Steerable Observability for Bulk-Synchronous Parallel Systems

Submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Department of Electrical and Computer Engineering

Ankush Jain

B.Tech., Computer Science and Engineering, IIIT Hyderabad

Carnegie Mellon University
Pittsburgh, Pennsylvania

May 2026

This research was sponsored by the Los Alamos National Laboratory's Institute for Reliable High Performance Information Technology (IRHPIT), the NetSketch project, and the Mochi project (DOE/NNSA Contract Nos. DE-AC52-06NA25396/394903, DE-AC52-06NA25396/394957, and 89233218CNA000001/394957 respectively), and the New Mexico Consortium. I also thank the current and former members of the Parallel Data Lab (PDL) Consortium at Carnegie Mellon University, including Bloomberg LP, Datadog, Everpure, Google, Jane Street, LayerZero Labs, Meta, Microsoft Research, Oracle Corporation, Salesforce, Uber, and Western Digital, for their interest, insights, and feedback. The views and the conclusions are those of the author and do not necessarily represent the opinions of the sponsoring organizations, agencies, or the U.S. Government.

Acknowledgments

I thank my thesis committee — George Amvrosiadis (chair), Greg Ganger, Andy Pavlo, Vyas Sekar, and Brad Settlemyer — for their guidance and feedback throughout this work.

While the bulk of this document is devoted to the technical contributions of a scenic route through grad school, it is pertinent to take a moment to acknowledge the substantial human element enabling this work. The story arc necessitates that it begin with Saikat, my mentor at MSR Bangalore from 2016–18, who prevailed in an argument about me not wanting to go to grad school. While not entirely convinced, I deferred to his superior intellect and applied and surprisingly got some admits. Next, George, my advisor for the last 8 years. The time and space he gave me to go after misbehaving systems appeared questionable in the moment, but proved absolutely decisive in enabling this work. Chuck for being a wizard of a consultant for when anything caught fire: build systems, fabrics and drivers, my code, other people's code, but somehow never his own. Brad for being engaged throughout: it is rare and valued. Qing for onboarding me with DeltaFS and remaining a close collaborator throughout. Phil and Rob for both being very wise and for willing to make it accessible to me when needed. Vyas for being a black box whose office one leaves with less uncertainty than they enter with — some forced add-ons in the form of lessons from Heilmeier and Walt Disney not diminishing the overall deal value.

Next, the Parallel Data Lab — the people, compute, and the calendar. Greg, now tragically deposed and exiled to an obscure hyperscaler (Google MTV), for an excellent job running the show for most of my tenure here. Bill, Karen, and Joan for all the magical things they do to make everything run, and Joan again for the last-minute typesetting lessons. Jason, Chad, and Mitch for handling all my tantrums when Wolf did not work, and for all the times it did. My appetite for the scale of systems I wanted to do went down by a couple orders of magnitude after I offered to pitch in with unloading the new cluster, and an hour of manual labor later I had unpacked a grand total of 20 blades. The PDL retreat for being the wonderful event it is. While it took me a long time to understand how to leverage it well, this work would not have been possible without the conversations I got to have there. Andy, while unlikely to have been aware of this work until I asked him to be on my committee, for an excellent seminar roster and a YouTube channel, influences from which led to Garth suspecting a direct role in my pro-SQL advocacy. Garth for building this enduring institution and for thought-provoking conversations during the odd appearance. Finally, the students: Rajat, Angela, Michael, Saurabh, Jack, Pratik, Juncheng, Sara, Val, Daniel, Hojin, Daiyaan, Nj, Ziyue, Jekyeom, Sheng, Sherman, Sanjith, Wan, Sam, Theo, and Tim. Time, a plague, and age unfortunately make exhaustive recollection difficult.

I was fortunate to infiltrate cliques beyond PDL through some strategic maneuvering. First, networks counterparts who did not ask questions once I learned to shake my head disappointingly while muttering BBR: Philip, Nirav, Hugo, Anup, Shawn, and Chris. Wiselab folks who learned to tolerate me after I refused to leave CIC 2300: Adwait, Artur, John, Akarsh, Arjun, Tianshu,

Edward, Tarana, Sagar, Shruti, Mallesh, and Humza. Tao for reliable company and for the long marches to Bao and back. Anthony Rowe for funding coffee, 3D printers, and other toys — may the sun shine brightly on his solar panels. Among the broader Cylab/CIC community: Yoshi for teaching me Wittgenstein's takes on paradoxes, Michael Stroucken for the scholar charm that seems to be loosely correlated with progress accelerating, Maverick for being a fellow hueligan, Milind, Lucas, and Theo for the serendipitously formed observability cabal, Brigette for the periodic coffee resupplies, and Zhuo for nothing in particular.

Next, fellow MSR alumni who wandered into CMU contemporaneously and friends I made through them — Sandeep, PV, Suhas, Don, Chirag, Philip, Anish, Raut, and Boby Robert. A random Walmart tennis impulse that Sandeep and PV went along on led to a vibrant tennis scene through which I met many people and burned many calories, although substantially offset by the Page's trips that typically followed after. Philip and his roommates Miguel, Ioannis, and Maggie opened their backyard to my Mazda, where it acquired many brake pads, belts, and control arms, setting an example for other backyard-havers. Likewise for Don and Suhas for abandoning a gaming session and appearing with a spare wheel when one was needed. Chintu for provisioning the consideration for provisions that were instrumental to the job search effort, and Philip (again) for brokering access to the aforementioned provisions. Anup for exhaustive and transparent accounts of his job search process, replete with relevant figures, replicating the methods wherein with a fraction of the diligence proved adequate in securing employment.

Next, businesses, communities, and people that helped build, tune, and maintain my vehicles and myself. Rocky at Kraynick's bike shop for helping build my e-bike and maintain it over the years. Harbor Freight, Rock Auto, and AliExpress for efficient, value-adding operating models. UPMC for being a one-stop shop for systems bottlenecks within myself, and Pitt Dental for picking up the minor residual slack. Highland Park Tennis Club for the wonderful and free summer clinics. Justin and TechSpark for teaching me woodworking—not being able to use the wood CNC router there remains a minor regret. Sandeep, again, for loaning his Mazda as a source of sensors when a notoriously tedious P0171 code appeared in mine. Soylent and Huel for replicable and reasonably complete nutritional formulations. Ami for a meme that slowly acquired salience, leading to substantial personal changes around Spring 2023.

Finally, my roommate Jovina and my parents. Jovina for a fascinating and entertaining character study, teaching me patience, deep breathing, and the ability to find joy in sub-par humor and random babies passing by. My parents for bottling up any skepticism and befuddlement that inevitably accumulated along with the years, and for the moderate but durable lifestyle reforms they undertook to halt the march of atherosclerotic cardiovascular disease. It is finally done, and a new chapter of gainful employment beckons.

Abstract

The *bulk-synchronous parallel* (BSP) paradigm—powering large-scale scientific simulations and distributed ML training—poses unique efficiency challenges. These workloads impose periodic global synchronization across 10,000s of ranks, which compounds small inefficiencies into cluster-wide bottlenecks. Many arise from conditions unknowable before runtime, such as load imbalance and performance pathologies, and can only be addressed through adaptation. Efficiency in production therefore hinges on *adaptation latency*: how quickly conditions can be observed, characterized, and acted on. BSP synchronization boundaries provide a natural structure for such observations and interventions: they define when distributed state is globally interpretable and when coordinated change is well-defined. This thesis argues that general-purpose infrastructure for adaptive feedback loops is the missing piece in today's BSP systems.

We develop this claim through two studies that instantiate the same primitive—a feedback loop over application state—with starkly different outcomes. On the data path, CARP closes an online loop for range-query-optimized data ingestion: it periodically materializes a view of global key distributions via in-situ reduction and uses it to recompute load-balanced range partitions as distributions evolve, matching the query performance of a full sort with up to $4.9\times$ faster ingestion. On the compute path, we study placement in adaptive mesh refinement codes, where apparent load imbalance is confounded by fail-slow hardware, fabric pathologies, and tuning artifacts. Here the loop was manual and offline: each iteration required collecting bulk traces, analyzing them post-hoc, and re-executing with a refined hypothesis. Isolating true placement effects consumed months even though individual mitigations typically took days. With reliable telemetry established, we design CPLX, a tunable placement policy that achieves up to 21.6% runtime improvement. CARP demonstrates that online feedback loops are effective when they exist. The AMR study demonstrates the cost when they do not.

Because these effects recur unpredictably, production BSP systems need an always-on, programmable loop for observing and responding to runtime behavior. Coarse views surface deviations, and the loop can be steered on demand into narrower views to localize and act on them. We call this *steerable observability*. ORCA realizes this via a tree-based overlay for programmable feedback and control in BSP systems, materializing runtime views through in-situ reduction and providing a timestep-consistent control plane for coordinated interventions. On a real AMR code at 4096 ranks, ORCA collects detailed traces at 1–2% overhead versus conventional HPC tracers at 56–81% and eliminates 98.5–99.999% of telemetry at source for evaluated queries. In an evaluation of its closed-loop capabilities using an LLM agent, a performance anomaly that took months to diagnose manually is localized in 27 minutes.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Contents

The Bulk Synchronous Parallel (BSP) paradigm powers some of our most critical computational tools, from large-scale scientific simulations in astrophysics and climate science to distributed training of foundation models in machine learning. These workloads—consisting of large parallel computations interspersed with I/O and global synchronization—currently operate at the scale of tens of thousands of accelerator nodes [1–4] and continue to grow with time [5]. As these applications scale, their efficiency has datacenter-level impacts on both cost and time-to-discovery.

