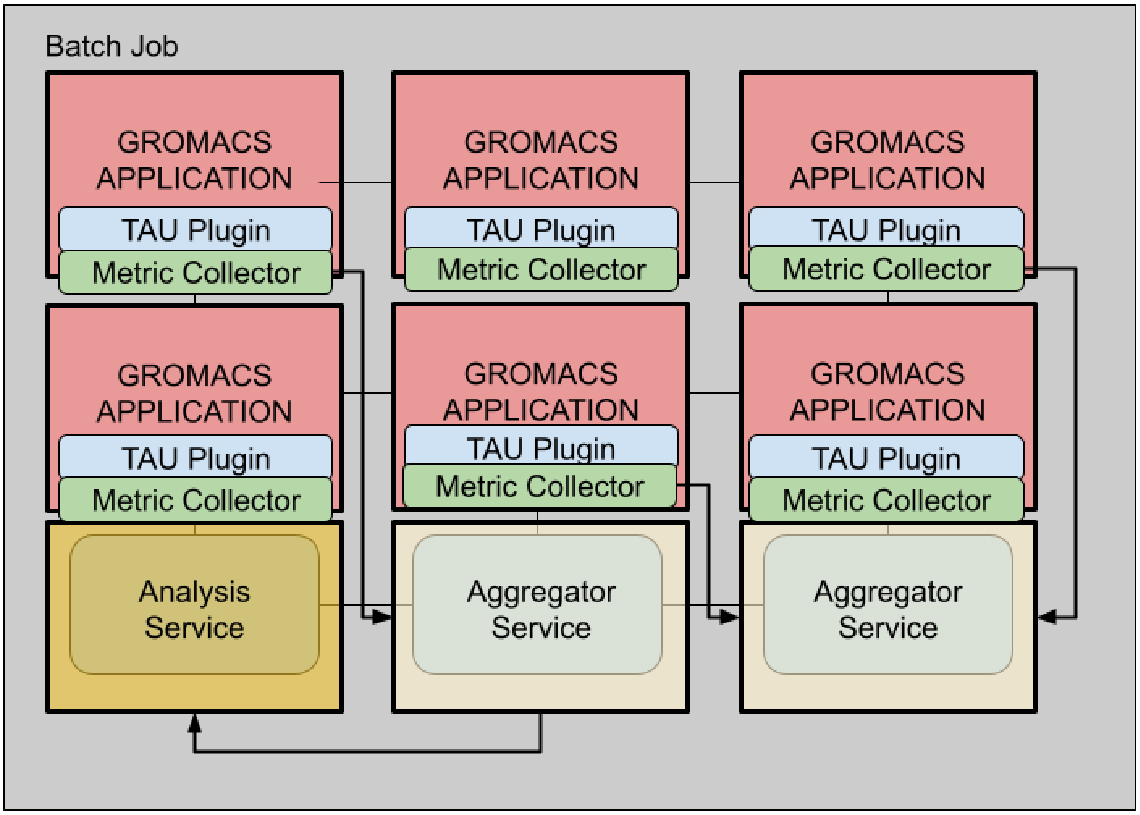


Figure 43. Performance Monitoring of GROMACS Ensembles

tracking and analyzing the *current* runtime of each task in the ensemble as it executes. We achieved this by leveraging TAU's instrumentation capabilities and launching the application with the `tau_exec` script.

- Setting up the AGGREGATOR and ANALYZER microservices: Enabling the online storage and analysis of the captured performance data from each ensemble task requires the AGGREGATOR and ANALYZER to be launched as "special" tasks within the ensemble before the actual ensemble execution begins. These tasks are unique because: (1) there is a strict ordering requirement between the monitoring service tasks and the other tasks with the ensemble, and (2) the AGGREGATOR and ANALYZER run for the entire duration of the ensemble execution, i.e., they are long-running. The monitoring service tasks need to be initialized first as they are required to make their RPC addresses publicly available for the regular ensemble tasks to consume.

- Exporting and analyzing data: The TAU plugin for SYMBIOMON, described in Section 5.4.1 was employed to capture, store, and export data from within each task to the monitoring service, with one modification — the *cohort* in this context comprises all the ensemble tasks. Given that the goal here is to identify the degree of performance variation between these ensemble tasks, we assign the ensemble task with ID "0" as the leader of the cohort. While rank 0 of each task constantly publishes updates of the current execution time of the task, the leader of the cohort is responsible for contacting the ANALYZER service to trigger the corresponding analysis of the current degree of performance variation in the system. The ANALYZER reads partially reduced metric values from the AGGREGATOR to perform this analysis. It reduces this data globally, thereby identifying the minimum and maximum task runtimes (range). This process

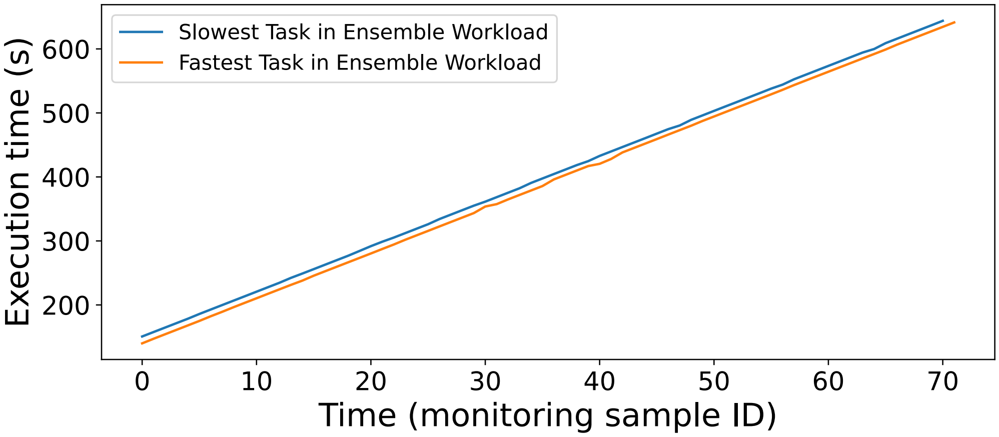


Figure 44. Online Monitoring of the Degree of Ensemble Task Runtime Variation

is repeated at periodic intervals. The ANALYZER exposes this value as an SYMBIOMON time-series metric that can be ingested by a remote monitoring entity such as the Py-COLLECTOR or an adaptivity component within the ensemble system. Figure 44 depicts a plot of the degree of variation of task runtime during actual execution. Although in this particular execution, the degree of variation was not significant, and there are open questions to be answered concerning the methodology for accessing and integrating data from multiple sources, this study demonstrates the utility of service-based monitoring infrastructure.

5.5 Overhead Analysis

5.5.1 TAU Plugin Operation. In this section, we present a study of the performance of some of the plugins described in the section on Usage Scenarios. We pay special attention to the overheads involved in setting up and using multiple plugins together on the Intel Xeon Phi Knights Landing (KNL) processors.

5.5.1.1 Hardware Setup and Software Background. Experiments on Intel KNL processors were performed on the Cori supercomputer[2] at the National Energy Research Scientific Computing Center (NERSC). The KNL partition on Cori has a total of 9,688 compute nodes, where each node is a single-socket Intel Xeon Phi 7250 processor with 68 cores. Compute nodes are connected through the Cray Aries interconnect. The platform-optimized Cray-MPICH was used on Cori.

LULESH is a proxy application that represents a typical hydrocode such as ALE3D. LULESH is implemented using a variety of programming models. We use a version of LULESH that is parallelized using MPI and OpenMP. LULESH is a typical HPC simulation application — it consists of an outer iteration loop that runs for a fixed number of times specified by the input. The problem size scales with the number of MPI processes used. We instrumented LULESH using PDT to retrieve MPI as well as application-level information. We also added TAU annotation to trigger the TAU plugin module (using the TRIGGER plugin interface) at the end of every LULESH major iteration.

All the experiments in our study are run on 64 Intel Xeon Phi KNL nodes with pure MPI being used to parallelize LULESH. 64 processes of LULESH are executed per node for a total of 4096 MPI processes. The length of the cube mesh along each side is set to 60, totaling 884,736,000 elements. Every experiment executes 500 iterations of the LULESH outer loop.

LULESH was executed using the selective tracing plugin in combination with the trigger plugin that aggregates and snapshots TAU profile information globally at the end of every major iteration. The environment variable `TAU_PLUGINS` is used to specify the use of multiple plugins through the command line.

[2] https://www.nersc.gov/users/computational-systems/cori/

XSBench is a mini-app that captures the core computational kernels within the larger OpenMC neutron transport code. XSBench is an MPI+OpenMP code. For our overhead studies, we used pure MPI to parallelize XSBench. We instrumented XSBench using PDT and TAU's PMPI wrapper.

The experiments for XSBench were run on 64 Intel Xeon Phi KNL nodes. 64 MPI processes were executed per node. We experimented with the small problem size with 34 lookups per particle for a total of 500,000 particles. The grid type was set to be unionized. XSBench was executed using the event counter plugin in combination with the load tracking agent plugin.

5.5.1.2 Results. An important aspect of our study is the overhead involved in enabling multiple plugins that perform non-trivial tasks. Table 23 depicts the overheads introduced by enabling the selective tracing and trigger-based profile aggregation plugins. Selective tracing is performed by disabling one or more events. For the study described in Table 23, tracing is enabled only for the MPI events. Meanwhile, *all* the events generated by PDT instrumentation are being profiled by TAU's measurement system. PDT instrumentation alone adds 1.4% of overhead over the baseline (un-instrumented) code. The selective tracing plugin is responsible for an additional 1.4%. Overall, the use of both plugins adds a total of 5.6% overheads over the baseline run. The trigger-based plugin aggregates 29 interval events and 2 atomic events during every LULESH iteration (for a total of 500 times).

Selective tracing is especially useful when one is trying to control the trace file size generated by a large parallel application. We measured the trace file sizes under the following conditions:

- Tracing all events.

- Selectively tracing only MPI events.

Table 23. LULESH: Overheads with multiple plugins

Run Description	Time (seconds)
Default	710
PDT instrumentation	720
PDT + Tracing all events	730
PDT + Selective tracing	730
PDT + Selective tracing + Aggregation	750

Table 24. LULESH: Trace size with selective tracing

Run Description	Trace Size (GB)
Tracing all events	6.6
Tracing only MPI events	5.5
Tracing all events except MPI_Wait	1.1

– Selectively tracing all MPI events except MPI_Wait events.

Table 24 clearly demonstrates that MPI is responsible for a bulk of the events in the trace. Specifically, MPI events account for 83.3% of the trace file size. Further, removing just the MPI_Wait class of events from the trace reduces the trace file size by 77.2% as compared to the default execution that includes all events.

For the XSBench application, we measured the overheads involved in enabling the event counter and load tracking agent plugins. Table 25 suggests that PDT instrumentation by itself does not add much overhead, but enabling the event counter plugin for *all* instrumented events leads to high overheads — over 100x in runtime. The sheer number of instrumented events (specifically, `FUNCTION_ENTRY` and `FUNCTION_EXIT` events) and the frequency of their invocation is a true stress test for our plugin architecture. Of course, we would not use this plugin in this manner. We reason that the source of the overhead is largely due to the hashing function for the strings representing function names.

However, when we used TAU's PMPI wrapper to instrument only the MPI events, the overheads are negligible. The lesson here is two-fold. It is important

Table 25. XSBench: Overheads with multiple plugins

Run Description	Time (seconds)
Default	105
PDT instrumentation	106
PDT + Event counter (all events)	>100,000
PMPI wrapper + Event counter (only MPI events)	106
PMPI + Event counter (only MPI events) + Load tracking agent	107

to consider the impact of the number of events and their frequency when managing plugin overhead. The hashing logic needs to be optimized in the plugin framework. Interestingly, there are no noticeable overheads when the load tracking agent plugin is enabled. It is important to note that *every* MPI process spawns its own agent thread, accounting for a total of 64 agent threads per KNL node. This thread wakes up every 2 seconds to record the load and memory usage.

5.5.2 SYMBIOMON Monitoring and Analysis.

5.5.2.1 Hardware and Software Setup. We present a study of the overheads involved in using the TAU SYMBIOMON plugin to monitor, and we analyze the LULESH performance data. We pay special attention to the process of separating overheads due to the COLLECTOR, the AGGREGATOR, and the REDUCER. All of the experiments are conducted on a testbed consisting of Intel Xeon processors interconnected with a high-speed InfiniBand network. Each Intel Xeon processor hosts 28 CPU cores and 128 GB of memory. For this study we employed 343 LULESH processes spread evenly across 16 nodes. We used a total of 4 AGGREGATOR instances (one per node) with one thread each and a single REDUCER instance to perform the global reduction. Following are the various stages of the process.

- Stage 0: Uninstrumented LULESH binary. The program is built and run without the TAU plugin or any other SYMBIOMON component.

– Stage 1: The COLLECTOR is employed to make measurements at the end of every timestep loop. However, reduction and aggregation support are disabled.

– Stage 2 (frequency “f”): The COLLECTOR is employed to make measurements at the end of every timestep loop. Metrics are reduced locally once every “f” iterations, and the reduced results are sent to the AGGREGATOR. Global reduction is disabled.

– Stage 3 (frequency “f”): The COLLECTOR is employed to make measurements at the end of every timestep loop. Metrics are reduced locally every “f” iterations, and the reduced results are sent to the AGGREGATOR. Rank 0 performs the global reduction (by invoking the REDUCER) every “f” iterations.

5.5.2.2 Results. Figure 45 demonstrates the runtime overheads involved in each step of employing SYMBIOMON to monitor and analyze 250 iterations of the LULESH timestep loop with a problem size of 45. The COLLECTOR does not add any noticeable overhead to the execution. When `tau_exec` is employed to instrument LULESH, 60 metrics are created and tracked on each MPI process. The local reduction and aggregation operations add 2.3–6.5% of the overheads, depending on how frequently these operations are invoked. The locally reduced result for each metric is transferred to the AGGREGATOR by using a dedicated `sdskv_put` RPC call. The global aggregation step (Stage 3) is relatively inexpensive compared with local reduction and aggregation.

5.5.2.3 Aggregator Sensitivity Analysis. Among the three SYMBIOMON components, the number of AGGREGATORS is likely to have the most significant impact on the overall performance of the monitoring workflow. As we have demonstrated here, the COLLECTOR introduces virtually zero overhead because

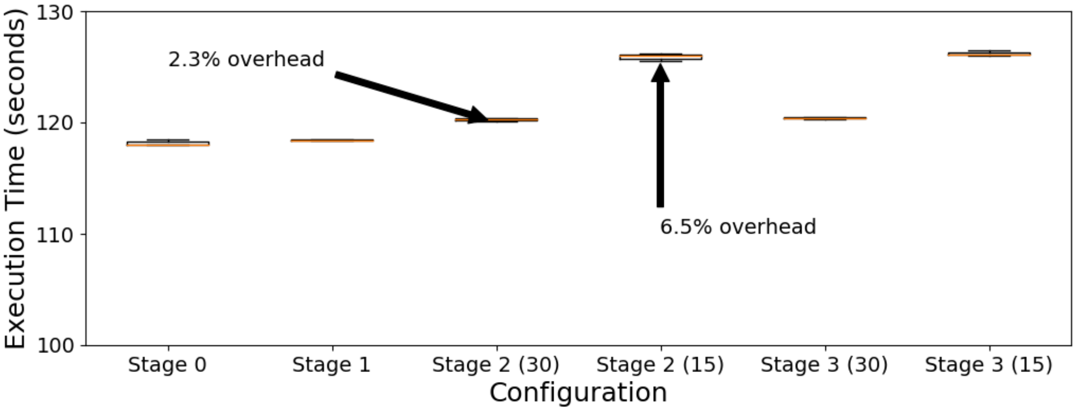


Figure 45. LULESH: Overhead Study

Table 26. Aggregator Sensitivity Analysis

Aggregator Count	Execution Time (s)	Avg. Client Connection Time (s)
2	125.0	0.90
4	126.1	1.1
8	126.2	1.2
16	129.0	1.5

the metric update operations are local to the client. The REDUCER is unlikely to be on the critical path because: (1) it is invoked infrequently and typically only by one client (leader of the cohort), and (2) it performs a single RPC call for each metric being tracked. However, the AGGREGATOR is the central piece coordinating and interacting with all the COLLECTOR and REDUCER instances in the system.

Our experimental analysis reveals that the optimal number of AGGREGATORS in the system is dependent on three critical characteristics of the workload: (1) number of clients interacting with the AGGREGATOR service, (2) the frequency of invocation of AGGREGATOR operations, and (3) the number of metrics being tracked. While performance tuning is necessary for each unique workload, we present the results of employing different numbers of AGGREGATOR instances to monitor the LULESH workflow.

Table 26 suggests that this particular workflow is not sensitive to the number of AGGREGATOR instances in the system. Interestingly, 2 AGGREGATOR instances are sufficient to serve the monitoring needs of 343 MPI processes. However, a deeper analysis of the individual AGGREGATOR operations reveals the contribution of different procedures to the overhead. Before the metrics can be aggregated, each COLLECTOR client needs to be aware of the addresses of all the AGGREGATOR instances in the system and request a connection to them to open the SDSKV database. This requires an `sdskv_open` RPC call to be made for each connection (a one-time cost). SYMBIOSYS [223] instrumentation added to the AGGREGATOR service reveals that the average cost that clients pay for the `sdskv_open` RPC call steadily rises with the number of AGGREGATOR instances. Although it is a one-time cost, this operation is serialized on the server and can lead to client delays. Thus, a trade-off exists between the raw performance of the AGGREGATOR service and the upfront cost that clients pay for balancing the metric load across multiple server instances.