The tightly synchronized nature of BSP creates distinct challenges along the data and compute paths. On the data path, bursts of checkpointed state create massive I/O demands on the storage and analytics infrastructure [6–8]. On the compute path, frequent global synchronization amplifies the system's sensitivity — factors such as execution variability [9–19], failures [20, 21], and correctness risks [22] from corrupted state affect workload-wide progress. Runtime artifacts that would be manageable in loosely coupled systems, such as minor OS-level interruptions or thermal throttling, become critical scaling barriers when every worker must wait for the slowest participant.

Across two studies spanning the data and compute paths, we find that many of these efficiency bottlenecks are driven by *runtime unknowns*: conditions that cannot be predicted reliably before execution. In this regime, the decisive factor is how quickly these conditions can be observed, characterized, and mitigated. On the data path, CARP tackles range-query performance under skewed and evolving key distributions by reconstructing global distributions in situ and adapting partitioning as the run evolves. On the compute path, our placement study in *adaptive mesh*

refinement (AMR) codes shows the failure modes of slow turnaround: placement effects were repeatedly confounded by fail-slow hardware, fabric pathologies, and tuning artifacts, forcing months of reruns and offline telemetry analysis. We view these as two instances of the same primitive—a runtime feedback loop over distributed state—online and closed in one case, manual and slow in the other. Because the governing conditions are fundamentally unknowable in advance, addressing them in production requires general-purpose infrastructure for closing these loops, which BSP workloads largely lack today.

Closing these loops requires two capabilities: views materialized from distributed application state, and coordinated actions informed by those views. BSP synchronization boundaries are the natural points where both are well-defined: global state is coherent, and coordinated change can be applied consistently across ranks. Together, these enable a class of adaptive workflows over always-on visibility into runtime behavior, ranging from steering what is observed to intervening on application behavior—a paradigm we call *steerable observability*. This subsumes fixed-function loops like CARP's renegotiation protocol, and enables online monitoring, tuning, and root-cause localization for problems encountered during the AMR study. We propose ORCA (*Observability with Real-time Control and Aggregation*), a programmable overlay network that realizes steerable observability with synchronization-aligned views and control semantics.

Thesis Statement: *A large class of efficiency bottlenecks in BSP systems arise from conditions that are unknowable before runtime. Such conditions are systematically addressable through adaptive mechanisms if the requisite feedback loops exist, and perpetuate if they do not. A general-purpose substrate for steerable observability can realize programmable feedback loops at scale.*

The rest of this chapter motivates programmable feedback loops through two studies spanning the data and compute paths (§1.1), presents steerable observability and its realization in ORCA (§1.2), lists contributions (§1.3), and outlines the rest of the thesis (§1.4).

1.1 The Case for Programmable Feedback Loops in BSP

We begin by defining the anatomy of a feedback loop over BSP execution (§1.1.1), then examine two studies that instantiate the pattern with different outcomes. CARP demonstrates a fixed-function online loop for adaptive range partitioning on the data path (§1.1.2). Our AMR placement study demonstrates the cost when the loop is manual and offline on the compute path (§1.1.3). We conclude by characterizing the infrastructure gap that separates the two (§1.1.4).

1.1.1 Anatomy of a BSP Feedback Loop

BSP execution proceeds in *timesteps*, with work partitioned across parallel processes called *ranks*. Ranks compute independently during a timestep, then synchronize via global collectives before the next timestep begins. Periodically, *checkpoint epochs* flush distributed state to storage for fault tolerance and offline analysis. A coherent global view of execution is only defined at

synchronization boundaries. Local state between boundaries explains how individual ranks contribute to the next boundary outcome, but the boundary view is the entry point for diagnosing both correctness and performance. A feedback loop over BSP execution operates on this cadence: operators consume views of system state and drive actions in response.

- **View.** A view answers a specific question about execution, such as load distribution or straggler identification via transforms over system state. Views can either reduce global state to expose system-wide conditions, or rank-local state to attribute them to individual ranks.
- **Action.** An action is a coordinated update applied at all ranks in a timestep-consistent manner. Actions may steer observability itself, or steer the application via interventions such as hyperparameter updates.
- **Operator.** Operators consume views and drive actions — they may be human or automated agents, or fixed policies.

1.1.2 CARP: Adaptive Online Range Partitioning

Range queries over scientific data are fundamental to analysis, but expensive to support at scale. Post-processing approaches like out-of-core sorts [23] require multiple additional passes over data, incurring write amplification and reducing the effective application bandwidth. While in-situ hash partitioning for point lookups has been demonstrated viable for scientific data [24], range partitioning is harder — key distributions are unknown, highly skewed, and evolve over time.

CARP [25] addresses this with adaptive online range partitioning guided by a reconstructed view of the global key distribution. A *renegotiation* protocol materializes this view via in-situ reductions over per-rank state. A policy triggers renegotiation at fixed intervals to obtain the current distribution, which is used to compute new load-balanced partition boundaries and reshuffle data en-route to storage. By range-partitioning data in a single streaming pass, CARP avoids the write amplification of post-hoc sorting while enabling comparable query performance, achieving up to $4.9\times$ faster ingestion in our evaluation.

1.1.3 AMR Placement: A Manual Loop for Compute Pathologies

Adaptive Mesh Refinement (AMR) codes are a simulation paradigm widely considered challenging to load-balance [26, 27]. The mesh evolves dynamically to track changing physics, requiring frequent rebalancing, but load characteristics and placement effects have not been well studied in practice. We initially set out to build telemetry-driven placement policies that could adapt to these characteristics at runtime. In practice, we found placement effects repeatedly confounded by unrelated artifacts—fail-slow hardware, fabric pathologies, and tuning issues—that had to be identified and eliminated before placement could be isolated.

The resulting loop—collect traces, analyze offline, refine hypothesis, reconfigure and rerun—was entirely manual and slow. Conventional HPC tracers [28, 30] emit large per-rank logs in formats [31, 32] that required expensive *extract-transform-load* (ETL) before any analysis could begin. Eliminating confounders consumed months of iteration even when individual mitigations

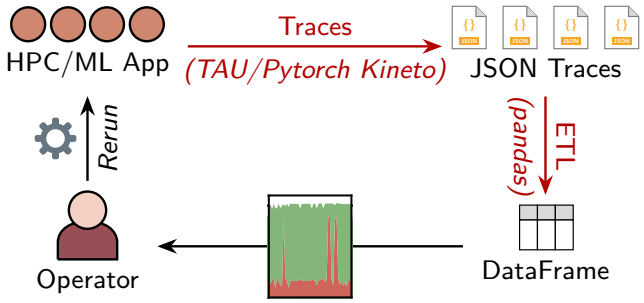


Figure 1.1: *Analysis workflow via state-of-the-art HPC/ML tracing tools [28, 29]. Traces as collected as per-rank logs, requiring expensive* extract-transform-load *(ETL) before any analysis can begin. Reconfigurations such as additional instrumentation require rerunning the application and repeating the workflow, slowing down diagnosis.*

typically took days once identified. With a stable measurement baseline finally established, we found a fundamental tradeoff between compute load balance and communication locality, and designed a placement policy called CPLX [33] to provide tunable control over this tradeoff. In evaluation, CPLX reduced runtime by up to 21.6% in real AMR workloads over tuned baselines, but like CARP's renegotiation interval, exposed another context-dependent tunable knob.

1.1.4 Why Programmable Feedback Loops?

Both studies instantiate the same primitive—a feedback loop over distributed execution—but with starkly different costs. CARP uses an online loop, built on a materialized view of global key distribution, to adapt to skew and drift. The AMR study documents broader examples of runtime inefficiencies, and the costs of acquiring feedback via offline analysis of bulk traces. Three compounding factors contribute to these costs:

- **Deferred Materialization Costs.** Raw telemetry is captured at full volume and persisted before any reduction occurs. Because the right subset is not knowable without feedback, the paradigm intrinsically tends to overcollect. Reductions that could run in situ over an HPC fabric are instead forced through a write–read–parse cycle of telemetry from O(10-100K) ranks on the slowest tier in the hierarchy: storage.
- **ETL Costs.** Conventional tracers [28–30, 34, 35] write per-rank logs [31, 32] in flat layouts with little awareness of BSP structure. Causal relationships created by synchronization are not reflected in the on-disk organization, so even simple questions require expensive ETL [33, 36] before queries can run.
- **Rerun Costs.** Refining a hypothesis, changing what is observed, or testing a mitigation typically requires reconfiguration and re-execution. Turnaround is bounded by reruns and offline pipelines rather than by the running job.