5.6 Limitations

While the monitoring infrastructure described in Chapter V is attractive for serving as a flexible, ubiquitous monitoring solution for in-situ workflows, there are open questions that need to be addressed before this monitoring infrastructure can realize its full potential:

– Data Model: While SYMBIOMON's storage and analysis microservices are broadly applicable to any in situ workflow component, the ubiquitous application of the time-series data model may not be appropriate. Consider an ML workflow involving training thousands of individual tasks, requiring a monitoring solution to store and retrieve information about the ML model being trained. Alternatively, an MPI application may want to report monitoring data in a customized structure. For both these use cases, posing this data as a (set of) time-series data may not be ideal — the data model must be appropriate for the type of analysis required.

- Integration of Custom Reduction Operators: Presently, SYMBIOMON exports a limited set of reduction operators to manage the large volumes of time-series data before they are stored for subsequent monitoring and analysis. This set of operators needs to be extended to support custom reduction schemes that may require the generation, storage, and monitoring of derived metrics. Such support would enable a broader decision-making range for the adaptive execution of distributed services and workflow components.

5.7 Summary

Chapter V presented a ubiquitous monitoring solution for HPC services, applications, and ensembles, thereby addressing the research question **RQ2**. We achieved this goal by combining a plugin architecture for the TAU performance system with the SYMBIOMON monitoring service. SYMBIOMON's composable service architecture offers the user a significant degree of flexibility, making it feasible to monitor any coupled HPC workflow. Notably, SYMBIOMON's microservices were designed and implemented using Mochi components, demonstrating that we can potentially employ these microservices for a broader set of usage scenarios in addition to their original intention for serving the applications' data storage needs. Chapter VI extends the applicability of Mochi microservices, demonstrating their usage for building shared, high performance, in situ visualization services.

CHAPTER VI

EXTENDING THE APPLICABILITY OF HIGH PERFORMANCE MICROSERVICES

This chapter contains unpublished material with co-authorship. The research presented in this chapter results from a collaboration between Dr. Hank Childs, Dr. Allen Malony, and me. The research presented in this chapter is under review at SC 2022 [22]. While performing this research, I received regular guidance from Dr. Hank Childs and Dr. Allen Malony. I did all the experiments, writing, and data collection. Dr. Hank Childs and Dr. Allen Malony helped with suggestions and edits for the paper. All the authors helped in proofreading the submission.

6.1 Introduction

One of the key takeaways from Chapter V is that Mochi microservices have a broader range of applications than the original purpose for which they were designed. Figure 46 depicts the core components of the Mochi framework (Margo, Argobots, and Mercury) and how they are employed to build various microservices. Viewing the Mochi framework as two separate products — (1) out-of-the-box microservices to build custom data services and (2) a core framework on which to build *arbitrary* HPC microservices allows for their extension to other domains of high performance computing applications. Chapter VI demonstrates the extension of these microservices to build a shared, in situ visualization service.

In situ visualization for HPC applications offers a solution to the storage bottlenecks encountered by traditional offline analysis methods. In particular, traditional visualization can require the storage and loading of a potentially large amount of data resulting from the execution of an HPC application (henceforth referred to as the *simulation*). Typically, the simulation code is an iterative MPI

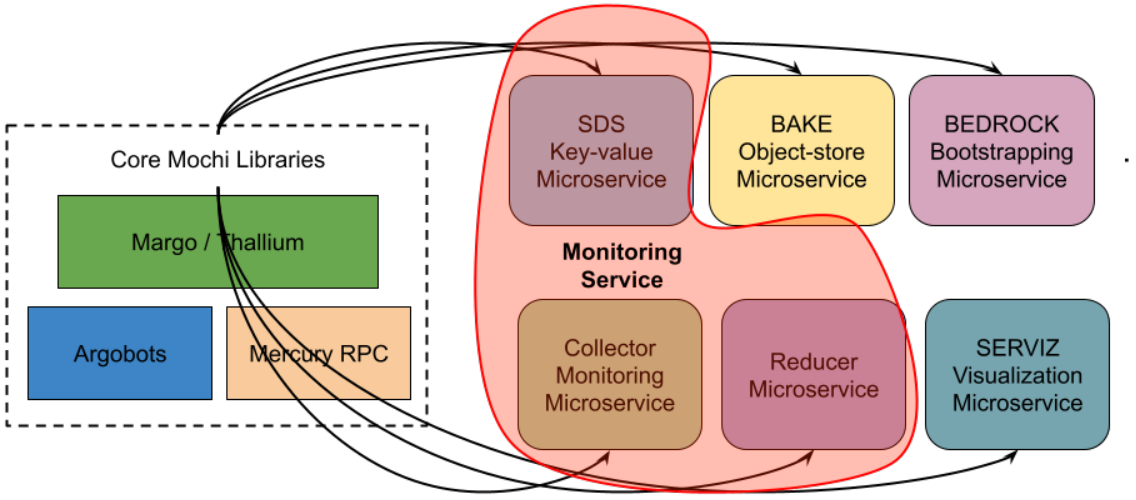
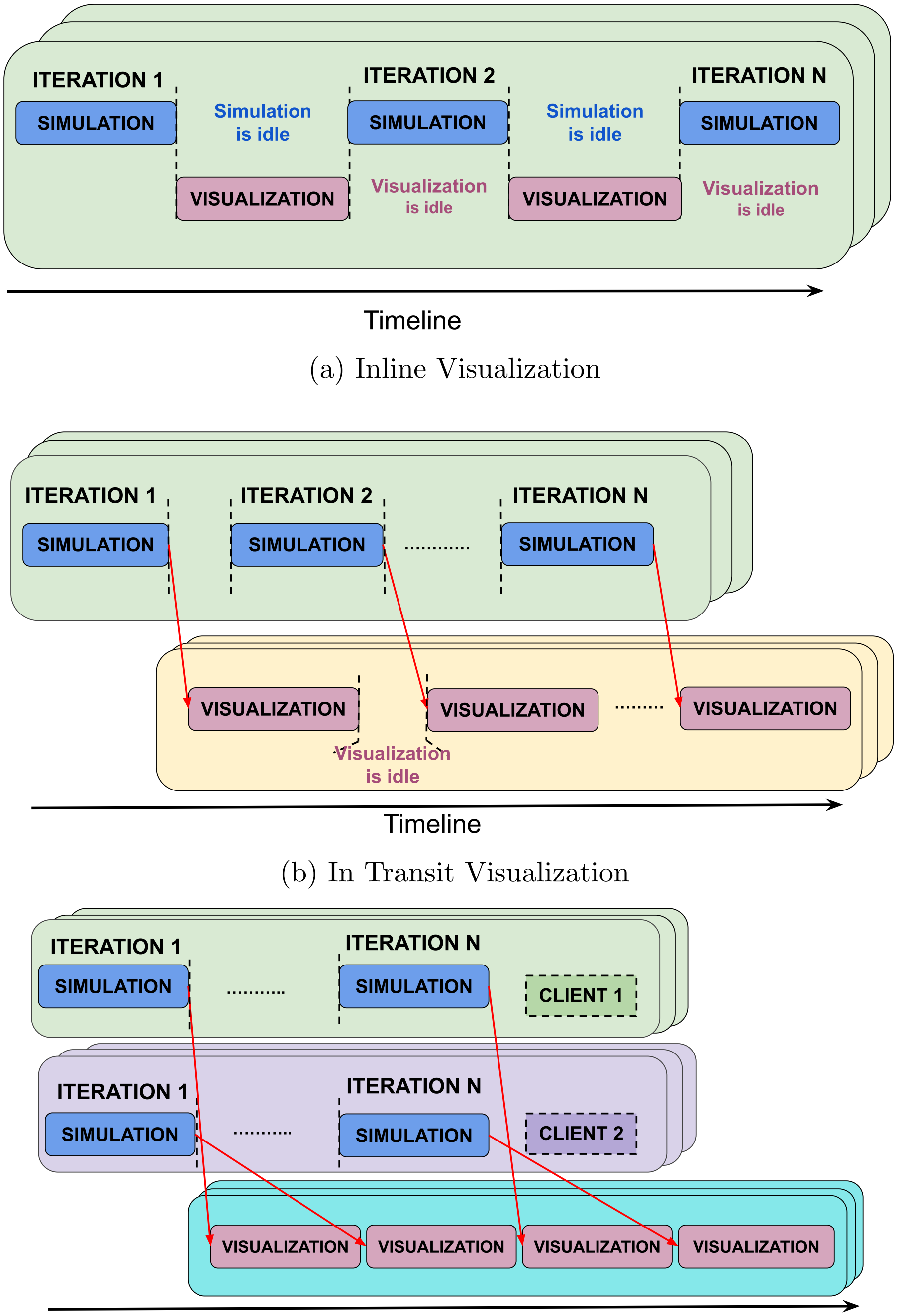


Figure 46. Broad Applicability of Mochi Microservices

application that writes data out to the parallel file system after some number of cycles. Once the simulation has written all the data, a visualization program then processes it to produce results.

In situ schemes avoid the storage overhead by operating directly on the simulation data, effectively circumventing the parallel file system. There are two popular "flavors" of in situ implementations [224]: which we refer to in this paper as *inline* and *in transit*. In the inline scenario, the application invokes the visualization module through a library call while passing along a reference to the simulation data. The simulation and the visualization module run on the same computing resources and time-share the computing resources between themselves. Specifically, the application is idle (blocked waiting) while the visualization module runs and vice-versa. Figure 47a depicts this scenario.

In the in transit case depicted in Figure 47b, the visualization module runs on in transit resources. One of the benefits of running in an in transit configuration is that the visualization can proceed asynchronously and run concurrently with the



(a) Inline Visualization

(b) In Transit Visualization

(c) Service-based In Transit Visualization

Figure 47. Different approaches to in situ visualization. The red arrow indicates data transfer in (b) and (c).

simulation, allowing it to proceed as soon as the visualization data has been safely copied out. Also, the in transit mode allows the visualization program to run at a lower level of concurrency than the simulation, potentially improving the efficiency of the visualization operation [225, 224].

However, there are additional costs associated with in transit — transfer of data to the in transit visualization and the additional in transit computing resources (which may sit idle). That said, previous work by Kress et al. [225] showed the in transit approach can still achieve cost savings: the savings from running at lower concurrency can exceed costs for transfer and additional resources. In their experiments, Kress et al. [225] used the traditional approach of connecting a single simulation code to a single in transit resource that ran at a smaller scale. However, their experiments ran visualization every simulation cycle — much higher than typically used in practice. If they had run less frequently, the in transit resources would have suffered from idle time, which offset any cost savings from running at a lower concurrency. As a result, while their research demonstrated that in transit has opportunities for cost savings, it did not demonstrate these cost savings in configurations frequently used in practice. Our work aims to realize the cost savings demonstrated by Kress in practical configurations, i.e., configurations where simulation codes perform in situ visualization and analysis at lower frequencies.

With this work, we propose a shared service-based approach which we call *SERVIZ*. SERVIZ allows multiple simulations to simultaneously connect to the visualization program (henceforth called the *server*), thereby keeping it busy and reducing the idle time of the in transit resource allocated for it. Figure 47c depicts how these additional cost savings are achieved over the standard in transit model. SERVIZ is developed using Mochi [6], which embodies the principle of composition by utilizing a microservice architecture to create and deploy distributed services. By instantiating SERVIZ as a shared Mochi-based visualization service, the simulation can easily be programmed for in transit operation and benefits directly from an HPC-optimized RPC library [120] for client-server communication. SERVIZ achieves cost savings of up to 12% over the inline configuration and up to 4x reduction in idle time over the dedicated in transit setup. Furthermore, the SERVIZ service-based model offers a programming interface that naturally subsumes the inline and in transit models.

The main contributions of our work are:

– The formulation of a cost model for a shared, in transit visualization service

– The design and implementation of a shared, in transit visualization service based on robust and high-performance microservices technology

– Experimental results demonstrating the cost savings of a service-based, shared, in transit visualization approach over standard inline and in transit models

6.2 Related Work

6.2.1 In situ Visualization. SENSEI [128], Seer [125], and Damaris-viz [133] are three in situ visualization software packages that are capable of space-division (in transit) couplings. SENSEI employs the ADIOS [5] I/O service to allow the transfer of data to a separate executable that implements the visualization operation. Damaris-viz reserves one CPU core on every compute node for visualization processing and allows the transfer of data through shared memory. SEER enables simulation steering of the application through a Mochi [6] service. Meanwhile, Ascent [130] is currently limited to time-division (inline) coupling.

Harvesting unused computing cycles has been explored in the context of inline visualization and analytics. The TINS [129] package leverages work-stealing strategies to execute analytics tasks when there are no available simulation tasks scheduled. GoldRush [226] and Landrush [227] 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.

While in transit visualization can be appealing due to the potential for reducing the simulation execution time through asynchronous operation, the additional computing resources (nodes/cores), the data transfer time, and in transit idle time need to be factored in to estimate if overall cost savings are achieved. Abram et al. [228] explored in transit as a valid design choice for implementing in situ visualization frameworks. Their flexible "ETH" architecture could be configured to explore different strategies to deploy the simulation and analysis/visualization. Kress et al. [225] found that running the visualization module as a separate in transit program with fewer MPI ranks than the simulation has the potential to offset these additional costs. In particular, they identify the "visualization cost-efficiency factor" (VCEF) as a critical parameter for achieving overall cost benefits. Our work builds upon and extends on this observation, with our focus on the frequency of visualization. The Kress experiments performed visualization tasks every cycle, significantly reducing idle time. Simulation codes often run in situ visualization at a much lower frequency — with the approach used by Kress et al., any gains from VCEF would then be wasted as resources sit idle. Our SERVIZ service-based architecture, however, allows the server to be shared. Our approach has the potential to make more efficient use of the in transit resources, i.e., still benefiting from VCEF but not suffering from idle since the resources are kept busy by servicing requests from multiple simulation codes.

6.2.2 Services in HPC. The HPC community has applied service-oriented architectures for deploying distributed applications. DataSpaces [8] employs an RDMA-enabled, transient data staging service for coupling multiple applications within a workflow. ADIOS [5] functions as an I/O processing engine that presents a uniform interface to perform both parallel file storage and data staging. ADIOS is especially useful as a coupling or data staging platform that allows multiple applications to share data asynchronously. GLEAN [154] employs a set of dedicated data staging nodes to buffer data writes to the parallel file system. GLEAN has been used to accelerate the in situ analysis and analytics operations for the FLASH astrophysics code [229]. The Mochi [6] software stack exposes a set of programming tools for building and composing HPC data services. Mochi has enabled the development of a wide range of data services, such as HEPnOS [122, 6], a transient key-value store, and UnifyFS [113], a user-level file system for accessing burst-buffers. Finally, Melissa [230] is an in transit service for performing large-scale sensitivity analysis that bears similarities to SERVIZ in its design. In particular, both follow an explicit client-server model and perform an MxN data re-distribution using a two-level approach. However, the primary motivation for Melissa is to avoid or reduce the number of parallel file system operations — in contrast, SERVIZ aims to provide cost savings over existing in situ analysis techniques.

Huang et al. [231] present a comprehensive review of the state-of-the-art libraries for "in-memory" computing. For all the libraries reviewed, the primary motivation for employing a service architecture was to hide the expensive parallel file system operations. To the best of our knowledge, no existing work has explored the use of a shared distributed service to improve the throughput of visualization operations.

6.3 Ascent Visualization Library

A strength of the service-based approach is that software can be incorporated and integrated without significant effort. As a result, SERVIZ was not built from scratch but rather via existing software. While we considered only a single in situ visualization library (Ascent) for our study, the SERVIZ approach could easily accommodate working with many libraries simultaneously.

Ascent [130] is a lightweight library for in situ visualization designed for use on HPC platforms. Ascent supports distributed-memory parallelism through MPI. It supports shared-memory parallelism on CPUs and GPUs through OpenMP and the NVIDIA CUDA API. The mesh data in Ascent is processed as a Conduit [140] *node*. Conduit nodes are serializable data structures used to describe and represent hierarchical data. This node data can be converted into the human-readable YAML and JSON formats or as plain Internally, Ascent supports several runtimes. Flow [232] is a standard built-in runtime based on Python. Ascent can optionally be built with the VTK-h library that supports distributed-memory parallelism on top of the VTK-m [233] shared-memory implementation. To use Ascent, the simulation needs to convert the simulation data into a Conduit node and pass it to Ascent through Ascent's API. The Ascent API consists of four main routines that are briefly discussed below:

– ***Open***: This call initializes Ascent and takes as a single parameter a Conduit node representing "options" for which visualization runtime to use and the MPI communicator specification, if operating within a distributed setting. OpenMP can be also be used to parallelize visualization operations within a process. This API is invoked once, outside the main simulation loop.

- **Publish**: The publish API call expects a single, valid Conduit node as input, specifying the mesh data on which to perform visualization operations. Internally, Ascent stores this data and returns control back to the simulation. This API call is invoked inside the simulation loop.