Capability	Design Requirement	Example Use Cases
Online views	Coordinated cross-rank streaming aggregations	Straggler distributions at collectives
Offline analytics	In-situ analytics to filter relevant telemetry, repartitioning for faster offline analytics	Compute kernel telemetry to facilitate development and simulation of placement policies
Online interventions	Control plane with consistency guarantees	Additional instrumentation to localize anomalies, or hyperparameter tuning
Offline interventions	*Not in scope*	New placement algorithms, bad hardware removal

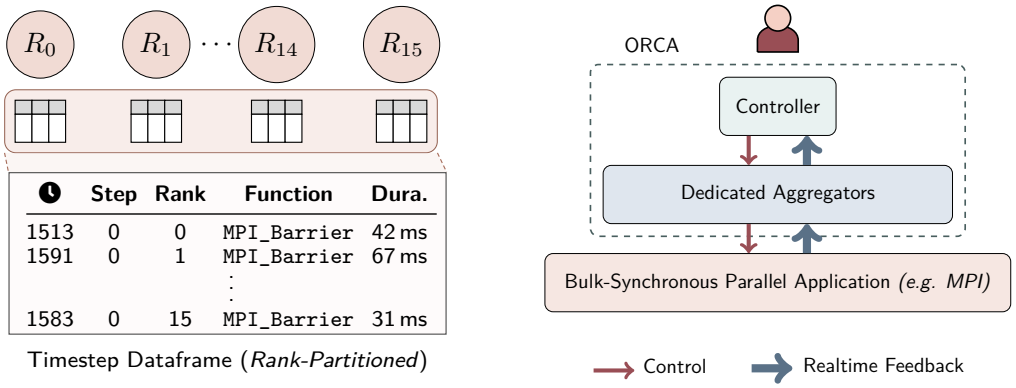
Table 1.1: *Steerable observability capabilities and design requirements, along with example AMR use cases.*

These costs can be addressed via a common streaming analytics and control plane — one that provides online feedback loops via materialized views and online interventions, and accelerates offline workflows via in-situ filtering and data reorganization en route to storage. Table 1.1 grounds these capabilities in example use cases from the AMR study. CARP serves as a template in two ways: it demonstrates the effectiveness of fixed-function feedback loops over distributed state, as well as in-situ reorganization to accelerate offline analyses. A spectrum of workflows can thus be accelerated, from offline telemetry analytics to feedback-guided online interventions.

1.2 ORCA: Programmable, Real-time Feedback and Control

Our goal is always-on, programmable, BSP-aware observability: lightweight aggregates in the common case, with depth when needed. We call this paradigm *steerable observability*. While event-based streaming systems [37, 38] are widely deployed in cloud environments, they treat telemetry as detached events that are serialized and shipped to separate analytics clusters. Causal windows must then be reconstructed manually: establishing the straggler at a barrier requires aggregating that barrier's durations from tens to hundreds of thousands of ranks in real applications. At these scales, exporting telemetry to an external analytics cluster through a conventional TCP stack moves it off the primary compute resource, introduces TCP-induced jitter, and fails to leverage the available high-performance fabrics. We show that BSP telemetry has a regular causal structure amenable to columnar representations [39], RDMA, and operator pushdown. Global synchronization partitions execution into causally independent timesteps: telemetry can therefore be accumulated locally and flushed each timestep into rank-partitioned in-memory *timestep dataframes* for analysis. These timestep dataframes (Fig. 1.2a) capture both the global workload-wide state at the end of the timestep and the rank-local events that led to that state.

Leveraging the timestep dataframe abstraction, we propose ORCA (*Observability with Real-time Control and Aggregation*), a system for steerable BSP observability. ORCA (Fig. 1.2b) builds a *tree-based overlay network* (TBON) for real-time feedback loops between operators and applications via a controller at the root and a tier of dedicated aggregators attached to the application ranks. Feedback is materialized via in-situ reductions over timestep dataframes, while control is applied



(a) *The timestep dataframe abstraction* ***(b)*** *ORCA architecture*

Figure 1.2: *ORCA architecture. (Left) telemetry is flushed into* timestep dataframes *after every timestep. Consisting of rank-partitioned shards, these form causally independent units for in-situ analysis. (Right) a simplified view of ORCA's tree-based overlay, showing a root controller and a tier of dedicated aggregators, enabling online feedback and control via in-situ reductions and broadcast.*

via broadcast. A SQL-based analytics interface transforms dataframes en route to storage and visualization, with operator pushdown and RDMA to minimize data movement across the overlay. A custom two-phase commit variant called TS2PC ensures that control inputs are applied on all ranks at the same timestep. On a real AMR code at 4096 ranks, ORCA collects detailed traces at 1–2% overhead versus 56–81% for conventional tracers, eliminates 98.5–99.999% of telemetry at source for evaluated queries, and in a closed-loop evaluation with an LLM agent localizes a performance anomaly in 27 minutes that previously took months of manual iteration.

Scope. ORCA provides two capabilities: SQL-based streaming analytics and timestep-consistent control. The system is agnostic to how telemetry originates and to who drives the loop— operators may be human or automated agents, or fixed policies. Telemetry can originate from probes, counters, or application annotations, as long as it can be represented as structured columnar data. Where SQL is insufficient, ORCA supports user-defined *map* operators within the overlay.

Why Now. Two trends make steerable observability both practical and urgent:

- Trace analytics have converged on dataframe-style queries [34–36, 40], but telemetry is still collected as per-rank logs whose formats and layouts force expensive ETL. Columnar formats such as Arrow [39] and Parquet [41], along with DataFusion [42] — an extensible Arrow-based query engine — provide effective building blocks for efficient in-situ pipelines.
- Operational demand in large-scale ML training is driving observability for issues such as straggler localization [14–19], fault diagnosis [20, 21], and training correctness [22]. These are typically implemented as bespoke pipelines, but can be generalized by a programmable BSP-aware substrate.

1.3 List of Contributions

CARP (*Continuous Adaptive Range Partitioner*).

- An adaptive in-situ range partitioner for streaming scientific data, designed to accelerate range queries under unknown and evolving key distributions.
- Partitions data in a single streaming pass with $1\times$ write amplification during ingestion, and is 2.8–$4.9\times$ faster than post-processing approaches.
- Shows that partially ordered layouts can deliver range-query performance comparable to fully sorted layouts, while being up to two orders of magnitude faster than auxiliary bitmap indices on the evaluated workloads.
- Complements DeltaFS's in-situ hash partitioning [24] by providing the in-situ range-partitioning counterpart needed for range queries.

Placement in AMR (*Adaptive Mesh Refinement*) Codes.

- An end-to-end empirical study integrating collection, tuning, and placement, showing how artifacts such as overheating, fabric bugs, and operation ordering obscure placement effects.
- CPLX, a tunable placement policy that exposes the tradeoff between compute load balance and communication locality, achieving up to 21.6% runtime reduction over tuned baselines across 512–4096 ranks.

ORCA (*Observability with Real-time Control and Aggregation*).

- A BSP-aware observability and control substrate that aligns both analysis and coordinated interventions to synchronization boundaries.
- The *timestep dataframe* abstraction, partitioning BSP telemetry into causally independent windows, and a *(timestep, rank)*-partitioned Parquet layout, yielding up to $44\times$ smaller traces and up to $6{,}800\times$ faster post-mortem analysis.
- An incast-optimized RDMA transport for tree topologies that enables detailed trace collection at under 2% runtime overhead with real AMR codes, versus 56–81% for conventional tracers.
- A SQL-based in-situ programming interface over timestep dataframes with operator push-down, eliminating 98.5%–99.999% of telemetry at source in evaluated queries.
- A timestep-consistent control plane via TS2PC (*timestep-linked two-phase commit*) that enables coordinated runtime changes aligned to BSP synchronization.
- In a closed-loop evaluation using an LLM agent, an anomaly that took months to isolate manually is localized in 27 minutes.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

- Chapter 2 provides background on BSP environments and efficiency bottlenecks.
- Chapters 3, 4, and 5 present CARP, the AMR study, and ORCA respectively.
- Chapter 6 concludes the thesis and outlines future directions.

Chapter 2

Background: Data, Compute, and Feedback Bottlenecks in BSP Systems

Contents

We first review key concepts and challenges in the bulk-synchronous parallel (BSP) paradigm, beginning with an overview of the computing environment (§2.1), followed by data management challenges (§2.2), compute efficiency challenges (§2.3), and limitations of existing feedback mechanisms (§2.4). While primarily focusing on scientific computing, we also discuss distributed machine learning training workloads, as their shared bulk-synchronous characteristics create similar challenges and inform our later work.

2.1 BSP: Execution, Architecture, and Applications

The BSP model consists of alternating compute and synchronization phases (see fig. 2.1). Execution begins with decomposition of the problem into independent subproblems assigned to workers, and proceeds as discrete timesteps. Each timestep consists of a compute phase followed by a global synchronization phase, where synchronous collectives [43, 44] ensure all ranks advance together. Every few timesteps, a checkpoint is triggered for both fault tolerance and offline analysis. This tight coupling is the root of cluster efficiency problems: synchronous checkpoints create coordinated I/O bursts that stress the storage system, while global collectives make overall performance acutely sensitive to the slowest worker. Growth in cluster sizes leads to increased checkpointing frequency, as the increased failure probabilty makes forward progress more valuable [6].