- ***Execute***: The execute call takes as input a Conduit node specifying one or more "actions" to apply to the published mesh data. *Scenes* are used to create images from the simulation data. *Pipelines* data into other derived forms to be further analyzed. *Extracts* are used to move data out of Ascent, potentially into other file formats such as ADIOS or HDF5. *Queries* are used to evaluate and *Triggers* are used to execute a set of actions based on certain conditions. The execute call is also invoked inside the simulation loop (following publish).

- ***Close***: The call takes no input arguments. This call finalizes the Ascent library and invokes a cleanup operation.

The *publish-execute* call sequence fully describes the data and the visualization actions. Individual call sequences are independent of each other and require the storage of no state information in between. Therefore, they can be executed in any order so long as the publish and execute operations are performed in an atomic manner. Ascent's API and execution model significantly reduces the complexity of implementing a shared, distributed visualization service.

6.4 SERVIZ: A Shared Visualization Service

6.4.1 Original In transit Cost Model.

6.4.1.1 Terminology and Base Cost Model. While comparing the costs of an in transit scheme relative to the inline version, Kress et. al. [225] discovered that running the in transit application at a lower level of concurrency as compared to the simulation can improve the cost efficiency of the visualization operation. To this end, they defined the "visualization cost-efficiency factor" (VCEF) as the ratio of the total node seconds to perform visualization at two different concurrency levels. Any in transit implementation will involve additional costs over and above the equivalent inline implementation. These include costs for the transfer of data and the additional in transit node resources. The original VCEF base model for the in transit scenario assumes that each client is assigned a dedicated in transit visualization program. Specifically, cost savings for in transit over inline is achieved when:

$$N_{sim} * (T_{sim} + T_{inline}) > N_{sim} * (T_{sim} + T_{client}) + N_{intrans} * T_{intrans} \tag{6.1}$$

Table 27 defines the terms used in this model. Expanding the terms T_{client}, $T_{intrans}$, $VCEF$, and σ gives us:

$$T_{client} = T_{send} + T_{gather} \tag{6.2}$$

$$T_{intrans} = T_{recv} + T_{idle} + T_{delay} + T_{viz} \tag{6.3}$$

$$\sigma = N_{intrans}/N_{sim} \tag{6.4}$$

$$VCEF = T_{inline}/\sigma * T_{viz} \tag{6.5}$$

In practice, we find that T_{recv} and T_{delay} are zero and the client pays the cost of serializing and sending data to the server. Plugging in these expanded terms into Equation 6.1 and simplifying, we get:

$$T_{inline} * (1 - \frac{1}{\sigma * VCEF}) > T_{send} + T_{gather} + \sigma * T_{idle} \tag{6.6}$$

In general, M simulation processes are coupled with N in transit visualization processes. When N is less than M, σ is less than 1, implying that simulation clients incur a cost needed to gather the data before it is sent over to the N in transit visualization processes. Note that this model assumes that there exists no "in-between" layer such as ADIOS to perform data staging. In other words, data is transferred directly from simulation process memory to the in transit application without any data staging nodes. T_{gather} represents this data gathering cost. When M is equal to N, T_{gather} is zero.

The idle time on the in transit application, denoted by T_{idle} is a natural outcome of an in transit coupling. When there is only one simulation coupled to an in transit visualization program, reducing the idle time on the in transit resources can be difficult to achieve, requiring careful tuning of the MxN ratio. Further, to keep the in transit resources busy, the application would need to invoke visualization operations more frequently than needed. Otherwise, the benefits from this mode of operation may not be able to justify the additional costs involved.

6.4.1.2 Shared-service Cost Model. The fundamental motivation to have multiple simulations share the same in transit service is to reduce T_{idle}. The other benefit of having multiple simulations connect to a shared, in transit server instance is to do away with the issue of having to invoke the visualization program more frequently than needed. The application can invoke the visualization as frequently as it would like, while the service can take on additional clients if it notices that the T_{idle} is large. Essentially, keeping the service as busy as possible ensures the full utilization of in transit resources and a better chance of achieving cost savings with an in transit implementation. In the shared scenario with C (homogeneous) application clients sharing a single in transit service, Equation 6.1 becomes:

$$C * N_{sim} * (T_{sim} + T_{inline}) > \\ C * N_{sim} * (T_{sim} + T_{client}) + N_{intrans} * T_{intrans} \quad (6.7)$$

Under the simplifying assumptions where there are no server delays, the simulation pays the data transfer cost for the RDMA operation, and the server is continuously busy doing only visualization work, then:

$$T_{intrans} = T_{viz} \quad (6.8)$$

Plugging these back in to Equation 6.7 and simplifying, we get:

$$T_{inline} * (C - \frac{1}{VCEF}) > C * (T_{send} + T_{gather}) \quad (6.9)$$

The problem size determines the T_{send} value and the MxN ratio determines the T_{gather} time.

6.4.2 Service Architecture and Implementation. SERVIZ is implemented as a Mochi microservice. This microservice has two components — the client library and the server library. The client library exposes RPC APIs that a visualization client (simulation) can invoke directly. The RPC API on the server-side is implemented as member functions within a service *provider* — a uniquely addressed object that can receive and execute RPC calls. Each provider is associated with one or more Argobots work queues from which it can pull and execute RPC requests. The provider can also be assigned a set of dedicated OS threads to execute multiple RPC requests concurrently. Mochi allows the instantiation of one or more service providers on a server process.

6.4.2.1 SERVIZ API. Table 28 lists and describes the SERVIZ client RPC APIs. These APIs mimic the functionality exposed by Ascent (see Section 6.3). All these APIs are implemented as asynchronous RPCs, i.e., the simulation does not

Table 27. Cost Model: Definition of Terms

Term	Description
N_{sim}	Number of simulation nodes
$N_{intrans}$	Number of in transit visualization nodes
T_{sim}	Total simulation time (excluding visualization)
T_{inline}	Total inline visualization time
$T_{intrans}$	Total in transit visualization time
T_{client}	Total client time for in transit operations
T_{send}	Total time to send simulation data
T_{recv}	Total time to recieve simulation data
T_{gather}	Total time to gather simulation data in MxN coupling
T_{idle}	Total idle time on the server
T_{delay}	Total time for in transit server delays
T_{viz}	Total time for in transit visualization operations
C	Number of application clients per in transit server
σ	Ratio of in transit nodes to simulation nodes
$VCEF$	Visualization cost efficiency factor

Table 28. SERVIZ API Description

`serviz_open`	Initializes Ascent library through ascent::open()
`serviz_close`	Finalizes Ascent library through ascent::close()
`serviz_publish`	Invokes Ascent publish API through ascent::publish()
`serviz_execute`	Invokes Ascent execute API through ascent::execute()
`serviz_publish_execute_atomic`	Invokes Ascent publish,and execute calls atomically
`serviz_execute_pending_requests`	Executes pending (stored) requests on the server

have to block for the execution of the RPC to complete on the server. Once the data is safely copied out of the simulation buffer, the API call returns a request handle to the simulation, and the simulation code can proceed with its computation. This request handle can be used to query the status of the RPC call at a later point in time. Among the APIs listed in Table 28, the `serviz_publish_execute_atomic` API holds particular importance in the context of a shared service. This API takes the Conduit nodes holding the Ascent options, the mesh data to be published, and the Ascent actions to execute on the mesh data as input arguments. When multiple simulations are simultaneously invoking Ascent operations on the server, it is necessary to have each simulation *fully describe* the visualization operation to prevent the intermediate storage of client-specific state information on the in transit server. In other words, atomically executing the ascent::publish() and ascent::execute() routines inside a single RPC function on the server simplifies the service implementation.

6.4.2.2 SERVIZ Implementation. The SERVIZ client library is a *stub* capable of serializing and transferring data for the RPC call through the Mercury library (see Section 6.3). The service itself is an MPI executable that implements distributed-memory parallelism in Ascent. MPI is not used for the transfer of RPC data to the service. Each MPI process within the server instantiates a single SERVIZ provider object. This SERVIZ provider runs within the context of the primary process thread, i.e., no extra OS threads are used. Also, each provider is associated with a dedicated Argobots work queue.

The SERVIZ provider object implements the RPC APIs presented in Table 28. The first four APIs listed in Table 28 result in a direct invocation of the corresponding Ascent API call after the Conduit nodes are parsed from the RPC arguments. Notably, these APIs can be invoked *only* in a dedicated in transit server setting. When SERVIZ is shared amongst multiple simulations, each simulation invokes the `serviz_publish_execute_atomic` API call. Consider the following scenario that can occur when executing SERVIZ in a shared setting. There are simulations $\mathbf{s_1}$ and $\mathbf{s_2}$ sharing a service consisting of two processes $\mathbf{p_1}$ and $\mathbf{p_2}$. At some point during the execution, $\mathbf{p_1}$'s work queue contains the requests $\mathbf{r_1}$ (from $\mathbf{s_1}$) and $\mathbf{r_2}$ (from $\mathbf{s_2}$) in that order. The work queue on $\mathbf{p_2}$ contains the same requests, but in the reverse order. Given that each RPC request is internally processed as a sequence of MPI calls (for Ascent) involving both $\mathbf{p_1}$ and $\mathbf{p_2}$, these processes must be synchronized and in agreement concerning which request they are currently processing. However, the SERVIZ providers on each process execute RPC work requests from the work queue in the order in which they were inserted (FIFO). Note that this situation arises because of the inherent non-determinism in a multi-client setting and cannot be avoided. Given that SERVIZ is *both* — an RPC-based service as well as an MPI program, this situation needs to be addressed to ensure the correct operation.

MPI rank 0 of each simulation broadcasts a high-resolution timestamp to its peers to remedy this situation. This timestamp is an additional argument to the `serviz_publish_execute_atomic` API call. On the SERVIZ provider, this timestamp and the Conduit node data are retrieved and stored in a `priority_queue` (heap) that partially sorts the requests based on the timestamp. Each SERVIZ provider MPI process then pulls the top element from the queue and performs an MPI_Allreduce to check if its peers agree with respect to the request ID. If they are, the Ascent computation proceeds in parallel. If they are not, this request is stored for future execution using `serviz_execute_pending_requests`. The latter scenario can occur due to delays in receiving and processing the RPC request at the network layer (beneath Argobots).

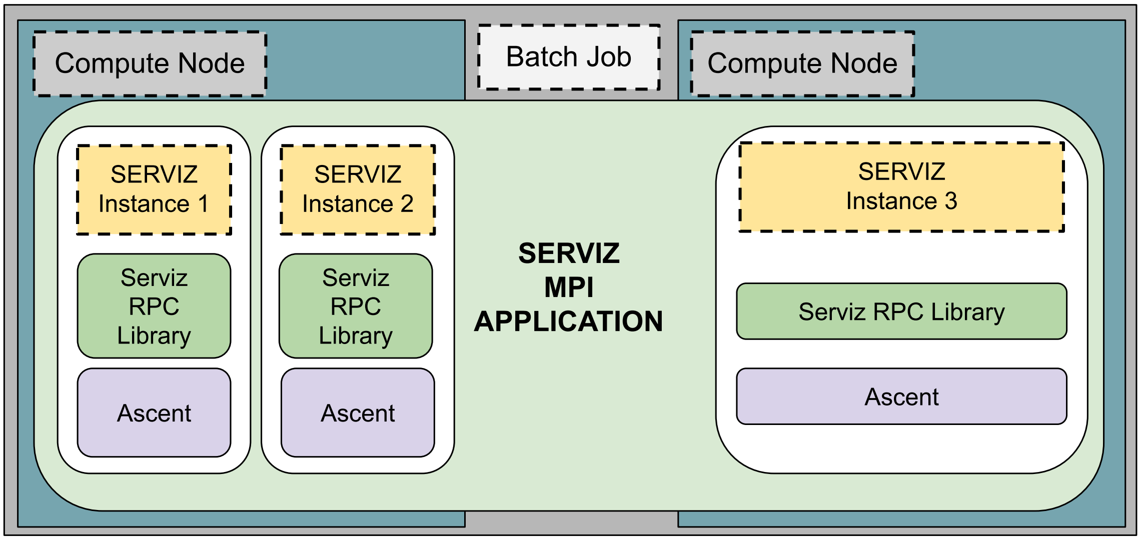


Figure 48. SERVIZ: Division Into Multiple Instances

6.4.2.3 SERVIZ Execution Model. The SERVIZ MPI processes can be divided into one or more service instances as depicted in Figure 48. This strategy benefits large HPC clusters that do not allow multiple MPI applications to share a single compute node. Each service instance is given its own MPI communicator to use for performing parallel Ascent operations.

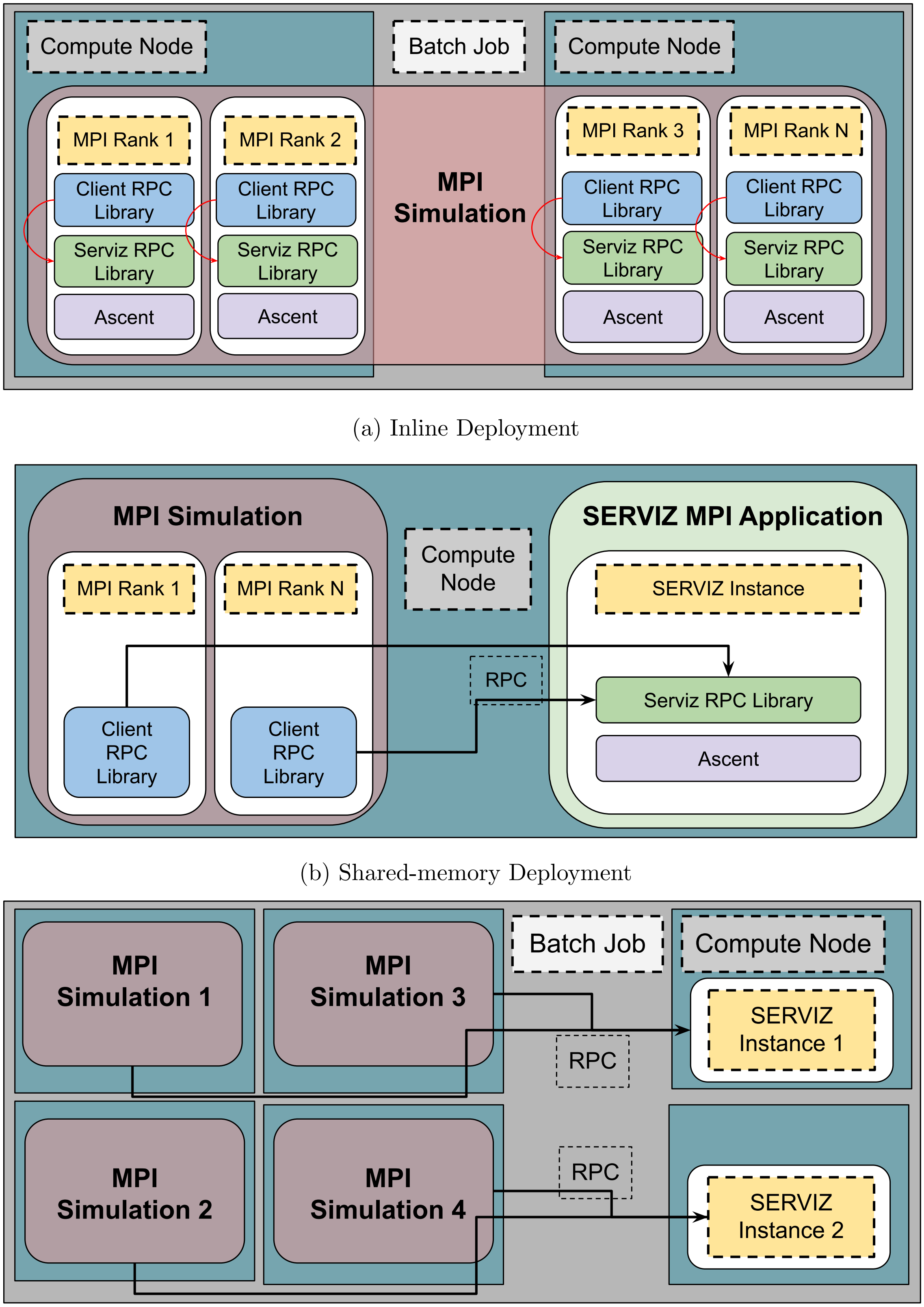
The various steps in the invocation and the execution of the service are described below:

- **Generation of Server Addresses**: The service 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 SERVIZ provider and makes its RPC address public through a file. The service is now ready to accept requests.
- **Discovering and Connecting to Servers**: Each simulation is provided access to a specific instance of the service (among those available) through an environment variable. During its initialize phase, each simulation process reads the service addresses from the corresponding address file and creates a SERVIZ client object through which it can make RPC calls. This client object is created once. If the simulation notices that the number of server ranks is less than the number of client ranks (MxN, where M > N), it sub-divides its MPI_COMM_WORLD into N sub-communicators such that the number of ranks in each sub-communicator is M/N. This step is not necessary in an MxN coupling where M = N. Note that we make a simplifying assumption that M divides N exactly.
- **Gathering Data on the Simulation**: At each iteration where visualization is to be performed, rank 0 of each simulation sub-communicator gathers the Conduit mesh data from its peers using MPI. This step is necessary to ensure correctness of the visualization operation when M > N.
- **Execution of the RPC**: Once the mesh data has been gathered, rank 0 of each simulation sub-communicator invokes the SERVIZ API to perform the RPC call. Note that the SERVIZ API call is asynchronous and returns as soon as the data has been safely copied out. The simulation ranks that do not participate in the RPC

call continue on their normal execution path. The service operation is described in Section 6.4.2.2.

6.4.2.4 SERVIZ Deployment Configurations. One of the benefits of a service-based model is the flexibility in choosing the SERVIZ deployment configuration. Figure 49 depicts three ways in which SERVIZ can be deployed and coupled alongside the simulation. Figure 49a is a SERVIZ deployment configuration that mimics the (default) inline implementation. The SERVIZ provider is instantiated on the simulation MPI process in this configuration. Therefore, the RPC calls between the simulation and the SERVIZ provider are implemented as regular function calls. Figure 49b depicts a situation where SERVIZ shares the compute node with the simulation. This mode benefits from using shared memory to transfer the data for visualization. It is beneficial on clusters that allow multiple MPI applications to share a compute node. Note that Damaris-viz [133] can be viewed as a particular case of this deployment configuration — where one core on the node is reserved for the visualization service. Finally, Figure 49c is an in transit deployment configuration that allows SERVIZ to run on a dedicated set of compute nodes. This mode allows SERVIZ to operate as a shared visualization service.

6.4.2.5 SERVIZ Operating Modes. SERVIZ operates in two modes — *Immediate* and *Delayed*. In the *Immediate* mode, SERVIZ processes a visualization request at the earliest possible time, i.e., as soon it receives the request. In contrast, when SERVIZ operates in a *Delayed* mode, the visualization data corresponding to the request is stored for future execution. In the Mochi RPC execution model, once the RPC execution begins, the RPC does not yield control of the OS thread until the RPC execution is complete. The same OS thread is also responsible for progressing Mercury RDMA communication to pull client data. Therefore, the mode of operation



(a) Inline Deployment

(b) Shared-memory Deployment

(c) Shared, In transit Deployment

Figure 49. SERVIZ: Deployment Configurations

determines the maximum number of simulations that can be coupled with a shared service instance without causing a simulation-side delay. In the *Delayed mode*, the SERVIZ provider continues to store simulation requests until one of two conditions are met — the SERVIZ provider realizes that there are no more simulation requests in the queue, or the `serviz_execute_pending_requests` RPC call is invoked. The SERVIZ provider begins processing the queued requests and does not yield control until completion.

6.5 Evaluation

AMR-Wind was chosen as the simulation to evaluate SERVIZ as a shared visualization service. AMR-Wind, part of the ExaWind [234] project, is a massively parallel, adaptive mesh solver used to run wind-farm simulations. AMR-Wind is an MPI-based code with OpenMP and NVIDIA CUDA support for leveraging shared-memory parallelism. AMR-Wind is integrated with the Ascent inline visualization framework through the AMReX [235] library. AMR-Wind is an iterative code that uses an input deck specifying the problem size (grid size), the (maximum) number of time steps, and the Ascent invocation frequency. We used two problems sizes in our evaluation: **Small** ($160x64x48 = 491,520\ cells$) and **Large** ($320x64x48 = 983,040\ cells$). The number of AMR-Wind time steps was fixed at 50 for all our experiments.

6.5.1 Setup.

6.5.1.1 Hardware. All experiments were run on the Theta supercomputer at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray XC40 system hosting an aggregate of 4,392 Intel Knights Landing (KNL) nodes. Each KNL node has 64 compute cores, 192 GB of DDR4 memory, and 16 GB of high-bandwidth

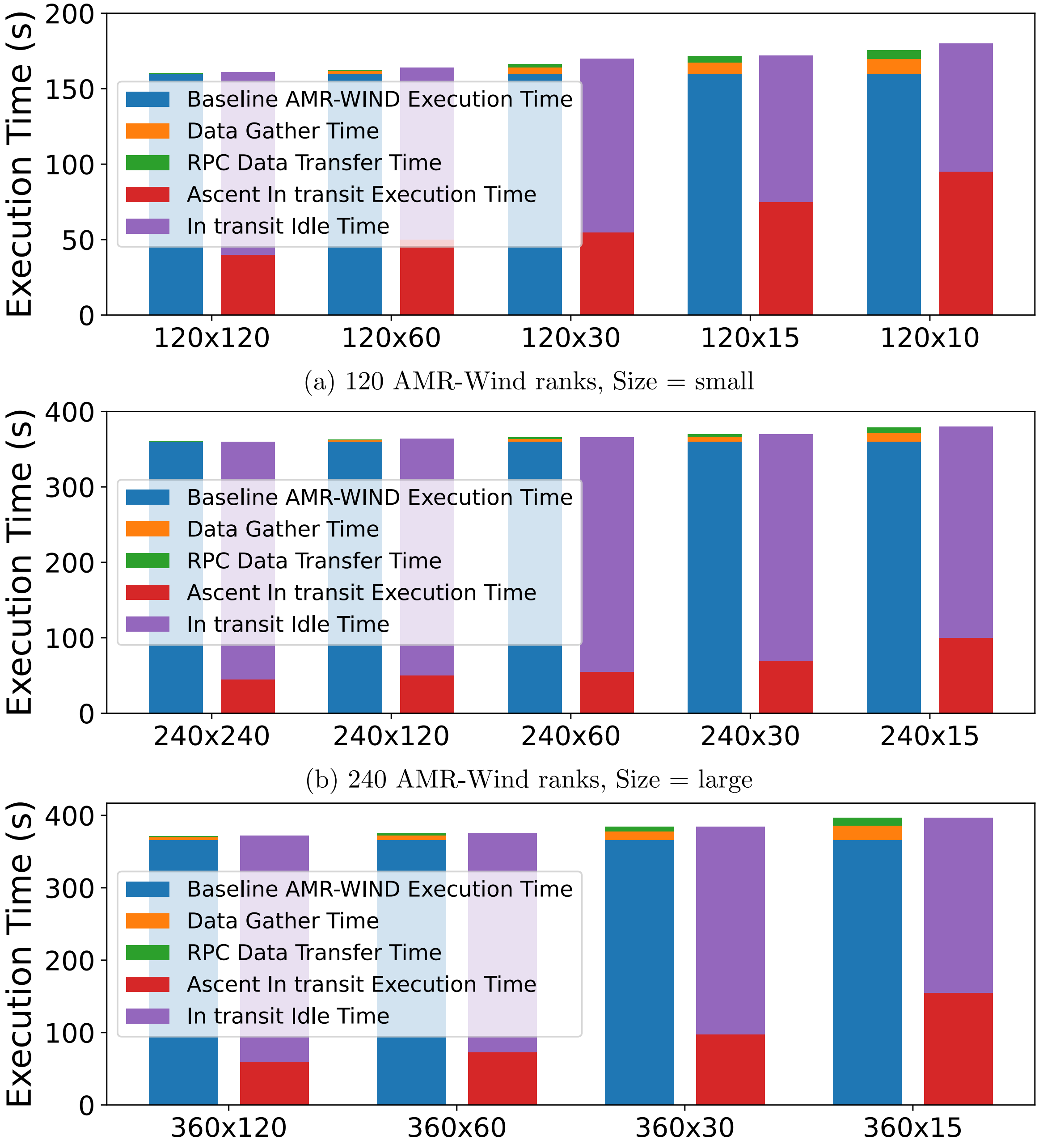
MCDRAM memory. The MCDRAM memory can either be configured in cache mode or flat mode. All the experiments were conducted with the MCDRAM in cache mode.

6.5.1.2 Software. SERVIZ and its Mochi component dependencies were installed using the Spack [189] package manager. The RADICAL Pilot [236] ensemble system was employed to set up multiple, simultaneously executing instances of the AMR-Wind application as a way to emulate a multi-client, shared-service environment. The latest development versions of Ascent, Conduit, VTK-h, and VTK-m were installed using the Spack package manager. AMR-Wind and Ascent were built without OpenMP and GPU support. For all our evaluations, a maximum of 60 (simulation or server) MPI ranks were placed on a single KNL compute node.

6.5.2 Single Client Experiments. Single client experiments were conducted to understand the costs of gathering the data in an MxN setting (T_{gather}), sending the data using RPC (T_{send}), performing in transit visualization (T_{viz}), and the idle time (T_{idle}) on the in transit resource for different simulation and server process counts. Table 29 depicts the baseline values of the simulation execution time (T_{sim}) and the inline Ascent visualization time (T_{inline}) for AMR-Wind process counts of 120 (2 KNL nodes), 240 (4 KNL nodes), and 360 (6 KNL nodes). When visualization is performed at every AMR-Wind iteration, it is observed that Ascent operations account for 20% of the total execution time at a 120-process count, 13% at a 240-process count, and 12% at a 360-process count. Note that SERVIZ was set up to run in *Immediate* mode to mimic the inline execution model.

Table 29. AMR-Wind Inline Visualization Time

# Ranks	**AMR-Wind Baseline (T_{sim}) (s)**	**Ascent (T_{inline}) (s)**
120	162	40
240	360	53.5
360	366	48.0



(a) 120 AMR-Wind ranks, Size = small

(b) 240 AMR-Wind ranks, Size = large

(c) 360 AMR-Wind ranks, Size = large

Figure 50. Single Client In transit Visualization Experiments. X-axis Represents the Simulation x Server (MxN) Process Counts.

Figure 50 depicts a detailed break-up of the execution time for a single-client in transit coupling using different process counts. Across the three simulation process counts, several common observations can be made. First, in the SERVIZ implementation, the simulation pays the cost of gathering the data and transferring the data to the server using RPC. Second, these costs of gathering and transferring the data steadily grow with the MxN ratio. Third, the wall-time for performing visualization on the in transit resource also grows with the MxN ratio. This observation is expected as the concurrency level for performing visualization drops with a larger MxN ratio. However, the most important observation to make in this context is that the server has plenty of idle time in all the configurations tested. For example, as Figure 51 depicts, when there are just 15 in transit server processes coupled (*Immediate* mode) to 120 AMR-Wind simulation processes, the idle time accounts for nearly 60% of the total in transit server time. When each simulation is coupled with a dedicated in transit server, it is difficult to fill up this idle time on the server. The only option here would be to reduce the number of in transit server processes until the idle time is minimized. Even with this fine-tuning, there could still be idle time if the application invokes the visualization infrequently.

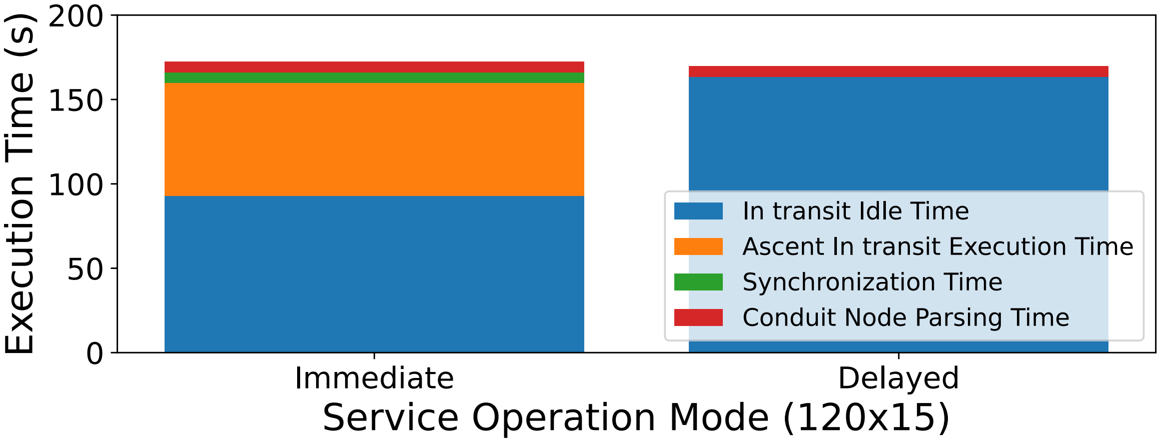


Figure 51. Server Idle Time in Different Modes (Single Client)

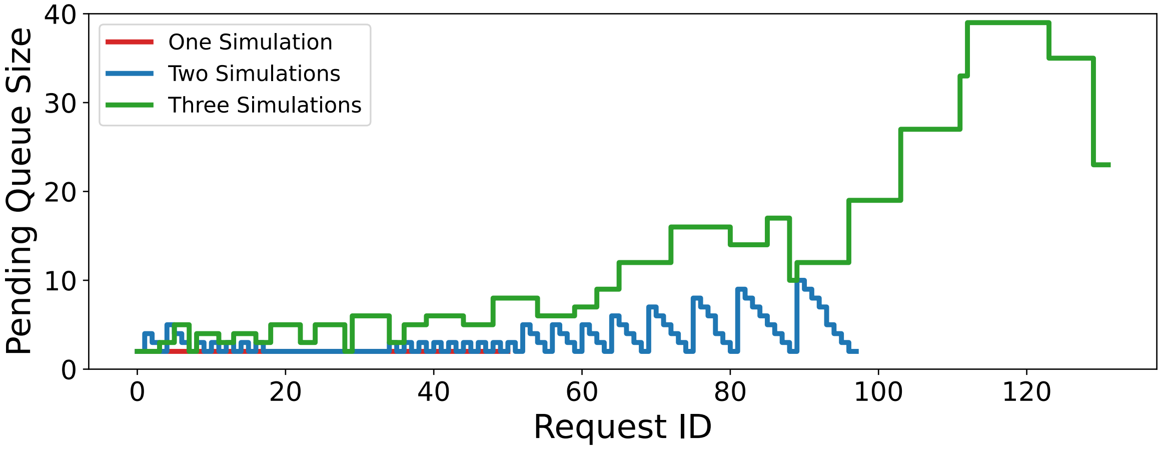


Figure 52. Pending RPC Queue Size: Immediate Mode (120x15)

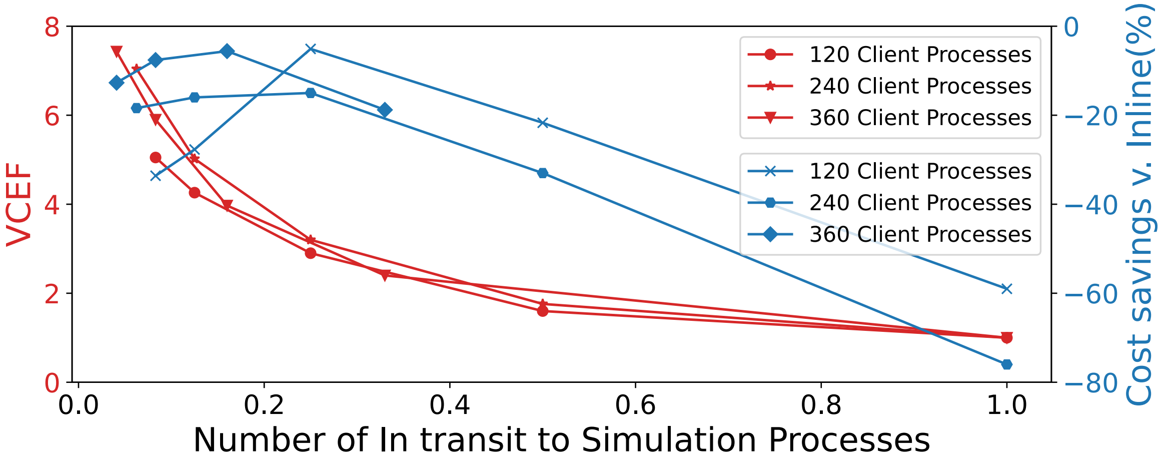


Figure 53. VCEF and In transit Cost Savings (Single Client)

Recall that the motivation for reducing the concurrency level of the visualization is to achieve cost savings through VCEF. A high VCEF value is indicative of a higher potential for cost savings. Figure 53 is a plot of the VCEF values and cost savings achieved for a range of MxN couplings for each of the three simulation process counts — 120, 240, and 360. These MxN values are derived from Figure 50. With a server to simulation process ratio of 0.041 (corresponding to a 360x15 coupling), for example, the VCEF value achieved is 7.4, stating that in transit visualization with 15 processes is 7.4 times more efficient than the default inline implementation at a 360-process scale. Even with this high VCEF value, the dedicated in transit implementation cannot achieve cost savings over the inline implementation. Specifically, due to three factors, a 360x15 in transit coupling is 10% less cost-effective than the default inline case. First, the data gathering cost (T_{gather}) and the RPC data transfer cost (T_{send}) increase with the MxN ratio. Second, there is significant idle time on the in transit server. Third, the cost calculation considers the total number of nodes used and not the total number of cores (processes). This methodology is consistent with how most job schedulers on HPC clusters such as Theta allocate and charge for resources, i.e., at node-level granularity. Further, these job schedulers do not allow multiple applications to share the same compute node, limiting the number of MPI programs that can be launched on the compute node to one.

6.5.3 Shared-Server Experiments. The shared-service model realized through the SERVIZ architecture can address some of the shortcomings of the dedicated in transit server approach. In particular, the shared-service goal is to reduce the idle time on the server and make efficient use of in transit node resources. Recall that SERVIZ operates in two modes — *Immediate* and *Delayed*.

6.5.3.1 Immediate Mode. In the *Immediate* mode, the server processes the visualization requests at the earliest possible time after each server process has "seen" the request. When there are multiple simulations coupled to a server, this can reduce the idle time on the server. However, as detailed in Section 6.4.2, operating the server correctly under a multi-simulation, *Immediate* setting necessarily requires synchronization across server processes. As shown in Figure 51, this synchronization adds a small but non-zero cost to the overall request processing time. When determining the "optimal" number of clients (C) per server instance, the objective is to minimize the idle time (T_{idle}) without causing visualization requests to pile up on the server.