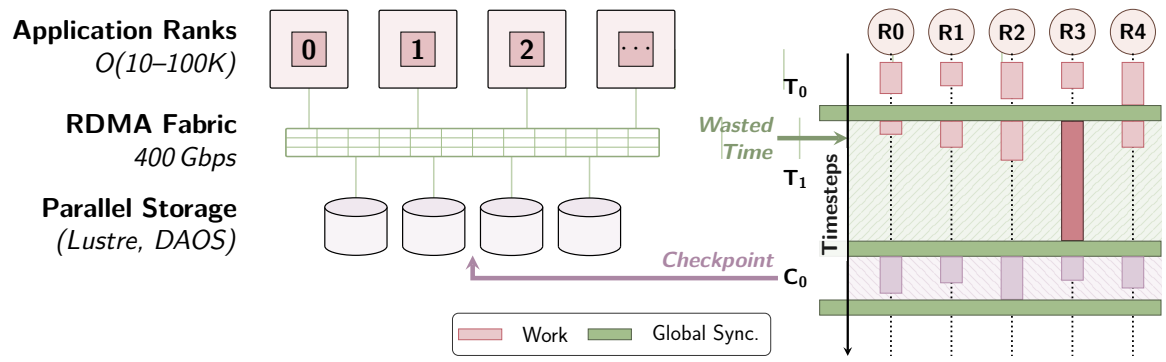


Figure 2.1: *BSP architecture and execution model. (Left) Applications consist of CPU or GPU ranks, connected via an RDMA fabric and parallel storage. (Right) Execution proceeds in timesteps followed by global synchronization via collectives. Runtime is therefore dictated by the slowest rank. Periodically, checkpoints are triggered for fault tolerance and offline analysis.*

HPC Cluster Architecture and Applications. Large-scale HPC clusters typically consist of O(10–100K) nodes [45] connected by a high-performance fabric [46–48]. Each node typically consists of CPUs and typically multiple GPUs, which are increasingly important for scientific computing [49, 50]. Storage is traditionally provided by parallel filesystems [51] but object stores [52] and richer commercial offerings [53, 54] are increasingly common [55].

Scientific applications target fundamental questions across domains such as cosmology and climate science, problems that require simulating complex physical phenomena over extended spatial and temporal scales. These applications can occupy the majority of a supercomputer's resources for months at a time [1, 2, 56]. From a systems perspective, they can be simplified into two common archetypes: *particle codes* [57] and *mesh codes* [26]. Particle codes track the state and interactions of billions individual discrete entities as they move through the simulation domain. Mesh codes discretize the spatial domain into grids and compute the evolution of properties of interest at each grid point.

Distributed ML Training. Distributed ML training represents a similar and increasingly important cluster workload, occupying tens of thousands of GPUs and running for months [3–5]. The cluster architectures are comparable, though ML clusters often add a *frontend* Ethernet network for general datacenter connectivity [58]. While the parallelization strategies differ—ML employs a mix of data, tensor, and pipeline parallelism—the execution model still relies on the same core BSP primitives of synchronous collectives [58] and periodic checkpointing [7].

2.2 Data Management Challenges

Checkpointing creates I/O-bound bottlenecks in BSP systems, as coordinated writes across thousands of ranks stress parallel filesystems and consume significant runtime [59]. Traditional workflows then require scientists to re-read and analyze this data after simulation completion, consuming additional I/O bandwidth and computational resources while delaying scientific

discovery. Ingesting data using conventional databases is not viable, as databases incur multiple on-disk writes for each application write [60] (*write amplification*), offering poor tradeoffs for ingestion-heavy workloads.

In-situ processing [61] analyzes data while it streams from application to storage, reducing both I/O overhead and time-to-insight. This approach is employed for multiple scientific use cases ranging from visualization [62–65] to indexing [66, 67]. In-situ reorganization using hash-partitioning and shuffling has been demonstrated to be viable at HPC scale [24]. However, unlike databases that provide general query interfaces over composable indexes, existing in-situ systems are purpose-built for specific use cases and lack programmability. A critical gap remains in range query support, essential for analytics on continuous physical attributes but absent from existing in-situ frameworks.

2.3 Compute Efficiency Challenges

Cluster utilization challenges persist in BSP systems: a study of exported traces found more than 60% of runtime spent in MPI [43] operations (communication and synchronization), even at modest scales. We now provide additional context on performance tooling and workload characterization bottlenecks that contribute to these challenges.

Performance Analysis Tools. While real time cluster-level monitoring is common [68], application-level performance analyses are largely post-mortem [28, 30, 35, 69–71]. Two main approaches are tracing and profiling. Tracing generates fine-grained event logs for each worker (*rank*), which are then aggregated into a composite view via postprocessing. The volume of generated trace data presents an analytics challenge, delaying insight [34], and remains unviable at the largest scales [16]. In comparison, profiling generates phase-wise summaries of execution for each rank. While far more scalable, this obscures fine-grained interactions that degrade tightly-coupled BSP execution.

Workload Characterization Bottlenecks. While efficiency challenges are known and attributed to load imbalance [26, 27, 72], deeper understanding remains elusive, partly because of visibility bottlenecks discussed above. Characterization of workload properties such as compute kernel distributions, communication bottlenecks, placement and load-balancing requirements remain poorly understood. This is further exacerbated by the separation of analyses from data collection, as the latter are undertaken by separate groups without access to the underlying hardware that generated the data [73]. As a result, distinguishing workload characteristics from environment artifacts becomes even more difficult.

2.4 Feedback Mechanisms and Limitations

Computational Steering. Computational steering refers to interactive workflows where output from a running simulation is observed and parameters are adjusted mid-execution [74]. The idea

dates to the early 1990s [74, 75] and continues through modern in-situ visualization tools [62, 65] — mostly fixed-function pipelines over domain-specific data models. Many systems recognized that steering distributed simulations requires consistency guarantees: CUMULVS [74] and RMOST [76] observe that synchronization boundaries are the natural points at which updates may be applied, with RMOST formalizing this as a DSM-based (*distributed shared memory*) consistency model. Pathfinder [77] proposes coordinated steering via transaction-based consistency detection with rollback. However, none addresses the full problem: a scalable protocol for consistent control inputs from asynchronous agents, runtime-programmable analytics, and standard data models for the ad-hoc aggregations that observability requires.

Performance Analysis Infrastructure. MRNet [78] implemented a TBON (*tree-based overlay network*) [79] for scalable tool communication, used as a reduction backend for commercial debuggers [80], but its compiled packet-based filter model was not designed for analytics. Chimbuko [81] implements in-situ trace analysis over TAU [28] and ADIOS2 [82], but does not preserve cross-rank causality, provides no consistent control path, and its analytics are fixed-function. Trace analytics increasingly use dataframe-style interfaces [34, 35, 40, 83], but require expensive ETL from per-rank logs and operate exclusively post-hoc [33, 36].

ML Training Observability. The ML training community has independently converged on the same problem. As clusters scale to tens of thousands of GPUs, the same BSP fragility has driven a wave of recent systems for detecting and localizing training performance anomalies [14–19], diagnosing faulty hardware [20, 21], and enforcing training correctness [22]. Each builds bespoke infrastructure for subsets of collection, detection, and actuation from scratch, using selective tracing and event streaming [37] to limit telemetry volume [18] or relies on heuristics to trigger collection of local counters if pathologies are detected [15, 21].

Chapter 3

CARP: Range Query-Optimized Indexing for Streaming Data

Contents

CARP enables ingestion of scientific checkpoint data for efficient range queries in a single streaming pass, via an adaptive in-situ range partitioning mechanism. Range queries over continuous attributes like temperature and velocity are fundamental to scientific analysis, enabling filtering of high-energy particles [84] or tracking shock waves in fluid simulations [85–87]. Efficient range queries require key locality in the stored layout, but existing mechanisms for creating such layouts require post-processing. This incurs additional I/O passes and reduces effective ingestion bandwidth. As checkpoint data volumes grow with application scale, these post-processing costs increasingly dominate time-to-discovery [88].

Among existing indexing approaches, those that enable range queries force undesirable

tradeoffs between ingestion and query performance, while in-situ approaches do not produce locality-friendly layouts without additional I/O passes. Distributed out-of-core sorts [23] produce fully ordered layouts which enable both efficient *location* of keys overlapping a given range, and efficient *retrieval* via large sequential reads. This enables excellent query performance but requires multiple additional passes, slowing ingestion. Auxiliary bitmap indices [89] do not reorder data but generate additional index structures. They can accelerate location of matching keys, but retrieval still incurs random reads over scattered keys. Among in-situ approaches, DeltaFS [24] demonstrates the viability of in-situ hash partitioning via an embedded all-to-all shuffle on MPI ranks, but hash-based partitioning destroys key locality, making it unsuitable for range queries.

The central challenge of extending in-situ partitioning to range queries is that of providing efficient ingestion for unknown key distributions. Our analysis of real scientific codes shows that these distributions are often highly skewed and can shift significantly across epochs (§3.2). Distributed databases manage range-partitioned layouts by reactively reordering on-disk data in response to partition imbalance, incurring unacceptable write amplification [60, 90]. Our goal is an adaptive partitioner that makes no assumptions about key distributions, and maximizes ingestion performance by never reordering on-disk data, while still enabling efficient range queries. This requires mechanisms for discovering global key distributions, reacting as they evolve, and handling out-of-bounds keys, all while maintaining high storage utilization.

CARP realizes adaptive single-pass range partitioning via a feedback loop for reconstructing and adapting to global key distributions. A primitive called *renegotiation* is used to materialize the view for this loop using a reduction tree over local histograms maintained at shuffle senders, which is used to compute and broadcast new load-balanced partition boundaries.Trigger mechanisms invoke this process periodically, and these partition boundaries inform shuffling until the next renegotiation. An out-of-bounds mechanism enables both bootstrapping and handling keys that fall outside the current partition ranges. Shuffle receivers accumulate writes to per-rank append-only logs, never reordering on-disk data. Because the storage layer is fully utilized with append-only I/O, CARP incurs no write overhead over unpartitioned I/O in evaluated configurations. The resulting data layout is partially ordered, which requires query-time reconciliation. In our evaluation, this reconciliation overhead is negligible, with query performance comparable to fully sorted layouts and upto $4.9\times$ faster ingestion.