Figure 54 depicts the results of coupling one, two, and three 120-process AMR-Wind simulations with a 15-process server instance. When two AMR-Wind clients are coupled with the 15-process server, the idle time on the server drops from 77 seconds (with one client) to 41 seconds. The total time spent doing visualization goes up from 39% to 70% with two clients with no impact on simulation execution time. When three clients are coupled with the server, the total percentage of time spent doing visualization goes up by 2%. However, the simulation execution time increases by an average of 52 seconds. This is an artificial limitation of the Mercury RPC implementation, *requiring* asynchronous RPC requests to be coupled with a `wait()` call placed at the end of the simulation execution (before `MPI_Finalize()`) to ensure that all the data for visualization is safely copied out of simulation memory before Mercury is finalized. Future Mercury releases will do away with this unnecessary need for blocking the simulation for asynchronous RPC calls. Figure 52 is a depiction of the Argobots pending RPC queue sizes (sampled over time) for each of these three couplings. A large and growing value for the pending queue size suggests overloading the server. While the simulation's asynchronous RPC invocation strategy can afford to absorb the impact of some of this overloaded server state, it appears that when three simulations are coupled to a 15-process server, this leads to significant wait times on the AMR-Wind simulation. Therefore, the lesson here is to carefully monitor the Argobots queue size and the simulation-side wait times to determine the optimal number of simulations per server instance.

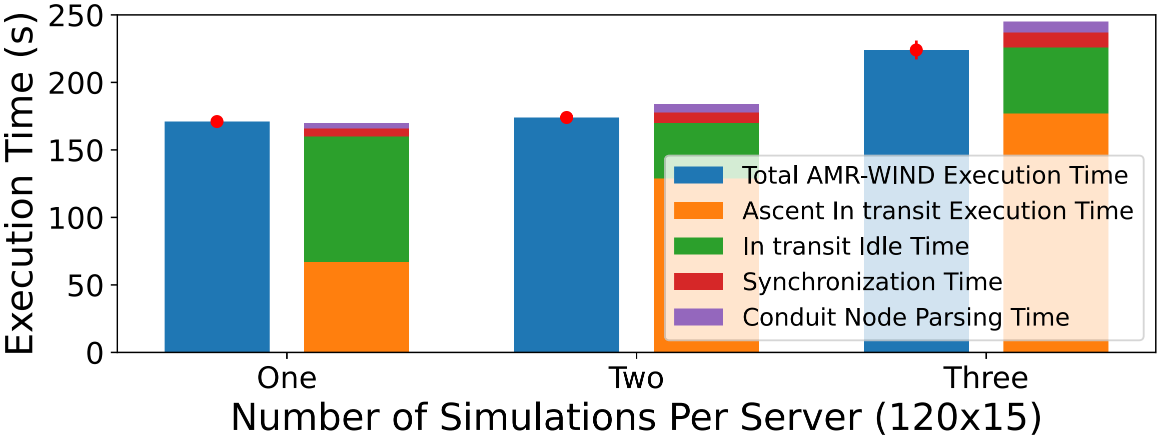


Figure 54. Multiple Simulations/Server: Immediate Mode (120x15)

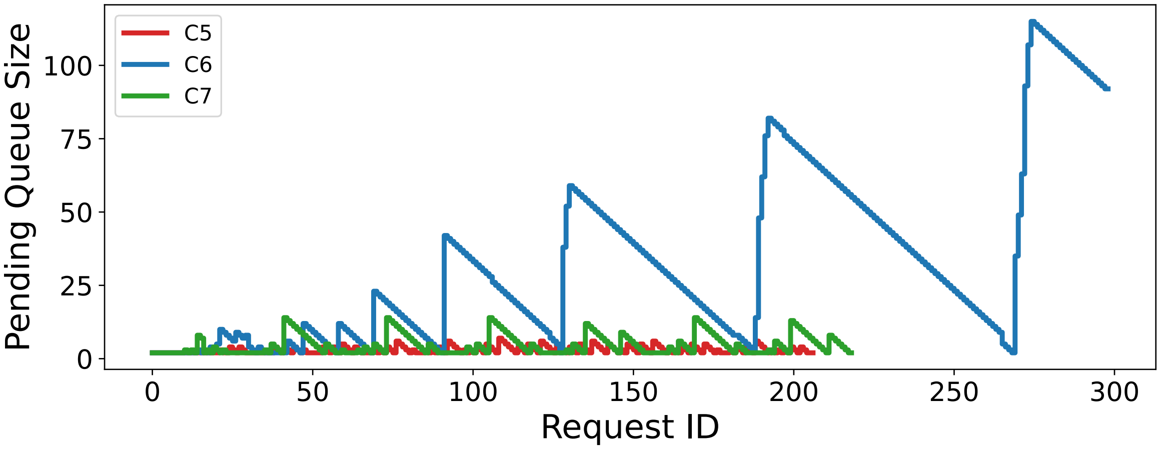


Figure 55. Pending RPC Queue Size: Immediate Mode (Shared)

Given this understanding of how to determine the optimal number of simulations per server instance in an *Immediate* setting, a subset of MxN configurations for each

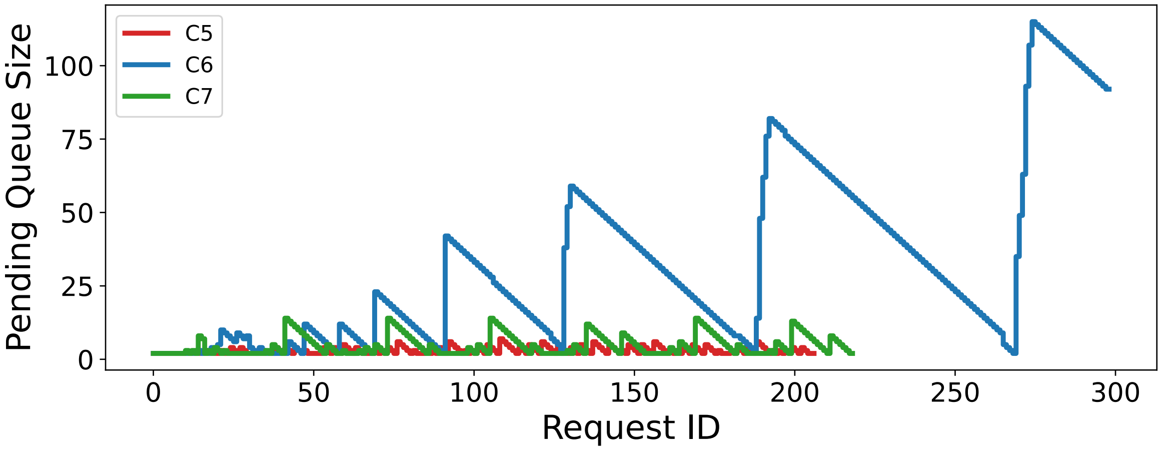


Figure 56. Simulation Execution Times: Immediate Mode (Shared)

of the three simulation process counts was chosen based on their model-projected cost savings (see Equation 6.7) when executed in a shared-server setting. Table 30 depicts the chosen simulation and server configurations for different MxN ratios. The RADICAL Pilot [236] (RP) ensemble system was used to emulate a shared-service setting wherein the simulations make concurrent requests to the server. Each simulation or server instance is executed as an independent RP task within a pilot job (batch job) allocation. A total of 17 nodes were allocated to $\mathbf{C_1}$, 33 nodes to $\mathbf{C_2}$, 49 nodes were allocated to $\mathbf{C_3}$, and 25 nodes to the $\mathbf{C_4}$ configuration. In each of these sub-allocations, the server tasks were launched first, allowing them to share their RPC addresses with simulations. Next, the simulations were launched simultaneously such that their execution timelines completely overlapped with each other. Configurations $\mathbf{C_1}$ to $\mathbf{C_4}$ were used as a "stress-test" for SERVIZ — visualization was invoked at every AMR-Wind iteration. For configurations $\mathbf{C_5}$ to $\mathbf{C_7}$, visualization was performed once out of every 8 AMR-Wind iterations to represent a more practical setting. At the same time, the number AMR-Wind simulations per server instance was increased 8-fold such that the total amount of visualization work in the system remained constant between the equivalent configurations of $\mathbf{C_1}$-$\mathbf{C_4}$ and $\mathbf{C_5}$-$\mathbf{C_7}$.

Table 30. Shared-Service Configurations
(**M**ode (immediate, delayed), Visualization **F**requency)

Config	MxN	#Clients, #Client Nodes	#Servers, Processes/Server	#Clients/Server	Size	M	F
$\mathbf{C_1}$	120x15	8, 16	4, 15	2	small	I	1
$\mathbf{C_2}$	240x30	8, 32	2, 30	4	large	I	1
$\mathbf{C_3}$	240x15	12, 48	4, 15	3	large	I	1
$\mathbf{C_4}$	360x60	4, 24	1, 60	4	large	I	1
$\mathbf{C_5}$	120x15	64, 128	4, 15	16	small	I	1/8
$\mathbf{C_6}$	240x30	64, 256	2, 30	32	large	I	1/8
$\mathbf{C_7}$	360x60	32, 192	1, 60	32	large	I	1/8
$\mathbf{C_8}$	120x15	64, 128	4, 15	16	small	D	1
$\mathbf{C_9}$	120x15	88, 176	4, 15	22	small	D	1
$\mathbf{C_{10}}$	360x60	30, 180	1, 60	30	large	D	1

Figure 56 depicts the execution times of simulations under the different MxN ratios. It is observed that the simulations in each configuration complete well within the baseline (inline) execution time. As Figure 57 demonstrates, for $\mathbf{C_1}$, SERVIZ achieves 9% of cost savings over the inline execution time, while for $\mathbf{C_4}$, SERVIZ is able to achieve up to 5% of cost savings. Simultaneously, for $\mathbf{C_2}$ and $\mathbf{C_4}$, SERVIZ can reduce the idle time on the server by four times compared to the dedicated in transit implementation. For the configurations tested, the maximum idle time on the server is 27% of the total server execution time for $\mathbf{C_7}$. As Figure 55 depicts, the pending RPC queue size for large-scale executions ($\mathbf{C_5}$ - $\mathbf{C_7}$) does not grow uncontrollably, indicating that the server configuration (single KNL node) is sufficient to handle the visualization workload.

6.5.3.2 Delayed Mode. The *Delayed* mode prioritizes responding to the visualization requests over processing them immediately. The visualization is performed later when no simulations are waiting for a data receipt to arrive from the server. In doing so, many more simulations can be coupled to a single server instance, compared to *Immediate* mode. Another benefit is that server processes need not synchronize, further increasing the capability of the server to take on more simulations.

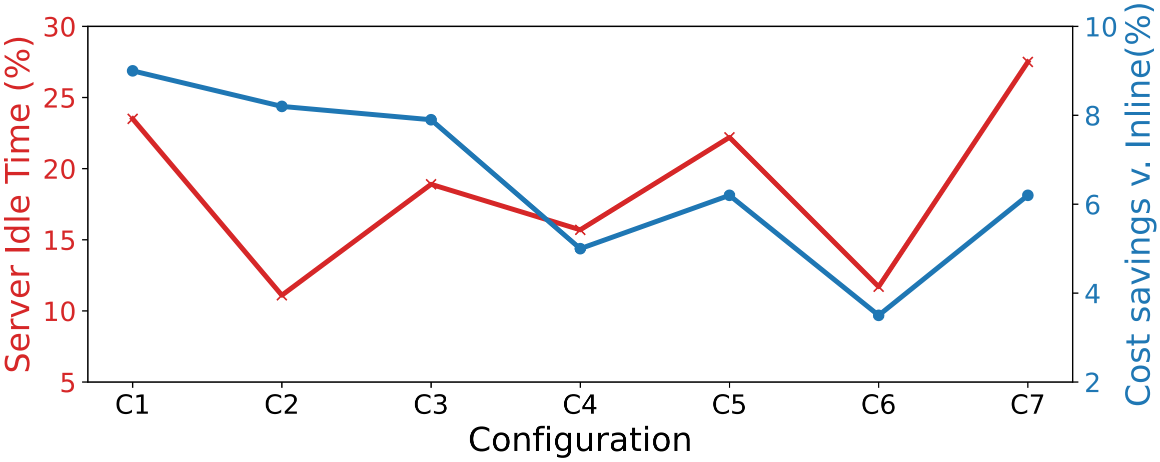


Figure 57. Cost Savings and Idle Time: Immediate Mode (Shared)

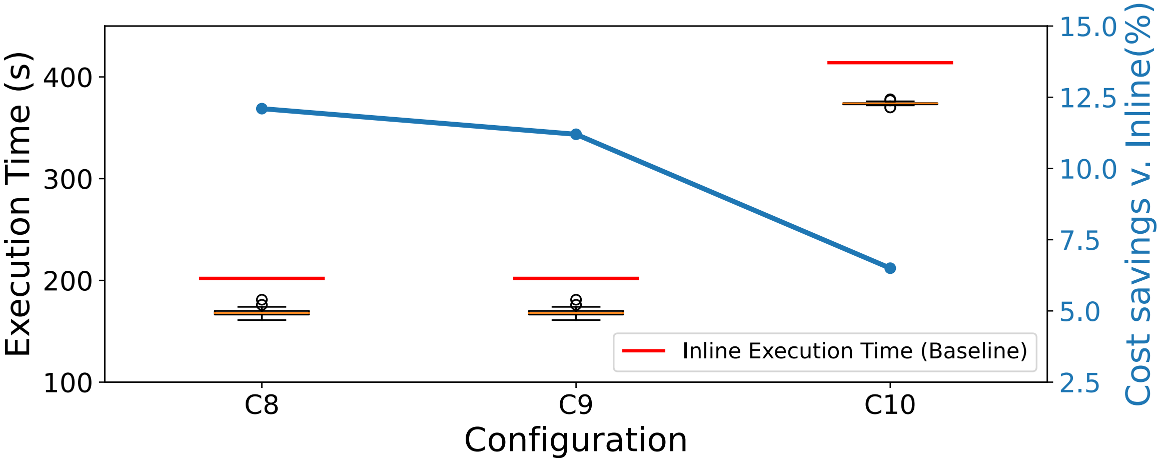


Figure 58. Simulation Time and Cost Savings: Delayed Mode (Shared)

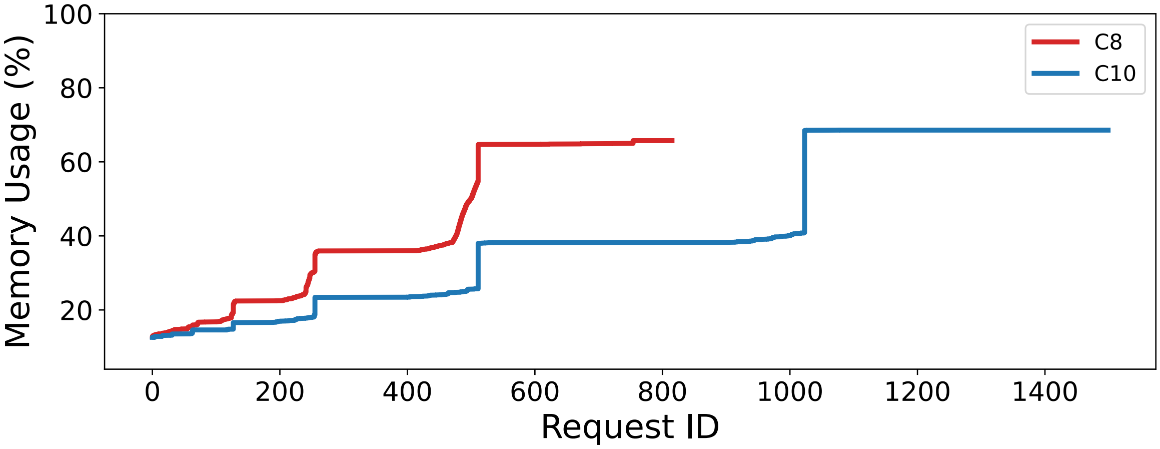


Figure 59. Memory Usage: Delayed Mode (Shared)

As Figure 51 demonstrates, the idle time on the server with a *Delayed* $\mathbf{C_1}$ configuration is more than 95% of the total server execution time (when the simulation is executing), suggesting that a single, 15-process server instance can comfortably handle 20 AMR-Wind simulations. However, when the server is actively responding to visualization requests, the memory usage in the *Delayed* mode grows with the number of visualization requests being stored for future execution.

Table 30 enlists the three shared-service configurations that were chosen for *Delayed* execution at a large scale. Similar to the *Immediate* configurations, these *Delayed* configurations were launched as a RADICAL Pilot ensemble job. The simulation task with the highest task ID was programmed within a given configuration to invoke the `serviz_execute_pending_requests` API after completing its time step iteration loop. At this point, the server immediately switched to executing the stored, pending visualization tasks until completion. The cost savings calculations were made by taking note of this time to execute the stored, pending requests. Batch systems do not allow partial freeing of compute nodes associated with completed tasks within a batch job. This constraint was relaxed while performing the cost savings calculations. Figure 58 depicts the cost savings achieved over the baseline inline implementation and the server idle time for configurations $\mathbf{C_8}$, $\mathbf{C_9}$, and $\mathbf{C_{10}}$.

The *Delayed* mode of operation offers up to 12.2%, 11.1%, and 6.2% cost savings over inline for $\mathbf{C_8}$, $\mathbf{C_9}$, and $\mathbf{C_{10}}$ respectively. The observations from Figure 59 suggest that the limiting factor for this mode of operation is node memory. Specifically, the memory usage steadily grows with the request ID (or simulation iteration count) until the execution of pending requests is triggered through `serviz_execute_pending_requests`.

Table 31. SERVIZ: Lines of Code

Component	Lines of Code
Data Gathering Logic	70
Service Discovery (simulation)	100
Service Invocation (simulation)	15
Service API Implementation	800
Service Template Code (Mochi framework)	3100

6.6 Discussion

As Section 6.4 elucidates, a service-based model is an attractive way to develop and deploy customized functionality in myriad ways. One element that was not emphasized in our experiments is the ease and speed with which it can be implemented. SERVIZ was assembled using off-the-shelf HPC software in a relatively short period, and minimal code changes were required to invoke and use the SERVIZ infrastructure. Table 31 depicts the total lines of code that were required to integrate SERVIZ into the AMR-WIND application. Notably, the simulation-side code changes were approximately 200 lines of code, of which a majority was dedicated to implementing the data gathering and the service discovery. Concerning the SERVIZ provider implementation, user-supplied lines of code were just 800, and the remaining 3100 lines of code were attributed to the off-the-shelf Mochi microservice programming template. The shared-service model also has additional cost advantages and functional advantages that should be explored as future work.

Concerning additional cost savings, we see two main opportunities with SERVIZ both from embracing a throughput model. Of note, most in situ visualization software to date has been designed with latency in mind. Those motivators are no longer present with the SERVIZ model, allowing it to focus on throughput instead of latency. The first is to increase VCEF by running at a smaller and smaller concurrency level in terms of the opportunities. For example, SERVIZ could experiment with running each successive visualization task from a simulation code on fewer and fewer cores to find the maximum VCEF factor. The second opportunity is to minimize idle time. SERVIZ needs sufficient to be sufficiently busy to achieve cost savings. “Work” should be evaluated relative to the computational resources, and one way to ensure this work exists is to use fewer resources. This could create a scenario where the backlog of visualization tasks becomes unacceptably large. However, the balance of tasks and resources can be perfectly “right-sized” with an adaptive monitoring component that can retain tasks on the client nodes when a backlog emerges. This could optimize our cost equations, i.e., maximizing VCEF and zeroing out idle.

On the functional side, SERVIZ shares some benefits with traditional in transit approaches and provides some new ones. Like traditional in transit, SERVIZ also has advantages over inline concerning fault tolerance and robustness. A functional advantage distinct to SERVIZ is the ability to do comparative analysis and visualization, whether across ensembles of related simulations or across time steps of a single simulation. SERVIZ would not be the first to do such an analysis, however, as Melissa [230] has been doing this for ensembles.

There are limitations with both our implementation and potentially with the approach overall. Concerning implementation, the SERVIZ development was constrained by restrictions in the HPC execution environment. Many platforms, Theta included, disallow the over-subscription of node-level resources. A node is limited to use by a single application, and the number of application processes is effectively capped to the number of available hardware cores. As a result, it was more difficult, if not impossible, to implement specific SERVIZ scenarios. For example, imagine a scenario where the simulation has fewer ranks than the total cores on each node. It would be interesting to *seamlessly* configure SERVIZ to run the visualization server on the simulation resources through a shared-memory coupling. By being co-located with the simulation, this mode of deployment can take advantage of the “free energy” on the compute node while minimizing the data transfer costs. Overall, concerning the approach, there is a practical question of how to charge resources. Most HPC centers provide a bank with an allocation of compute-node hours, and each job draws from that allocation. We envision SERVIZ living outside that model and accepting visualization tasks from disparate simulation codes. It is unclear how this time would be charged to the jobs. Ideally, if our research matures and proves practical, HPC centers would keep a permanent allocation running free of charge to make their entire center more efficient. Finally, even with this charging issue unresolved, our work is applicable for ensembles belonging to a single code team.

6.7 Summary

Chapter VI presented the SERVIZ shared in situ visualization service. We have demonstrated that a shared-service model offers significant cost savings over inline and dedicated in transit methods for performing visualization on large-scale clusters. By leveraging the Mochi microservice framework for development, SERVIZ was implemented and integrated quickly and with minimal code changes to the simulation code. Further, we demonstrated that SERVIZ could be configured in two different modes depending on how quickly the visualization results are desired.

CHAPTER VII

CONCLUSION AND FUTURE WORK

7.1 Conclusion

HPC software architectures have evolved over the past three decades to support an ever-increasing need to efficiently run simulations at a higher level of fidelity and concurrency and on more diverse hardware platforms. A notable outcome of this evolution is that HPC software has become increasingly modular. More recently, the emergence of ML workloads and data-centric computing paradigms has accelerated this trend, partly because integrating these new applications into traditional HPC workflows requires the support of a broader range of programming models than just MPI. The timing of the emergence of these new applications, conflated with the hardware trends that were already underway a decade ago, has resulted in the emergence of in situ workflows involving user-level HPC services. The onus rests on an end-user to identify how to optimally configure and deploy these services within these workflows. The lack of performance tools for these HPC services and the criticality of an optimal service configuration to the overall workflow performance motivates the search for a solution. This dissertation strives to answer the following main research question: **How to enable and use performance insight to improve service configuration when the service is a part of a coupled, HPC in situ workflow?** Chapter I provides an introduction to the challenges that need to be addressed to answer this central question and breaks down this broader question into its constituent elements. The following section summarizes the key takeaways from each chapter of this dissertation.

Chapter II explores the evolution of HPC software development over the past three decades to identify the key factors driving their design. This exploration identifies four broad factors contributing to the need to develop increasingly modular HPC software: (1) simulation scale and fidelity, (2) the need to support and integrate newer compute and storage hardware, (3) the need to support a broader range of applications requiring HPC, and (4) the structure and complexity of HPC software development teams. This chapter follows the parallel evolution of HPC performance tools with the goal of identifying the techniques that have been developed to observe and analyze modular software components and to determine the open areas requiring further study.

Chapter III narrows the focus of this dissertation to generating performance observability for HPC services that rely on a composition model to build and deploy custom, scalable distributed components. A background of the Mochi software framework [6] is discussed to identify the main performance-related queries that contribute to addressing the broader question of generating an improved service configuration.

Chapter IV presents the tool solutions to address each of the performance-related queries raises in Chapter III. The tool solutions, packaged as SYMBIOSYS and SYMBIOMON, comprise various techniques borrowed from the HPC community and the broader cloud community. The challenge lies in understanding how to adapt them to be applicable for performance observability and monitoring of HPC microservices. Distributed callpath profiling and tracing generate a picture of the dominant high-level microservice operations. These callpath profiles, when embellished with rich performance data from the RPC layer, make visible the occurrence of low-level events and account for their contribution to the total RPC execution time. Sampling is employed to provide a hardware-centric view of the service execution. We find statistical analysis of time-series metrics useful to identify a better service configuration when the cause of the performance inefficiency lies within the microservice API.

Chapter V addresses the challenge of enabling ubiquitous performance monitoring of a variety of workflow components, including MPI applications and ensembles. A plugin architecture for the TAU performance system enables the seamless integration of SYMBIOMON's monitoring capabilities into MPI applications and ensembles, while simultaneously leveraging TAU's existing measurement capabilities.

Chapter VI demonstrates the broader utility of the core Mochi components to develop and deploy high performance functionality as microservices. This capability is demonstrated through SERVIZ, a shared in situ visualization service. SERVIZ is a *hybrid* MPI + RPC distributed application capable of realizing more cost savings than traditional in transit visualization approaches. Through SERVIZ, we demonstrate that such shared services can be rapidly developed and deployed with MPI applications while requiring a minimal set of code changes.

7.2 Future Work

The research presented in this dissertation opens up broad avenues for future work with high performance microservices. Some of these future directions are discussed here.

7.2.1 Enabling Online Service Adaptivity. While the main focus of this dissertation is to enable the performance observability and monitoring of HPC microservices, the automatic adaptation of the service in response to changes in the client workload represents the ultimate goal for this performance infrastructure. Enabling online service adaptivity is a complex undertaking involving the navigation of a large parameter search space and a separate set of challenges concerning the actuation of these parameter changes. Our initial experiments suggest combining state-of-the-art ML-based search strategies (such as those employed by DeepHyper [237]) with the first-principles approach to performance observation analysis represented by the SYMBIOSYS and SYMBIOMON tools can potentially accelerate the process of narrowing down this large search space.

7.2.2 Elastic In Situ Visualization. The research presented in Chapter VI suggests that a shared visualization service can yield significant cost savings, thereby improving the overall efficiency of the HPC cluster. However, if we can demonstrate their elastic operation and their ability to respond appropriately to the changing mix of applications running on the platform, this would strengthen the case of exposing these shared services to user applications. Doing so requires a monitoring solution combined with the shared visualization service. Composing the existing SYMBIOMON monitoring system with the SERVIZ visualization service is an interesting first step to be explored as a solution.

7.2.3 User-level Shared Monitoring + Learning Service. The idea of a shared user-level service to improve the overall efficiency of the HPC cluster also finds a potential application in performance monitoring and analysis. Specifically, a delayed, throughput-oriented performance monitoring and analysis model can be implemented using shared services running on cluster-owned node resources. The applications (jobs) running on the cluster serve as the source for the performance data, while the performance monitoring infrastructure hosts the storage, analysis, and learning components. Further, these storage and analysis services can be partitioned to represent different usage modes: (1) one partition to serve the immediate adaptivity needs of applications currently running in the system, (2) one partition to enable remote monitoring of performance data, and (3) another partition to perform "long-term" performance analysis and learning to yield knowledge about the mix of jobs and how they use the cluster resources. This research would open up interesting questions about implementing quality of service for these different operating modes.

REFERENCES CITED

[1] F. Bertrand and R. Bramley, “Dca: A distributed cca framework based on mpi,” in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 80–89.

[2] “Openzipkin,” http://zipkin.io.

[3] T. Ben-Nun, T. Gamblin, D. Hollman, H. Krishnan, and C. 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).* IEEE, 2020, pp. 57–69.

[4] C. M. MacKenzie, K. Laskey, F. McCabe, P. F. Brown, R. Metz, and B. A. Hamilton, “Reference model for service oriented architecture 1.0,” *OASIS standard*, vol. 12, no. S 18, 2006.

[5] J. F. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin, “Flexible io and integration for scientific codes through the adaptable io system (adios),” in *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, 2008, pp. 15–24.

[6] R. B. Ross, G. Amvrosiadis, P. Carns, C. D. Cranor, M. Dorier, K. Harms, G. Ganger, G. Gibson, S. K. Gutierrez, R. Latham *et al.*, “Mochi: Composing data services for high-performance computing environments,” *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 121–144, 2020.

[7] C. Ulmer, S. Mukherjee, G. Templet, S. Levy, J. Lofstead, P. Widener, T. Kordenbrock, and M. Lawson, “Faodel: Data management for next-generation application workflows,” in *Proceedings of the 9th Workshop on Scientific Cloud Computing*, 2018, pp. 1–6.

[8] M. Romanus, F. Zhang, T. Jin, Q. Sun, H. Bui, M. Parashar, J. Choi, S. Janhunen, R. Hager, S. Klasky *et al.*, “Persistent data staging services for data intensive in-situ scientific workflows,” in *Proceedings of the ACM International Workshop on Data-Intensive Distributed Computing*, 2016, pp. 37–44.

[9] M. Dreher and T. Peterka, “Decaf: Decoupled dataflows for in situ high-performance workflows,” Argonne National Lab.(ANL), Argonne, IL (United States), Tech. Rep., 2017.

[10] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. Epperly, M. Govindaraju *et al.*, “A component architecture for high-performance scientific computing,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[11] Y. Alexeev *et al.*, “Component-based software for high-performance scientific computing,” *Journal of Physics: Conference Series*, vol. 16, no. 1, 2005.

[12] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, “Toward a common component architecture for high-performance scientific computing,” in *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469).* IEEE, 1999, pp. 115–124.

[13] S. Ramesh, A. D. Malony, P. Carns, R. B. Ross, M. Dorier, J. Soumagne, and S. Snyder, “Symbiosys: A methodology for performance analysis of composable hpc data services,” in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2021, pp. 35–45.

[14] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver *et al.*, “Dapper, a large-scale distributed systems tracing infrastructure,” 2010.

[15] M. P. Forum, “Mpi: A message-passing interface standard,” 1994.

[16] S. Ramesh, R. Ross, M. Dorier, A. Malony, P. Carns, and K. Huck, “Symbiomon: A high-performance, composable monitoring service,” in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC).* IEEE, 2021, pp. 332–342.

[17] ——, “Symbio: Enabling observability and adaptivity of high performance data services (poster, srs@hipc),” 2021.

[18] ——, “Symbiomon: A high performance, composable monitoring service for online application introspection and adaptation (poster, src@sc),” 2021.

[19] “Prometheus,” https://prometheus.io/.

[20] A. Malony, S. Ramesh, K. Huck, N. Chaimov, and S. Shende, “A plugin architecture for the TAU performance system,” in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[21] A. D. Malony, S. Ramesh, K. Huck, N. Chaimov, and S. Shende, “A plugin architecture for the tau performance system,” in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[22] S. Ramesh, H. Childs, and A. D. Malony, "Serviz: A shared in situ visualization service (submitted, sc)," 2022.

[23] J. M. Wozniak, R. Jain, P. Balaprakash, J. Ozik, N. T. Collier, J. Bauer, F. Xia, T. Brettin, R. Stevens, J. Mohd-Yusof *et al.*, "Candle/supervisor: A workflow framework for machine learning applied to cancer research," *BMC bioinformatics*, vol. 19, no. 18, pp. 59–69, 2018.

[24] P. M. Kasson and S. Jha, "Adaptive ensemble simulations of biomolecules," *Current opinion in structural biology*, vol. 52, pp. 87–94, 2018.

[25] A. Malony, "Performance understanding and analysis for exascale data management workflows," Univ. of Oregon, Eugene, OR (United States), Tech. Rep., 2019.

[26] B. Norris, J. Ray, R. Armstrong, L. C. McInnes, D. E. Bernholdt, W. R. Elwasif, A. D. Malony, and S. Shende, "Computational quality of service for scientific components," in *International Symposium on Component-Based Software Engineering.* Springer, 2004, pp. 264–271.

[27] W. Emmerich and N. Kaveh, "Component technologies: Java beans, com, corba, rmi, ejb and the corba component model," in *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 2001, pp. 311–312.

[28] A. Malony, S. Shende, N. Trebon, J. Ray, R. Armstrong, C. Rasmussen, and M. Sottile, "Performance technology for parallel and distributed component software," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 117–141, 2005.

[29] S. Shende and A. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[30] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey *et al.*, "Hpctoolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[31] D. Boehme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre *et al.*, "Caliper: performance introspection for hpc software stacks," in *SC.* IEEE Press, 2016, p. 47.

[32] "Price of hpc systems," https://qz.com/1301510.

[33] "Top500 list," https://www.top500.org/.

[34] G. F. Pfister, "An introduction to the infiniband architecture," *High performance mass storage and parallel I/O*, vol. 42, no. 617-632, p. 102, 2001.

[35] D. W. Walker and J. J. Dongarra, "Mpi: a standard message passing interface," *Supercomputer*, vol. 12, pp. 56–68, 1996.

[36] "Ecp proxy suite," https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite.

[37] D. Bailey, T. Harris, W. Saphir, R. Van Der Wijngaart, A. Woo, and M. Yarrow, "The nas parallel benchmarks 2.0," Technical Report NAS-95-020, NASA Ames Research Center, Tech. Rep., 1995.

[38] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[39] J. Liu, J. Wu, and D. K. Panda, "High performance rdma-based mpi implementation over infiniband," *International Journal of Parallel Programming*, vol. 32, no. 3, pp. 167–198, 2004.

[40] G. Chrysos, "Intel® xeon phi™ coprocessor-the architecture," *Intel Whitepaper*, vol. 176, p. 43, 2014.

[41] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE computational science and engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[42] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[43] B. Nichols, D. Buttlar, and J. P. Farrell, *Pthreads programming.* O'Reilly & Associates, Inc., 1996.

[44] S. Wienke, P. Springer, C. Terboven, and D. an Mey, "Openacc—first experiences with real-world applications," in *European Conference on Parallel Processing.* Springer, 2012, pp. 859–870.

[45] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[46] A. Munshi, "The opencl specification," in *2009 IEEE Hot Chips 21 Symposium (HCS).* IEEE, 2009, pp. 1–314.

[47] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[48] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to upc and language specification," Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[49] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[50] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "Papi software-defined events for in-depth performance analysis," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1113–1127, 2019.

[51] A. E. Eichenberger, J. Mellor-Crummey, M. Schulz, M. Wong, N. Copty, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "Ompt: An openmp tools application programming interface for performance analysis," in *International Workshop on OpenMP*. Springer, 2013, pp. 171–185.

[52] J. E. Kay, C. Deser, A. Phillips, A. Mai, C. Hannay, G. Strand, J. M. Arblaster, S. Bates, G. Danabasoglu, J. Edwards *et al.*, "The community earth system model (cesm) large ensemble project: A community resource for studying climate change in the presence of internal climate variability," *Bulletin of the American Meteorological Society*, vol. 96, no. 8, pp. 1333–1349, 2015.