The rest of this chapter provides background on data organization for range queries and the tradeoffs involved (§3.1), characterizes key distributions and their drift in real scientific codes (§3.2), presents the design of CARP and its storage backend KoiDB (§3.4), evaluates performance and tuning behavior (§3.6.2, §3.6.1), and discusses deployment considerations (§3.7).

3.1 Background and Motivation

Data generated by many scientific applications consists of simulated entities such as particles and mesh cells, each storing several attributes (e.g. energy, velocity) into a *record*. This data is

Table 3.1: *Range query indexing approaches.*

Approach	In-situ	Efficient Indexing	Efficient Querying
Bulk Sorting (Clustered)	×	×	✓
Bulk Sorting (Auxiliary)	×	×	×
Bitmap Indexes (FastQuery)	×	×	×
DB Indexes (LSM-Tree, B-Tree)	✓	×	✓
DeltaFS	✓	✓	×
CARP	✓	✓	✓

written out at fixed intervals and not mutated once written, making bulk indexing feasible. The data is then analyzed using point queries (e.g., retrieving particles by ID) or range queries (e.g., retrieving particles with energy in a certain range). Range queries are needed for interacting with continuous attributes and for filtering data using ranges of relevant values (e.g. energy bands, temperature and blast wave fronts).

An efficient range-query index must locate ranges of values quickly. Sorted indexes do this by creating indexed attribute (henceforth referred to as *key*) locality in the data layout, so that the keys for a range can be located in one or few data partitions. Clustered indexes further improve query performance by storing all records with their corresponding keys. This allows for data for a query to be retrieved efficiently via fewer, contiguous reads. In contrast, auxiliary indexes need multiple random reads to retrieve all attributes of interest in the range. In general, a database table can have only one clustered index. The clustered index is typically reserved for the attribute most commonly used in queries [91, 92].

Table 3.1 shows that no current approach enables both in-situ efficient indexing and efficient querying for ranges. We summarize the characteristics of the indexing approaches below. We discuss their *index construction* performance and break down their query performance into *index lookup* and *data retrieval*.

Index Construction. Bulk parallel/distributed sorting [23, 93, 94] approaches support range queries but require post-processing. These can be used to build either clustered or auxiliary sorted indexes. FastQuery [89], on the other hand, builds auxiliary bitmap structures where keys are sorted and stored separately with a pointer to their records. Clustered indexes enable more efficient queries but require more I/O to reorder and write the entire dataset. FastQuery does not reorder data, but results in data scans, computing large bitmap vectors, and writing auxiliary structures to disk.

To avoid expensive post-processing, techniques have been proposed to intercept application writes and index them in-situ before data is written to disk. DeltaFS [24] shuffles data into hash-based partitions and requires no post-processing, but it can only support point queries on the partitioned data. Distributed databases maintain online indexes but offer lower write

efficiency as they periodically reorganize on-disk data [60]. Databases are designed for mixed workloads with updates, while lightweight indexes are better suited to scientific data.

Index Lookup and Data Retrieval. Efficient range queries require quickly finding matching records using the index and retrieving the records from storage using large sequential reads. Only sorted and clustered indexes provide both of these properties. Auxiliary indexes can return the offsets at which matching rows are present, but retrieving those records requires multiple random reads. Additionally, FastQuery queries require provisioning enough compute nodes to fit large index structures in memory, and queries can only be processed once these indexes are loaded.

With CARP, we aim to pair the query latency of a sorted and clustered index with the reduced overhead of an in-situ approach. We discuss the resulting challenges in the next section.

3.2 Range Query Challenges

Sorted, clustered indexes provide the lowest query latency but require post-processing to reorder data. Sorting large on-disk datasets is even more expensive than building auxiliary indices, as it requires multiple read/write passes over the data [23].

As an alternative to reordering via post-processing, data can be partitioned into *range-based partitions* while in transit from the application to storage. This accelerate queries, as only a small number of relevant partitions need to be retrieved from storage for each query. While range partitioning is a well-known technique in distributed databases, adapting it to in-situ partitioning of scientific data requires overcoming two key challenges we describe next.

Write Rate. To organize data in a way that does not reduce the application write throughput, it is not sufficient to partition data online. The indexing pipeline must be designed to not require any storage bandwidth for reordering or index maintenance. Conventional databases use reordering to maintain their partitions, which reduces effective storage bandwidth [95, 96]. As on-disk partitions grow beyond configured sizes, they are split into smaller partitions and migrated across nodes.

The maximum achievable write rate is a function of the average number of I/O operations performed for each application write, also called the *Write Amplification Factor* (WAF). A WAF of 10 will turn an I/O phase lasting 12 minutes into 2 hours, which is wasteful for write-intensive large-scale scientific applications. WAF for write-optimized single-node database indexes has been measured at 19-37 [60] and is much higher for B-tree indexes and distributed databases [97]. In-situ strategies that have a high WAF would not outperform post-processing approaches (WAF of 2-3$\times$). To maximize gains from an in-situ indexing approach, we constrain our design to a WAF of 1$\times$.

Adaptivity. Effective range-partitioning requires the key space to be partitioned such that the generated data partitions are relatively balanced. This is important for both write and query performance. On the write path, this ensures that the application is not slowed down by a straggler writing a significantly larger partition than others. On the query path, imbalanced

partitions will lead to much slower performance for queries corresponding to those partitions.

However, we find that keys in scientific data are often skewed in unpredictable ways and change as the simulation progresses. Specifically, we studied the energy distributions of two applications — VPIC and Phoebus. VPIC is a plasma physics code that simulates particle physics phenomena such as magnetic reconnection [84]. Phoebus is a mesh-based hydrodynamics code that we run with a Sedov blast wave setup [87]. We ran both codes at 512 ranks and studied the energy distributions in the output data.

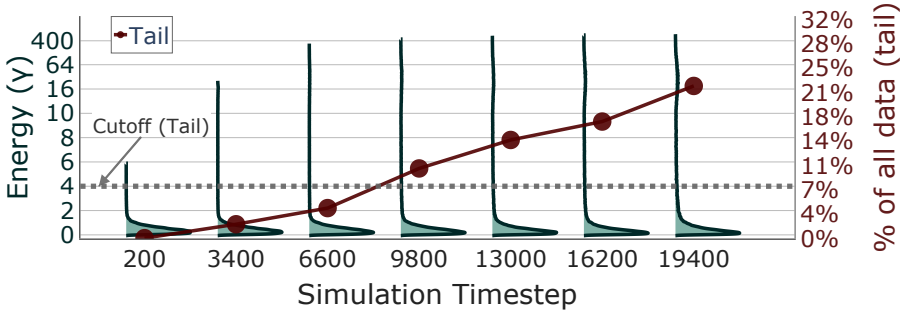


Figure 3.1: *VPIC Energy Distribution. Distributions are shown as histograms overlaid with line plots for interesting bands. VPIC distributions are highly skewed and shift significantly over time.*

figs. 3.1 and 3.2 show the energy distributions of both the codes over time. For VPIC (fig. 3.1), we find that the energy distributions are highly skewed with most particles falling between 0 and 1 with long tails that get longer and heavier as the simulation progresses. Towards the second half of the simulation, 20-30% of the data is contained within the tail generating a bimodal distribution with a second mode between 16 and 64.

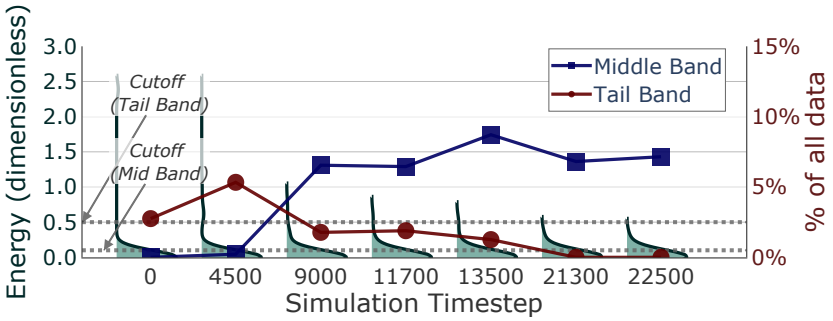


Figure 3.2: *Adaptive Mesh Refinement Energy Distribution. AMR distributions are also highly skewed and shift significantly over time.*

For Phoebus (fig. 3.2), we also see a highly skewed distribution with a tail. Initially, there is a high energy explosion but most of the mesh has no energy. Over time, the energy from the explosion dissipates into a bigger area and moves the distribution into a medium energy band.

These findings highlight the challenges with partitioning keys from scientific workloads. A highly precise reconstruction of the key distribution is necessary to divide the keyspace into a large number of balanced partitions. It is also impractical for any user-provided static range partitioning scheme to accommodate such skewed and dynamic key distributions. A distributed database would only be able to balance its partitions after a series of splitting, rewriting, and merging operations, incurring prohibitive index construction costs. An ideal range query index would discover and adapt to workload characteristics and reorder data without consuming excess storage bandwidth.