[53] S. Plimpton, P. Crozier, and A. Thompson, "Lammps-large-scale atomic/molecular massively parallel simulator," *Sandia National Laboratories*, vol. 18, p. 43, 2007.

[54] S. Habib, A. Pope, H. Finkel, N. Frontiere, K. Heitmann, D. Daniel, P. Fasel, V. Morozov, G. Zagaris, T. Peterka *et al.*, "Hacc: Simulating sky surveys on state-of-the-art supercomputing architectures," *New Astronomy*, vol. 42, pp. 49–65, 2016.

[55] "Coral-1 benchmarks," https://asc.llnl.gov/coral-benchmarks.

[56] "Coral-2 benchmarks," https://asc.llnl.gov/coral-2-benchmarks.

[57] J. Y. Choi, C.-S. Chang, J. Dominski, S. Klasky, G. Merlo, E. Suchyta, M. Ainsworth, B. Allen, F. Cappello, M. Churchill *et al.*, "Coupling exascale multiphysics applications: Methods and lessons learned," in *2018 IEEE 14th International Conference on e-Science (e-Science)*. IEEE, 2018, pp. 442–452.

[58] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," *Present and ulterior software engineering*, pp. 195–216, 2017.

[59] C. Szyperski, D. Gruntz, and S. Murer, *Component software: beyond object-oriented programming.* Pearson Education, 2002.

[60] A. L. Pope, *The CORBA reference guide: understanding the common object request broker architecture.* Addison-Wesley Longman Publishing Co., Inc., 1998.

[61] R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects.* John Wiley & Sons, Inc., 1997.

[62] D. E. Bernholdt, B. A. Allan, R. Armstrong, F. Bertrand, K. Chiu, T. L. Dahlgren, K. Damevski, W. R. Elwasif, T. G. Epperly, M. Govindaraju *et al.*, "A component architecture for high-performance scientific computing," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 163–202, 2006.

[63] Y. Alexeev *et al.*, "Component-based software for high-performance scientific computing," *Journal of Physics: Conference Series*, vol. 16, no. 1, 2005.

[64] S. G. Parker, "A component-based architecture for parallel multi-physics pde simulation," *Future Generation Computer Systems*, vol. 22, no. 1-2, pp. 204–216, 2006.

[65] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *Proceedings. The Eighth International Symposium on High Performance Distributed Computing (Cat. No. 99TH8469).* IEEE, 1999, pp. 115–124.

[66] R. Armstrong, G. Kumfert, L. C. McInnes, S. Parker, B. Allan, M. Sottile, T. Epperly, and T. Dahlgren, "The cca component model for high-performance scientific computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 2, pp. 215–229, 2006.

[67] T. G. Epperly, G. Kumfert, T. Dahlgren, D. Ebner, J. Leek, A. Prantl, and S. Kohn, "High-performance language interoperability for scientific computing through babel," *The International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 260–274, 2012.

[68] J. Ray, N. Trebon, R. C. Armstrong, S. Shende, and A. Malony, "Performance measurement and modeling of component applications in a high performance computing environment: A case study," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* IEEE, 2004, p. 95.

[69] N. Trebon, A. Morris, J. Ray, S. Shende, and A. Malony, "Performance modeling of component assemblies with tau," in *Compframe 2005 workshop, Atlanta*, 2005.

[70] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl, "The cca core specification in a distributed memory spmd framework," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 5, pp. 323–345, 2002.

[71] A. Cleary, S. Kohn, S. G. Smith, and B. Smolinski, "Language interoperability mechanisms for high-performance scientific applications," in *Proceedings of the 1998 SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing*, vol. 99, 1999, pp. 30–39.

[72] N. Trebon, A. Morris, J. Ray, S. Shende, and A. D. Malony, "Performance modeling of component assemblies," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 5, pp. 685–696, 2007.

[73] P. Hovland, K. Keahey, L. McInnes, B. Norris, L. Diachin, and P. Raghavan, "A quality of service approach for high-performance numerical components," in *Proceedings of Workshop on QoS in Component-Based Software Engineering, Software Technologies Conference, Toulouse, France*, vol. 20, 2003.

[74] L. C. McInnes, J. Ray, R. Armstrong, T. L. Dahlgren, A. Malony, B. Norris, S. Shende, J. P. Kenny, and J. Steensland, "Computational quality of service for scientific cca applications: Composition, substitution, and reconfiguration," *Preprint ANL/MCS-P1326-0206, Argonne National Laboratory, Feb*, 2006.

[75] V. Bui, B. Norris, K. Huck, L. C. McInnes, L. Li, O. Hernandez, and B. Chapman, "A component infrastructure for performance and power modeling of parallel scientific applications," in *Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, 2008, pp. 1–11.

[76] N. Furmento, A. Mayer, S. McGough, S. Newhouse, T. Field, and J. Darlington, "Optimisation of component-based applications within a grid environment," in *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, 2001, pp. 30–30.

[77] J. Bigot, Z. Hou, C. Pérez, and V. Pichon, "A low level component model easing performance portability of hpc applications," *Computing*, vol. 96, no. 12, pp. 1115–1130, 2014.

[78] C. Perez and V. Lanore, "Towards reconfigurable hpc component models," in *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2018, pp. 151–152.

[79] M. Malawski, D. Kurzyniec, and V. Sunderam, "Mocca-towards a distributed cca framework for metacomputing," in *19th IEEE International Parallel and Distributed Processing Symposium.* IEEE, 2005, pp. 8–pp.

[80] R. Schmidt, S. Benkner, and M. Lucka, "A component plugin mechanism and framework for application web services," in *Towards next generation grids: proceedings of the CoreGRID Symposium 2007, August 27-28, Rennes, France.* Springer, 2007, p. 107.

[81] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri, "A component based services architecture for building distributed applications," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing.* IEEE, 2000, pp. 51–59.

[82] M. Govindaraju, M. J. Lewis, and K. Chiu, "Design and implementation issues for distributed cca framework interoperability," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 5, pp. 651–666, 2007.

[83] S. Krishnan and D. Gannon, "Xcat3: A framework for cca components as ogsa services," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 90–97.

[84] K. Zhang, K. Damevski, V. Venkatachalapathy, and S. G. Parker, "Scirun2: A cca framework for high performance computing," in *Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments, 2004. Proceedings.* IEEE, 2004, pp. 72–79.

[85] J. D. d. S. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, "Uintah: A massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing.* IEEE, 2000, pp. 33–41.

[86] J. V. Reynders Iii, J. Cummings, and P. F. Dubois, "The pooma framework," *Computers in Physics*, vol. 12, no. 5, pp. 453–459, 1998.

[87] S. Balay, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, B. F. Smith, and H. Zhang, "Petsc," *See http://www. mcs. anl. gov/petsc*, 2001.

[88] R. D. Falgout and U. M. Yang, "hypre: A library of high performance preconditioners," in *International Conference on Computational Science.* Springer, 2002, pp. 632–641.

[89] M. Parashar and J. C. Browne, “Systems engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh,” in *Structured adaptive mesh refinement (SAMR) grid methods*. Springer, 2000, pp. 1–18.

[90] D. L. Brown, G. S. Chesshire, W. D. Henshaw, and D. J. Quinlan, “Overture: An object-oriented software system for solving partial differential equations in serial and parallel environments,” Los Alamos National Lab., NM (United States), Tech. Rep., 1997.

[91] J. M. Squyres and A. Lumsdaine, “A component architecture for lam/mpi,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2003, pp. 379–387.

[92] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, “Open mpi: Goals, concept, and design of a next generation mpi implementation,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2004, pp. 97–104.

[93] R. Keller, G. Bosilca, G. Fagg, M. Resch, and J. J. Dongarra, “Implementation and usage of the peruse-interface in open mpi,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*. Springer, 2006, pp. 347–355.

[94] T. Islam, K. Mohror, and M. Schulz, “Exploring the capabilities of the new mpi_t interface,” in *Proceedings of the 21st European MPI Users’ Group Meeting*, 2014, pp. 91–96.

[95] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. D. Panda, “Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau,” *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[96] E. Gallardo, J. Vienne, L. Fialho, P. Teller, and J. Browne, “Employing mpi_t in mpi advisor to optimize application performance,” *The International Journal of High Performance Computing Applications*, vol. 32, no. 6, pp. 882–896, 2018.

[97] M.-A. Hermanns, N. T. Hjelm, M. Knobloch, K. Mohror, and M. Schulz, “The mpi_t events interface: An early evaluation and overview of the interface,” *Parallel computing*, vol. 85, pp. 119–130, 2019.

[98] B. Mohr, A. D. Malony, S. Shende, and F. Wolf, “Design and prototype of a performance tool interface for openmp,” *The Journal of Supercomputing*, vol. 23, no. 1, pp. 105–128, 2002.

[99] M. Itzkowitz and Y. Maruyama, “Hpc profiling with the sun studio™ performance tools,” in *Tools for high performance computing 2009*. Springer, 2010, pp. 67–93.

[100] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, “Papi software-defined events for in-depth performance analysis,” *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1113–1127, 2019.

[101] J. Logan, M. Ainsworth, C. Atkins, J. Chen, J. Y. Choi, J. Gu, J. M. Kress, G. Eisenhauer, B. Geveci, W. Godoy *et al.*, “Extending the publish/subscribe abstraction for high-performance i/o and data management at extreme scale,” *Bulletin of the IEEE Technical Committee on Data Engineering*, vol. 43, no. 1, 2020.

[102] G. Merlo, S. Janhunen, F. Jenko, A. Bhattacharjee, C. Chang, J. Cheng, P. Davis, J. Dominski, K. Germaschewski, R. Hager *et al.*, “First coupled gene–xgc microturbulence simulations,” *Physics of Plasmas*, vol. 28, no. 1, p. 012303, 2021.

[103] V. Sarkar, W. Harrod, and A. E. Snavely, “Software challenges in extreme scale systems,” in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012045.

[104] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, “The future of scientific workflows,” *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159–175, 2018.

[105] D. A. Chappell, *Enterprise service bus.* ” O’Reilly Media, Inc.”, 2004.

[106] P. H. Beckman, P. K. Fasel, W. E. Humphrey, and S. M. Mniszewski, “Efficient coupling of parallel applications using paws,” in *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No. 98TB100244).* IEEE, 1998, pp. 215–222.

[107] D. Bernholdt, F. Bertrand, R. Bramley, K. Damevski, J. Kohl, S. Parker, and A. Sussman, ““mxn” parallel data redistribution research in the common component architecture (cca).”

[108] I. B. Peng, R. Gioiosa, G. Kestor, E. Laure, and S. Markidis, “Preparing hpc applications for the exascale era: A decoupling strategy,” in *2017 46th International Conference on Parallel Processing (ICPP).* IEEE, 2017, pp. 1–10.

[109] R. Latham, R. Ross, and R. Thakur, “Can mpi be used for persistent parallel services?” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting.* Springer, 2006, pp. 275–284.

[110] J. A. Zounmevo, D. Kimpe, R. Ross, and A. Afsahi, "Using mpi in high-performance computing services," in *Proceedings of the 20th European MPI Users' Group Meeting*, 2013, pp. 43–48.

[111] D. Zhao, Z. Zhang, X. Zhou, T. Li, K. Wang, D. Kimpe, P. Carns, R. Ross, and I. Raicu, "Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems," in *2014 IEEE international conference on big data (Big Data)*. IEEE, 2014, pp. 61–70.

[112] M.-A. Vef, N. Moti, T. Süß, T. Tocci, R. Nou, A. Miranda, T. Cortes *et al.*, "Gekkofs-a temporary distributed file system for hpc applications," in *CLUSTER*. IEEE, 2018, pp. 319–324.

[113] A. Moody, D. Sikich, N. Bass, M. J. Brim, C. Stanavige, H. Sim, J. Moore, T. Hutter, S. Boehm, K. Mohror *et al.*, "Unifyfs: A distributed burst buffer file system-0.1. 0," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2017.

[114] A. Anwar, Y. Cheng, H. Huang, J. Han, H. Sim, D. Lee, F. Douglis, and A. R. Butt, "Bespokv: Application tailored scale-out key-value stores," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 14–29.

[115] M. A. Sevilla, N. Watkins, I. Jimenez, P. Alvaro, S. Finkelstein, J. LeFevre, and C. Maltzahn, "Malacology: A programmable storage system," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 175–190.

[116] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[117] Martin Fowler, "Microservices," https://martinfowler.com/articles/microservices.html.

[118] O. Zimmermann, "Microservices tenets," *Computer Science-Research and Development*, vol. 32, no. 3, pp. 301–310, 2017.

[119] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis, and S. Tilkov, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[120] J. Soumagne, D. Kimpe, J. A. Zounmevo, M. Chaarawi, Q. Koziol, A. Afsahi, and R. B. Ross, "Mercury: Enabling remote procedure call for high-performance computing." in *CLUSTER*, 2013, pp. 1–8.

[121] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault *et al.*, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, 2017.

[122] M. Dorier, P. Carns, K. Harms, R. Latham, R. Ross, S. Snyder, J. Wozniak, S. K. Gutiérrez, B. Robey, B. Settlemyer *et al.*, "Methodology for the rapid development of scalable hpc data services," in *2018 IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS).* IEEE, 2018, pp. 76–87.

[123] H. Childs, S. D. Ahern, J. Ahrens, A. C. Bauer, J. Bennett, E. W. Bethel, P.-T. Bremer, E. Brugger, J. Cottam, M. Dorier *et al.*, "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[124] A. C. Bauer, H. Abbasi, J. Ahrens, H. Childs, B. Geveci, S. Klasky, K. Moreland, P. O'Leary, V. Vishwanath, B. Whitlock *et al.*, "In situ methods, infrastructures, and applications on high performance computing platforms," in *Computer Graphics Forum*, vol. 35, no. 3. Wiley Online Library, 2016, pp. 577–597.

[125] P. Grosset, J. Pulido, and J. Ahrens, "Personalized in situ steering for analysis and visualization," in *ISAV'20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2020, pp. 1–6.

[126] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2015, pp. 25–29.

[127] H. Childs *et al.*, "VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data," in *High Performance Visualization—Enabling Extreme-Scale Scientific Insight.* CRC Press/Francis–Taylor Group, Oct. 2012, pp. 357–372.

[128] U. Ayachit, B. Whitlock, M. Wolf, B. Loring, B. Geveci, D. Lonie, and E. W. Bethel, "The sensei generic in situ interface," in *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV).* IEEE, 2016, pp. 40–44.

[129] E. Dirand, L. Colombet, and B. Raffin, "Tins: A task-based dynamic helper core strategy for in situ analytics," in *Asian Conference on Supercomputing Frontiers.* Springer, 2018, pp. 159–178.

[130] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The alpine in situ infrastructure: Ascending from the ashes of strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, 2017, pp. 42–46.

[131] D. Morozov and Z. Lukic, "Master of puppets: Cooperative multitasking for in situ processing," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 285–288.

[132] M. Dorier, R. Sisneros, T. Peterka, G. Antoniu, and D. Semeraro, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 67–75.

[133] ——, "Damaris/viz: a nonintrusive, adaptable and user-friendly in situ visualization framework," in *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*. IEEE, 2013, pp. 67–75.

[134] S. Brandon, D. Domyancic, J. Tannahill, D. Lucas, G. Christianson, J. McEnereny, and R. Klein, "Ensemble calculation via the llnl uq pipeline: A user's guide," *Tech. Rep. LLNL-SM-480999*, 2011.

[135] D. H. Ahn, N. Bass, A. Chu, J. Garlick, M. Grondona, S. Herbein, H. I. Ingólfsson, J. Koning, T. Patki, T. R. Scogland *et al.*, "Flux: Overcoming scheduling challenges for exascale workflows," *Future Generation Computer Systems*, vol. 110, pp. 202–213, 2020.

[136] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster, "Swift/t: Large-scale application composition via distributed-memory dataflow processing," in *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*. IEEE, 2013, pp. 95–102.

[137] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications," in *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, 2012, pp. 1–12.

[138] J. L. Peterson, R. Anirudh, K. Athey, B. Bay, P.-T. Bremer, V. Castillo, F. Di Natale, D. Fox, J. A. Gaffney, D. Hysom *et al.*, "Merlin: Enabling machine learning-ready hpc ensembles," *arXiv preprint arXiv:1912.02892*, 2019.

[139] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers," *CoRR, abs/1512.08194*, 2015.

[140] LLNL, “Conduit,” https://llnl-conduit.readthedocs.io/en/latest/.

[141] P. Hintjens, *ZeroMQ: messaging for many applications.* ” O’Reilly Media, Inc.”, 2013.

[142]

[143] M. Dorier, O. Yildiz, T. Peterka, and R. Ross, “The challenges of elastic in situ analysis and visualization,” in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, 2019, pp. 23–28.

[144] A. Raveendran, T. Bicer, and G. Agrawal, “A framework for elastic execution of existing mpi programs,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum.* IEEE, 2011, pp. 940–947.

[145] C. Huang, O. Lawlor, and L. V. Kale, “Adaptive mpi,” in *International workshop on languages and compilers for parallel computing.* Springer, 2003, pp. 306–322.

[146] D. Rajan, A. Canino, J. A. Izaguirre, and D. Thain, “Converting a high performance application to an elastic cloud application,” in *2011 IEEE Third International Conference on Cloud Computing Technology and Science.* IEEE, 2011, pp. 383–390.

[147] L. Zhao and S. A. Jarvis, “Predictive performance modelling of parallel component compositions,” *Cluster Computing*, vol. 10, no. 2, pp. 155–166, 2007.

[148] L. A. Drummond, J. Demmel, C. R. Mechoso, H. Robinson, K. Sklower, and J. A. Spahr, “A data broker for distributed computing environments,” in *International Conference on Computational Science.* Springer, 2001, pp. 31–40.

[149] J. A. Kohl, T. Wilde, and D. E. Bernholdt, “Cumulvs: Interacting with high-performance scientific simulations, for visualization, steering and fault tolerance,” *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 255–285, 2006.

[150] L. Zhang, C. Docan, and M. Parashar, “The seine data coupling framework for parallel scientific applications,” *Advanced Computational Infrastructures for Parallel and Distributed Adaptive Applications*, vol. 66, p. 283, 2010.

[151] J. Larson, R. Jacob, and E. Ong, “The model coupling toolkit: a new fortran90 toolkit for building multiphysics parallel coupled models,” *The International Journal of High Performance Computing Applications*, vol. 19, no. 3, pp. 277–292, 2005.

[152] J.-Y. Lee and A. Sussman, "High performance communication between parallel programs," in *19th IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2005, pp. 8–pp.

[153] J. Dayal, D. Bratcher, G. Eisenhauer, K. Schwan, M. Wolf, X. Zhang, H. Abbasi, S. Klasky, and N. Podhorszki, "Flexpath: Type-based publish/subscribe system for large-scale science analytics," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2014, pp. 246–255.

[154] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *2011 IEEE Symposium on Large Data Analysis and Visualization*. IEEE, 2011, pp. 9–14.

[155] G. Eisenhauer, M. Wolf, H. Abbasi, and K. Schwan, "Event-based systems: Opportunities and challenges at exascale," in *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, 2009, pp. 1–10.

[156] J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert, "Flowvr: a middleware for large scale virtual reality applications," in *European Conference on Parallel Processing*. Springer, 2004, pp. 497–505.

[157] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer *et al.*, "Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir," in *Tools for HPC*. Springer, 2012, pp. 79–91.

[158] M. Wolf, J. Choi, G. Eisenhauer, S. Ethier, K. Huck, S. Klasky, J. Logan, A. Malony, C. Wood, J. Dominski *et al.*, "Scalable performance awareness for in situ scientific applications," in *2019 15th International Conference on eScience (eScience)*. IEEE, 2019, pp. 266–276.

[159] A. D. Malony, M. Larsen, K. A. Huck, C. Wood, S. Sane, and H. Childs, "When parallel performance measurement and analysis meets in situ analytics and visualization." in *PARCO*, 2019, pp. 521–530.

[160] A. D. Malony, S. Ramesh, K. Huck, N. Chaimov, and S. Shende, "A plugin architecture for the tau performance system," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[161] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system," *Key design insights from years of practical experience. Parallel Data Lab*, 2014.

[162] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger, "Diagnosing performance changes by comparing request flows." in *NSDI*, vol. 5, 2011, pp. 1–1.

[163] "Jaeger tracing," https://www.jaegertracing.io.

[164] X. Zhang, H. Abbasi, K. Huck, and A. Malony, "Wowmon: A machine learning-based profiler for self-adaptive instrumentation of scientific workflows," *Procedia Computer Science*, vol. 80, pp. 1507–1518, 2016.

[165] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony, "A scalable observation system for introspection and in situ analytics," in *2016 5th workshop on extreme-Scale Programming Tools (ESPT).* IEEE, 2016, pp. 42–49.

[166] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden *et al.*, "The lightweight distributed metric service: a scalable infrastructure for continuous monitoring of large scale computing systems and applications," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014, pp. 154–165.

[167] P. Roth, D. Arnold, and B. Miller, "MRNet: A software-based multicast/reduction network for scalable tools," in *SC'03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, 2003, pp. 21–21.

[168] M. Massie, B. Chun, and D. Culler, "The Ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, 2004.

[169] C. Kelly, S. Ha, K. Huck, H. Van Dam, L. Pouchard, G. Matyasfalvi, L. Tang, N. D'Imperio, W. Xu, S. 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*, 2020, pp. 15–19.

[170] "Graphite," https://graphiteapp.org/.

[171] K. Huck, S. Shende, A. Malony, H. Kaiser, A. Porterfield, R. Fowler *et al.*, "An early prototype of an autonomic performance environment for exascale," in *Proceedings of the 3rd Int. Workshop on Runtime and Operating Systems for Supercomputers*, 2013, pp. 1–8.

[172] 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 *Proceedings Frontiers' 95. The Fifth Symposium on the Frontiers of Massively Parallel Computation.* IEEE, 1995, pp. 422–429.

[173] N. Cheriere, M. Dorier, G. Antoniu, S. M. Wild, S. Leyffer, and R. Ross, "Pufferscale: Rescaling hpc data services for high energy physics applications," in *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID).* IEEE, 2020, pp. 182–191.

[174] P. M. Kasson and S. Jha, "Adaptive ensemble simulations of biomolecules," *Current opinion in structural biology*, vol. 52, pp. 87–94, 2018.

[175] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "Rados: a scalable, reliable storage service for petabyte-scale storage clusters," in *PDSW*, 2007, pp. 35–44.

[176] Q. Zheng, K. Ren, G. Gibson, B. W. Settlemyer, and G. Grider, "Deltafs: Exascale file systems scale better without dedicated servers," in *Proceedings of the 10th Parallel Data Storage Workshop*, 2015, pp. 1–6.

[177] A. D. Malony, "Performance observability," Ph.D. dissertation, University of Illinois at Urbana-Champaign, 1990.

[178] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken *et al.*, "Diagnosing performance changes by comparing request flows." in *NSDI*, vol. 5, 2011, pp. 1–1.

[179] R. R. Sambasivan, R. Fonseca, I. Shafer, and G. R. Ganger, "So, you want to trace your distributed system? key design insights from years of practical experience," *Parallel Data Lab., Carnegie Mellon Univ., Pittsburgh, PA, USA, Tech. Rep. CMU-PDL-14*, 2014.

[180] "Influxdb," https://www.influxdata.com/.

[181] T. Islam, K. Mohror, and M. Schulz, "Exploring the capabilities of the new mpi_t interface," in *Proceedings of the 21st European MPI Users' Group Meeting*, 2014, pp. 91–96.

[182] S. Ramesh, A. Mahéo, S. Shende, A. D. Malony, H. Subramoni, A. Ruhela, and D. K. Panda, "Mpi performance engineering with the mpi tool interface: the integration of mvapich and tau," *Parallel Computing*, vol. 77, pp. 19–37, 2018.

[183] K. A. Huck, A. D. Malony, S. Shende, and D. W. Jacobsen, "Integrated measurement for cross-platform openmp performance analysis," in *Int. Workshop on OpenMP*. Springer, 2014, pp. 146–160.

[184] H. Jagode, A. Danalis, H. Anzt, and J. Dongarra, "Papi software-defined events for in-depth performance analysis," *IJHPCA*, vol. 33, no. 6, pp. 1113–1127, 2019.

[185] W. Loewe, T. McLarty, and C. Morrone, "Ior benchmark," 2012.

[186] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Concurrency: the Works of Leslie Lamport*, 2019, pp. 179–196.

[187] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces-a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* IEEE, 2015, pp. 34–39.

[188] G. Douglas and R. Lawrence, "Littled: a sql database for sensor nodes and embedded applications," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 827–832.

[189] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. De Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2015, pp. 1–12.

[190] R. Izadpanah, N. Naksinehaboon, J. Brandt, A. Gentile, and D. Dechev, "Integrating low-latency analysis into HPC system monitoring," in *Proceedings of the 47th International Conference on Parallel Processing*, 2018, pp. 1–10.

[191] J. Reams, "Extensible monitoring with Nagios and messaging middleware," in *26th Large Installation System Administration Conference ({LISA} 12)*, 2012, pp. 153–162.

[192] W. Nagel, A. Arnold, M. Weber, H.-C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and analysis of MPI resources," 1996.

[193] O. Zaki, E. Lusk, W. Gropp, and D. Swider, "Toward scalable performance visualization with Jumpshot," *The International Journal of High Performance Computing Applications*, vol. 13, no. 3, pp. 277–288, 1999.

[194] K. Mohror and K. Karavanic, "Evaluating similarity-based trace reduction techniques for scalable performance analysis," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–12.

[195] M. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[196] e. a. M. Geimer, "The scalasca performance toolset architecture," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 702–719, Apr. 2010.

[197] e. a. P. Alonso, "Tools for Power-energy Modelling and Analysis of Parallel Scientific Applications," in *41st International Conference on Parallel Processing (ICPP)*, 2012, pp. 420–429.

[198] e. a. M. Schulz, "Open|SpeedShop: An Open Source Infrastructure for Parallel Performance Analysis," *Scientific Programming*, vol. 16, no. 2-3, Apr. 2008.

[199] M. Schulz and B. de Supinski, "PNMPI Tools: A Whole Lot Greater than the Sum of Their Parts," in *ACM/IEEE Conference on Supercomputing (SC)*, 2007, pp. 30:1–30:10.

[200] e. a. A. Malony, "Parallel Performance Measurement of Heterogeneous Parallel Systems with GPUs," in *40th International Conference on Parallel Processing (ICPP)*, 2011, pp. 176–185.

[201] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[202] A. Mandal, R. Fowler, and A. Porterfield, "System-wide introspection for accurate attribution of performance bottlenecks," in *Second International Workshop on High-perfromance Infrastruture for Scalable Tools*, 2012.

[203] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing frontiers and innovations*, vol. 2, no. 3, pp. 49–66, 2015.

[204] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, "The paradyn parallel performance measurement tool," *Computer*, vol. 28, no. 11, pp. 37–46, 1995.

[205] S. Benedict, V. Petkov, and M. Gerndt, "Periscope: An online-based distributed performance analysis tool," in *Tools for High Performance Computing 2009.* Springer, 2010, pp. 1–16.

[206] R. Miceli, G. Civario, A. Sikora, E. César, M. Gerndt, H. Haitof, C. Navarrete, S. Benkner, M. Sandrieser, L. Morin *et al.*, "Autotune: A plugin-driven approach to the automatic tuning of parallel applications," in *International Workshop on Applied Parallel Computing.* Springer, 2012, pp. 328–342.

[207] e. a. R. Schöne, "Extending the Functionality of Score-P through Plugins: Interfaces and Use Cases," in *10th International Workshop on Parallel Tools for High Performance Computing*, Oct. 2016, p. 59–82.

[208] D. Böhme, T. Gamblin, D. Beckingsale, P.-T. Bremer, A. Gimenez, M. LeGendre, O. Pearce, and M. Schulz, "Caliper: Performance introspection for hpc software stacks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 47:1–47:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=3014904.3014967

[209] A. Danalis, H. Jagode, T. Herault, P. Luszczek, and J. Dongarra, "Software-defined Events through PAPI," 2019.

[210] A. Morris, A. Malony, S. Shende, and K. Huck, "Design and Implementation of a Hybrid Parallel Performance Measurement System," Sep. 2010.

[211] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the Open Trace Format (OTF)," in *Proceedings of the 6th International Conference on Computational Science*, ser. Springer Lecture Notes in Computer Science, vol. 3992, Reading, UK, May 2006, pp. 526–533.

[212] K. Huck, A. Malony, R. Bell, and A. Morris, "Design and Implementation of a Parallel Performance Data Management Framework," in *34th International Conference on Parallel Processing (ICPP).* IEEE Computer Society, Aug. 2005.

[213] R. Bell, A. Malony, and S. Shende, "A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis," in *European Conference on Parallel Processing (EuroPar)*, vol. LNCS 2790, Sep. 2003, pp. 17–26.

[214] K. Huck and A. Malony, "PerfExplorer: A Performance Data Mining Framework for Large-Scale Parallel Computing," in *ACM/IEEE Conference on Supercomputing (SC).* ACM, Nov. 2005.

[215] F. Song, F. Wolf, N. Bhatia, J. Dongarra, and S. Moore, "An Algebra for Cross-Experiment Performance Analysis," in *Proc. of International Conference on Parallel Processing, ICPP-04*, August 2004.

[216] F. Wolf, B. Mohr, J. Dongarra, and S. Moore, "Efficient Pattern Search in Large Traces through Successive Refinement," in *Proceedings of the European Conference on Parallel Computing (EuroPar 2004, LNCS 3149).* Springer, 2004, pp. 47–54.

[217] e. a. S. Ramesh, "MPI Performance Engineering with the MPI Tool Interface: Integration of MVAPICH and TAU," in *24th European MPI Users' Group Meeting (EuroMPI)*, 2017.

[218] I. Karlin, J. Keasler, and J. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), Tech. Rep., 2013.

[219] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The vampir performance analysis tool-set," in *Tools for High Performance Computing.* Springer, 2008, pp. 139–155.

[220] C. Wood, S. Sane, D. Ellsworth, A. Gimenez, K. Huck, T. Gamblin, and A. Malony, "A scalable observation system for introspection and in situ analytics," in *2016 5th Workshop on Extreme-Scale Programming Tools (ESPT).* IEEE, 2016, pp. 42–49.

[221] V. Balasubramanian, T. Jensen, M. Turilli, P. Kasson, M. Shirts, and S. Jha, "Adaptive ensemble biomolecular applications at scale," *SN Computer Science*, vol. 1, no. 2, pp. 1–15, 2020.

[222] A. Brace, H. Lee, H. Ma, A. Trifan, M. Turilli, I. Yakushin, T. Munson, I. Foster, S. Jha, and A. Ramanathan, "Achieving 100x faster simulations of complex biological phenomena by coupling ml to hpc ensembles," *arXiv preprint arXiv:2104.04797*, 2021.

[223] S. Ramesh, A. Malony, P. Carns, R. Ross, M. Dorier, J. Soumagne, and S. Snyder, "SYMBIOSYS: A methodology for performance analysis of composable hpc data services," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 35–45.

[224] J. Kress, "In-line vs. in-transit in situ: Which technique to use at scale?" 2020.

[225] J. Kress, M. Larsen, J. Choi, M. Kim, M. Wolf, N. Podhorszki, S. Klasky, H. Childs, and D. Pugmire, "Opportunities for cost savings with in-transit visualization," in *International Conference on High Performance Computing.* Springer, 2020, pp. 146–165.

[226] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: Resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[227] A. Goswami, Y. Tian, K. Schwan, F. Zheng, J. Young, M. Wolf, G. Eisenhauer, and S. Klasky, "Landrush: Rethinking in-situ analysis for gpgpu workflows," in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid).* IEEE, 2016, pp. 32–41.

[228] G. Abram, V. Adhinarayanan, W.-c. Feng, D. Rogers, and J. Ahrens, "Eth: An architecture for exploring the design space of in-situ scientific visualization," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 2020, pp. 515–526.

[229] V. Vishwanath, M. Hereld, and M. E. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *2011 IEEE Symposium on Large Data Analysis and Visualization.* IEEE, 2011, pp. 9–14.

[230] T. Terraz, A. Ribes, Y. Fournier, B. Iooss, and B. Raffin, "Melissa: Large scale in transit sensitivity analysis avoiding intermediate files," in *Proceedings of the international conference for high performance computing, networking, storage and analysis*, 2017, pp. 1–14.

[231] D. Huang, Z. Qin, Q. Liu, N. Podhorszki, and S. Klasky, "Identifying challenges and opportunities of in-memory computing on large hpc systems," *Journal of Parallel and Distributed Computing*, 2022.

[232] C. Harrison, P. Navrátil, M. Moussalem, M. Jiang, and H. Childs, "Efficient dynamic derived field generation on many-core architectures using python," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. IEEE, 2012, pp. 583–592.

[233] K. Moreland, C. Sewell, W. Usher, L.-t. Lo, J. Meredith, D. Pugmire, J. Kress, H. Schroots, K.-L. Ma, H. Childs *et al.*, "Vtk-m: Accelerating the visualization toolkit for massively threaded architectures," *IEEE computer graphics and applications*, vol. 36, no. 3, pp. 48–58, 2016.

[234] M. A. Sprague, S. Ananthan, G. Vijayakumar, and M. Robinson, "Exawind: A multifidelity modeling and simulation environment for wind energy," in *Journal of Physics: Conference Series*, vol. 1452, no. 1. IOP Publishing, 2020, p. 012071.

[235] W. Zhang, A. Almgren, V. Beckner, J. Bell, J. Blaschke, C. Chan, M. Day, B. Friesen, K. Gott, D. Graves *et al.*, "Amrex: a framework for block-structured adaptive mesh refinement," *Journal of Open Source Software*, vol. 4, no. 37, pp. 1370–1370, 2019.

[236] A. Merzky, M. Santcroos, M. Turilli, and S. Jha, "Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers," *CoRR, abs/1512.08194*, 2015.

[237] P. Balaprakash, M. Salim, T. D. Uram, V. Vishwanath, and S. M. Wild, "Deephyper: Asynchronous hyperparameter search for deep neural networks," in *2018 IEEE 25th international conference on high performance computing (HiPC)*. IEEE, 2018, pp. 42–51.