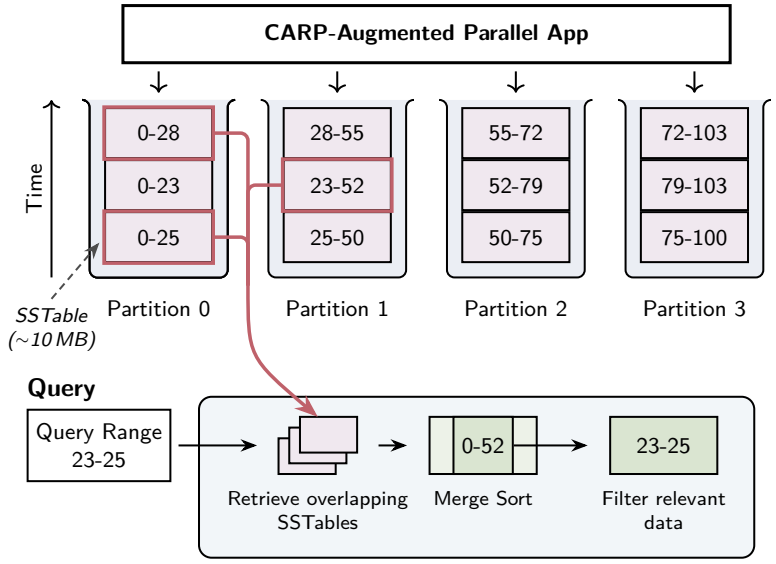


Figure 3.3: *A logical view of CARP's data layout and querying. Incoming data is partitioned and stored as SSTables, and partition boundaries shift with key distribution changes. Queries retrieve relevant SSTables and merge them.*

3.3 Design Principles of CARP

We have designed a streaming in-situ partitioner for large parallel applications, CARP, which intercepts writes from applications, transforms them, and persists them in a range query-friendly layout. A logical view of CARP's partitioning process is shown in fig. 3.3. CARP has no impact on application runtime, as it uses spare CPU and network capacity to transform data at a rate necessary to saturate storage. This partitioned output requires no post-processing and can be queried directly once the application terminates.

CARP augments conventional range partitioning with novel techniques to achieve its write and query performance goals. We build upon a simple insight — data does not need to be fully sorted to achieve the performance benefits of a sorted, clustered index. It is sufficient to partially order data in a way that allows queries to be processed via a small number of large reads. CARP exploits this relaxation to achieve its write performance and adaptivity goals.

We employ a greedy approach to generate best-effort range partitions in a single streaming pass so as to not require expensive reorganization of written data. To generate balanced partitions, CARP employs a novel primitive to construct and monitor the global key distribution. This distribution is divided into partitions with equal areas under the curve, and each partition is assigned to a rank. In the common case, all ranks shuffle data to corresponding partitions. As the workload's key distribution drifts, CARP recomputes partition boundaries and continues shuffling. Shuffled data is written to disk via a storage backend called *KoiDB* — KoiDB further optimizes CARP output to improve range query performance. Next, we describe how CARP achieves the design goals laid out in the previous section.

Write Performance. To ingest data at storage layer throughput, CARP is designed to not divert any storage bandwidth for reorganization. Even when partitions need to be recomputed, CARP does not touch previously written data — it simply records a change in partition boundaries on each rank and continues appending data. CARP also employs standard storage optimization practices such as batching small writes into large immutable units called *SSTables* and having its on-disk layout be an append-only log.

Query Performance. Efficient queries in CARP are enabled by 1) creating a highly partitioned layout, and 2) enabling efficient retrieval via large sequential writes. CARP creates one partition per application rank, automatically ensuring more data partitions as application scale increases. A selective range query only needs to retrieve a fraction of the total data from overlapping partitions. In addition, data is written in large SSTables, which can be read efficiently via large read requests. CARP merges its partially ordered SSTs at query time via a merge-sort operation, which is cheap compared to the I/O cost of retrieving data.

Adaptivity. To adapt to skewed and dynamic key distributions, we recompute CARP partitions as the incoming distribution changes. Unlike conventional databases, we do not repartition data that has already been written to disk. We simply record the change in partition boundaries and continue appending new data. Over the lifetime of a run, the range allocated to a CARP partition changes multiple times. As a corollary, data belonging to one point in the range may end up in different partitions at different times in the run. Instead of incurring write amplification to consolidate this data, we simply postpone this consolidation (also called *compaction*) to query time.

3.4 Design of CARP

We now discuss CARP's architecture, beginning with the scalable way data is routed to ranks by key (§3.4.1), the triggers that allow CARP to determine that data needs to be repartitioned (§3.4.2), the protocol that allows ranks to determine a new partitioning for the data (§3.4.3), and a reference storage backend design for logging partition data efficiently to storage (§3.4.4).

3.4.1 Communication: Scalable data flow

Fig. 3.4 shows the architecture of CARP using an example parallel application with N ranks. Application data is intercepted by CARP as a stream of records, one per each particle or cell. Each application rank owns one partition and acts as both a producer and a consumer of data. These records are routed to the corresponding partition on the basis of the key via an operation called *shuffling*. On the receiving end, the local partition is managed via a local instance of a storage backend called KoiDB (§3.4.4). This KoiDB instance is responsible for collecting data and writing it out to a per-rank on-disk log. For efficient shuffling, we leverage the scalable all-to-all communication overlay from the DeltaFS project [24], as it has been demonstrated to scale up to 131,072 ranks.

Due to the transmission delay introduced by shuffling, data written through CARP is only

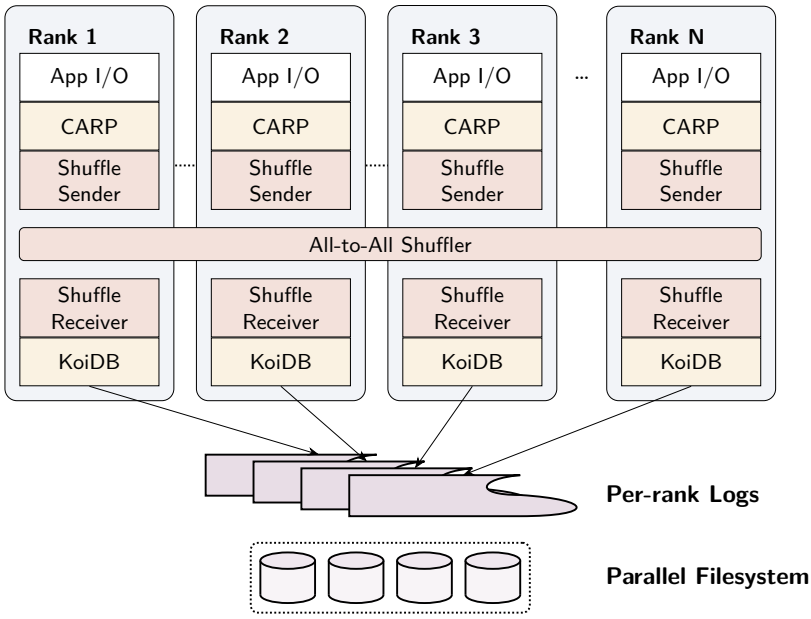


Figure 3.4: *Example of an N-rank application using CARP. Writes are intercepted and partitioned via an all-to-all shuffler. Each rank is both a sender and receiver. Partitioned data collected by receivers is stored by a local storage backend (KoiDB).*

available to query at the end of a checkpoint *epoch*. After each epoch, CARP flushes all data to the disk. By doing so, it aligns its fault-tolerance semantics with those of typical scientific applications.

3.4.2 Detection: Triggering repartitioning

CARP uses a triggering mechanism to determine if the current partitioning of data is leading the system to a state of load imbalance (i.e., routing more data to a subset of the partitions). Imbalanced loads affect the write path by creating stragglers and affect the query path when larger partitions are traversed to return query data. Figure 3.5 shows the data flow from the application to the shuffle sender layer. This is the path that CARP monitors to determine whether to trigger data repartitioning. We describe CARP's control flow through the three possible cases of data flow from the application to the shuffle sender layer below.

Common Case. In the common case CARP shuffles data as per the assigned partition ranges as described previously in §3.4.1. Collectively these ranges form a *partition table* that is replicated across all ranks. A distribution of keys transmitted by each rank is stored locally and used when it becomes necessary to participate in a global renegotiation of the partition table (§3.4.3.1).

Out-Of-Bounds. As simulations progress new keys are generated. If a rank is asked to insert a key that is currently out of the partition table bounds, then there is no valid destination for the data. In this case CARP temporarily buffers the data in an in-memory buffer on the sender called the *Out-Of-Bounds (OOB) Buffer*. Once the OOB buffer reaches a predetermined threshold, a renegotiation of the partition table is triggered.

Rebalancing Required. This trigger is only needed to adapt to distribution changes *within*

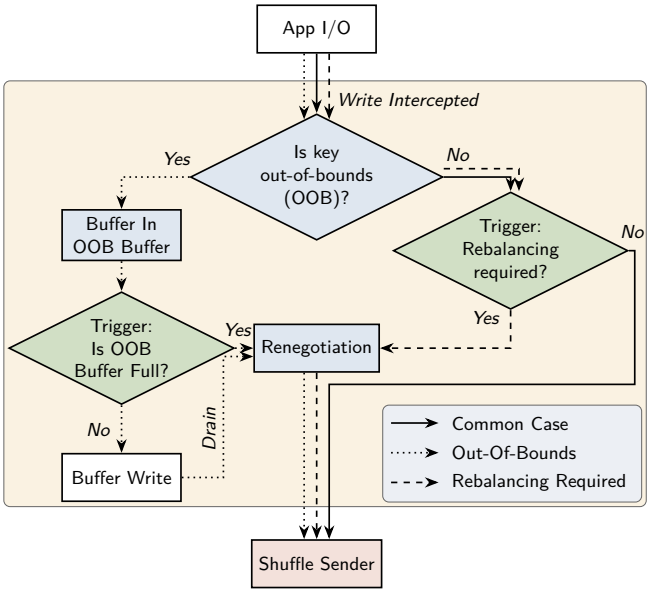


Figure 3.5: *CARP data and control flow. Application data is routed by the shuffle sender to its partition. But, if its key is outside all partition bounds, then data is added to an Out-Of-Bounds (OOB) buffer. When the OOB buffer fills up or partitions become imbalanced, CARP renegotiates the partition table.*

an epoch — for new epochs CARP bootstraps partitions from scratch. To resolve intra-epoch drift, a renegotiation should be triggered to compute a new and more relevant partition table. Instead of using background communication mechanisms to robustly detect load outliers, we have found it simpler to assume that a rebalancing is required at periodic intervals within an epoch. Some applications such as AMR (Adaptive Mesh Refinement) codes are aware of when they refine and can signal CARP for more precise control over renegotiation. We explore the impact of a fixed interval further in our evaluation.

3.4.3 Renegotiation: Determining new partitions

The goal of renegotiation is to rebalance the partition boundaries used by the shuffle layer to route data to its destination. Renegotiation replaces the partition table stored in each rank with a new more balanced one based on the latest statistics of the global distribution of the keyspace. Renegotiation is similar to distributed snapshotting algorithms such as *Chandy-Lamport* [98] but is designed to trade off some accuracy for scalability and performance. As a result, the computed global distribution is not a perfect representation of the actual distribution, but our evaluation shows that any estimation errors result in negligible imbalance in partition load. Our scalable implementation of renegotiation is described in §3.5. The steps involved in the renegotiation protocol are as follows:

1. A new round is triggered via a notification from some rank, which also pauses shuffling.
2. Local distribution estimates are computed on all ranks and then collected and merged on rank 0 to form an estimate of the global distribution.
3. A new partition table is computed and broadcast.
4. All ranks switch to the new partition table and flush their Out-Of-Bounds buffers according to the new partitions.

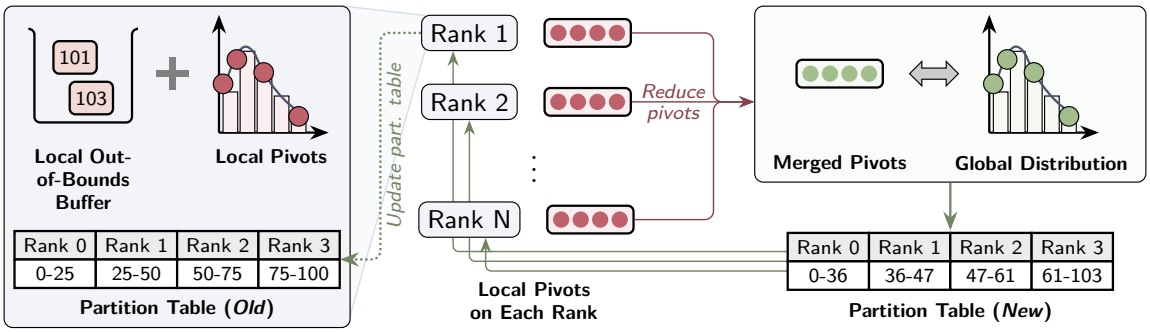


Figure 3.6: *Partition table renegotiation relies on pivots computed from histograms of keys tracked on each rank. Pivots are merged, and the new global histogram is divided into equal-mass bins before being broadcast.*

5. Ranks reset local distribution statistics and resume shuffling.

Local distribution estimates are computed and merged using summary statistics primitives. Renegotiation is initiated by trigger that can fire on any rank. We describe CARP's summary statistics primitives and trigger design next.

3.4.3.1 Summary Statistics

CARP ranks use summary statistics primitives to construct the global distribution during renegotiation. We use *histogram-based sampling* to track and aggregate ranks' local key distributions. Different quantile estimation techniques can be plugged into CARP, but we have found that histogram-based sampling is efficient to compute and allows for control over trading compactness for accuracy. CARP tracks rank-local key distributions using histograms. Each rank's histogram consists of one bin per application rank (or partition). For each processed key the corresponding bin counter is incremented. This histogram represents a lightweight, lossy representation of the local key distribution. When a renegotiation is invoked the global distribution is constructed using two primitives: histogram sampling and pivot union, as shown in fig. 3.6.

When ranks are notified of a renegotiation round, they compute *pivots* via the **histogram sampling** primitive. Pivots are a compact and lossy representation of a distribution that can be efficiently computed, communicated, and merged. Pivots are a set of k points in the keyspace that divide the histogram into k partitions of equal mass, i.e., bins containing the same number of samples. Increasing k reduces representation lossiness but requires more network communication for the additional information. For skewed distributions, this results in more pivots concentrated in histogram regions where bin counts are high and vice versa. Pivots are calculated by linear interpolation between bin boundaries. Keys in local OOB buffers are also factored into pivot computation.

Pivots from all ranks are collected at a designated rank (Rank 0) and aggregated to form pivots representing the global distribution via the **pivot union** primitive. This operation uses the information captured in rank-local pivots to construct a global distribution. This global

distribution is then resampled to compute the new partition table that is broadcast to all ranks.

3.4.3.2 Triggers

Renegotiation triggers are evaluated at each rank to determine when to renegotiate (see fig. 3.5).

The **Out-of-Bounds trigger** is used to extend partition boundaries when many keys outside the partition table boundaries are encountered. We partition the known keyspace without any gaps, so an invocation of this trigger strictly extends the allocated keyspace. An in-memory *Out-of-Bounds* (OOB) buffer is used on each rank to temporarily store keys that are currently outside the bounds of the current partition table. The OOB buffer has a predetermined size and when it fills up a renegotiation is triggered. The contents of all ranks' OOB buffers are factored into the new partition table so that the newly computed table has destinations for those keys. Keys in the OOB buffers are flushed to their respective destinations once the renegotiation completes. We have found OOB buffers with a capacity of 512–1024 items per rank sufficiently effective.

This trigger is also used to bootstrap CARP at the beginning of each epoch. As there is no partition table when an epoch begins, all ranks collect writes into their OOB buffers, and invoke a renegotiation to find the first partition table to start shuffling with.

The **Rebalancing trigger** is used to account for key distribution drift over time causing partition load to become imbalanced. In principle this trigger could be designed to fire only when key distribution drift is detected, however we have found that invoking it periodically is both simpler and effective. We found frequent fixed-interval renegotiation to provide effective partitioning without having a measurable runtime overhead (§3.6.3.4).

3.4.4 Storage: Logging for efficiency

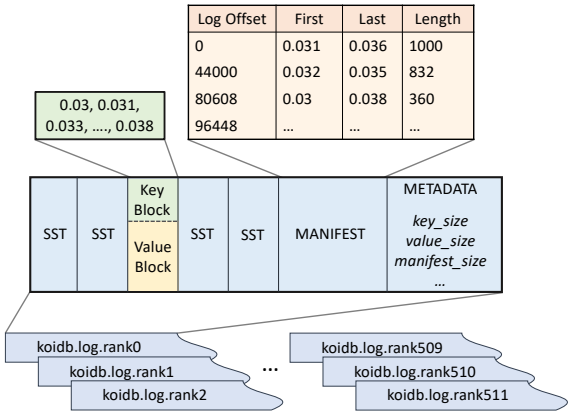


Figure 3.7: *KoiDB on-disk log format. Each log consists of SSTables organized into contiguous key and value blocks. Each SST has an entry in the manifest along with metadata.*

KoiDB is our reference implementation of CARP's storage backend. Its output is written to a parallel file system and can be directly queried by a query client on a single node. As query clients access files in read-only mode, multiple concurrent query clients are automatically supported. KoiDB collects data from the shuffle receiver and writes it to an append-only log on shared

storage. KoiDB instances are local to each rank and operate independently of each other. We discuss using CARP with other storage backends in §3.7.

fig. 3.7 shows KoiDB's on-disk log format. KoiDB produces this output by first collecting shuffled records in a memory buffer. When the buffer fills, KoiDB compacts the data into an *SSTable* (or SST) that is then appended to the log. Compaction operations include (optionally) sorting the contents by key, serializing the keys and values into separate sub-blocks for more efficient query-time parsing, and writing the serialized SST to storage. A manifest entry containing the SST's key range and location is also written into a manifest block that is used to find relevant SSTs for queries. Storing this manifest results in a small space amplification (~.01%). KoiDB uses double-buffering to allow compaction to run in the background while shuffling continues in the foreground.

We now describe two additional optimizations KoiDB applies to SSTs for query performance:

Repartitioning to refine SSTs. CARP partitioning quality can be degraded by *stray keys* created when CARP partition tables are updated. These keys end up being misdelivered because their correct shuffle destination changes between dispatch and delivery due to renegotiation. Letting them be written to the wrong partition reduces SST selectivity and will increase query latency as more SSTs need to be read for each query. One way to avoid stray keys is by flushing the network before each renegotiation, but this increases the cost of invoking the primitive. KoiDB mitigates this by simultaneously maintaining multiple open SSTs and separating stray keys into a different SST. As we show in §3.6.3.3, this improves the selectivity of CARP partitions by up to 48×.

Subpartitioning to create smaller SSTs. To further reduce the read amplification for highly selective queries, KoiDB can subdivide a rank's SSTs into smaller ranges before writing them out. Smaller SSTs can be retrieved faster, but a large number of subpartitions can also increase CARP's CPU and metadata overhead and adversely affect runtime.

3.5 Implementation

In this section we describe CARP's implementation and its relationship to scalability, memory footprint, and runtime overhead.

CARP [25] consists of ~10,000 lines of C/C++ code. Application writes are intercepted using a dynamically preloaded shared library. The CARP API can also be called directly without using a preload library. CARP uses two instances of the DeltaFS shuffle service [99] for communications: the *data* instance batches messages into 32 KB buffers for high throughput, while the *control* instance is used for renegotiation control messages and uses no batching for low latency. The shuffle service is built using the Mochi [100] Mercury RPC library [101]. We use RPCs as both shuffling and renegotiation are asynchronous operations better suited to RPC semantics (it also allows us to avoid complications with multiple concurrent MPI communicators). CARP creates its own threads for network communication, RPC callbacks, and asynchronous compaction in KoiDB. This setup allows straggler ranks to use multiple cores and provides some tolerance for load imbalance.

Scalability. CARP is designed to scale by composing scalable operations — the shuffle service has been demonstrated to scale to 131,072 ranks [24], and the renegotiation protocol is designed as a reduction operation with a logarithmic scaling factor (evaluated in §3.6.3.1). Each instance of CARP's storage backend (KoiDB) is an independent instance and can scale trivially. By default, CARP creates one file per rank, but at larger scales, the total number of files can be reduced (if needed) by having a subset of ranks be shuffle receivers.

We call CARP's scalable implementation of renegotiation the *Tree-based Renegotiation Protocol (TRP)*. A naive implementation of renegotiation, as described in §3.4.3, would require directly collecting all ranks' pivots on one rank before computing the new partition table. Such an implementation will have limited scalability, as its memory and network communication footprint will scale linearly with ranks. Our TRP implementation, on the other hand, uses a reduction tree-based design [102, 103] which scales logarithmically.

TRP is built on the observation that pivot unions (§3.4.3.1), being associative and commutative have all the properties of a lossy reduction operation. Pivots represent distributions and can be merged in any order, but the pivots lose some information in the process. To limit the impact of this lossiness, TRP employs a shallow tree hierarchy with a large fan-out (depth of 3, fan-out of up to 64). The tree's leaf layer consists of all CARP ranks, while intermediate layers consist of equally spaced ranks. Subsets of pivots are reduced on intermediate layers, and a final reduction happens at the root.

Memory Footprint. As CARP runs in application processes, it needs to have a low memory footprint to avoid competing with the application for memory. CARP achieves this by streaming application data directly to the shuffle pipeline. The primary source of memory overhead is due to buffers used to batch I/O for network and storage efficiency. To illustrate this overhead, we use a run with 4096 ranks and default CARP parameters that results in 27 MB per rank[1]. This is less than 1% of the per-core memory budget on the LANL Trinity supercomputer [104].

Runtime Overhead. CARP only runs during application I/O, so it does not impact the compute phase. Data shuffling and triggered renegotiations are potential sources of additional runtime overhead. Shuffle bandwidth increases with the number of writes and quickly outstrips storage bandwidth in our experiments. Thus, the shuffle service itself does not have any runtime overhead. CARP could underutilize storage by pausing I/O for renegotiation, but our experiments show that shuffle receivers buffer enough outstanding writes to keep storage busy and mask this overhead. If needed, CARP can continue routing data using the old partition table while a renegotiation is underway, however we have not found this necessary in our experiments.

3.6 Evaluation

Baselines. We use DeltaFS [24], FastQuery [89], and TritonSort [23] as baselines representing the state of the art from different research communities. DeltaFS represents the state of the art

[1]Each rank uses 2 MB for shuffle RPC buffers [24], 24 MB for two KoiDB memtables, 16 KB for 4096*4B partition table entries, 16 KB for partition shuffle counts, and 32 KB for 512-entry OOB buffer with 64B records.

for write-efficient online hash-partitioning, FastQuery employs auxiliary indices, and TritonSort performs data reordering.

Benchmark workloads. We evaluate the performance of CARP using traces collected from a 512-rank VPIC simulation [84], a popular particle simulation code described in §3.2, and YCSB [105], a standard benchmark suite with varying key-value workloads. To eliminate variability across simulation runs and ensure reproducibility, we replay traces captured from VPIC rather than using it as a streaming workload. We use the computational plasma trace described in section 3.2 for all our experiments. We index VPIC traces using energy as the key. This results in a 4 byte floating-point key and a 56 byte record for the rest of the particle data.

Experimental setup. We use a 32-node compute cluster with each node having two 8-core Intel Xeon E5-2670 CPUs and 64 GB DRAM. The nodes are connected via 40 Gbps Infiniband QDR. Simulation output is written to a shared Lustre storage cluster consisting of 20 nodes identical to those in the compute cluster, each having a 240 GiB SSD. DeltaFS, FastQuery, and CARP experiments use the compute cluster, while TritonSort runs directly on the Lustre nodes. TritonSort has identical storage resources as other approaches and is able to achieve higher throughput out of storage by not incurring the overhead of Lustre's coordination. While TritonSort appears to have lower compute resources, all applications are bottlenecked by the storage bandwidth and their runtime is dictated by their write amplification. FastQuery provides a number of tunable parameters, and we report the numbers from the best performing parameters on our cluster.

The rest of this section is organized as follows: In §3.6.1 we evaluate CARP's query performance. We evaluate CARP's runtime overhead and compare it with existing approaches in §3.6.2. Finally, in §3.6.3, we show selected results of a sensitivity analysis aimed at understanding the impact of tunable parameters and optimizations on CARP's runtime overhead, load balancing effectiveness, and index quality.

3.6.1 Query Performance

In this subsection we evaluate CARP's partitioned output using two aspects: query latency and read amplification. Query latency is a function of how much data the index needs to read and how efficient the storage layout is in serving that data. For CARP, minimum effective query selectivity is capped by the size of one CARP partition — 0.18% (or $\frac{1}{512}$) for 512 ranks. This percentage decreases with scale as the number of partitions increases and when subpartitioning is enabled. We now discuss CARP's query performance over two query suites: one constructed by us, and a YCSB query suite.

Observation 1: *CARP can serve most queries as efficiently as a fully sorted layout and 1-2 orders of magnitude faster than state of the art auxiliary indexing approaches.*

Query Suite. For query latency experiments we index 12 timesteps from the simulation. Each timestep is 188 GB of data, and the total simulation data is 2.2 TB. Query latencies for

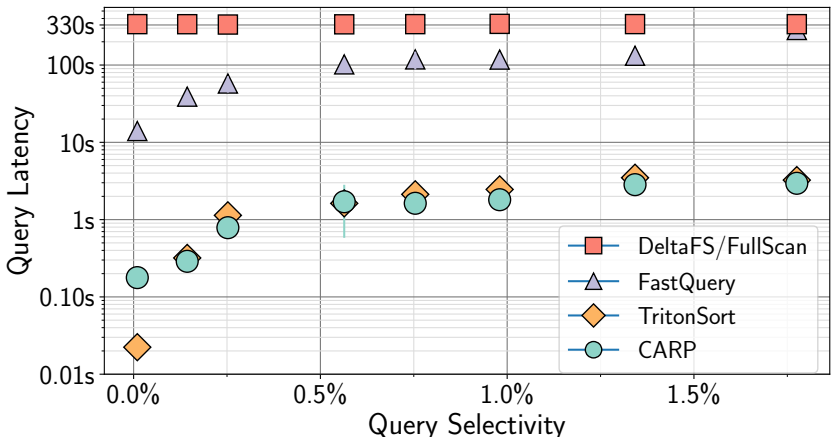


Figure 3.8: *Query Latency. CARP matches the query latency of TritonSort's fully-sorted data layout and outperforms both FastQuery by 100×.*

eight queries with different selectivity are shown in fig. 3.8 (each query is repeated 3 times and averaged). We compare the query latency for CARP's partitioned ordered output with that of FastQuery and a sorted, clustered index. We also provide numbers for a *full scan* over unindexed data for reference. The sorted output uses 12 MB SSTs and a manifest with one entry per SST, a format similar to KoiDB. We refer to the sorted, clustered index as TritonSort for convenience, but all sorts generate identical outputs. Queries for both CARP and TritonSort are processed by the same query client. The manifest is read first to find corresponding SSTs and then key blocks of corresponding SSTs are read in parallel. CARP SSTs overlap and must be merge-sorted for ordered range query semantics. The latency numbers for CARP include this sorting cost. The query client for CARP and TritonSort is run on a single compute node with access to the Lustre filesystem. FastQuery numbers are measured using its own query client reading from its natively supported HDF5 format. All query latency numbers include the time taken to fetch keys matching a query from a storage cluster to a query client and the compute time to subsequently process/filter the data fetched.

For both CARP and TritonSort, query latency appears to be linearly proportional to the amount of data read. Despite incurring an additional sorting cost, CARP's total query latency is similar to TritonSort. CARP is slower for highly selective queries (177 ms vs 22 ms for a query with 0.01% selectivity) because it is forced to read full partitions, but it is able to match (and even outperform) TritonSort for larger query ranges with selectivity $> 0.05\%$. FastQuery is much slower than either of the two approaches. This is partially due to it needing to read large bitmap indices, but largely due to it being an auxiliary index, requiring small, random reads.

Observation 2: *Despite incurring extra sorting overhead, CARP matches TritonSort's query latency performance for all except extremely selective queries ($< 0.05\%$ selectivity).*

YCSB Benchmark Suite. To thoroughly evaluate the query performance of CARP's approximate partitioning we use Workload E from YCSB (fig. 3.9). Workload E consists of short range queries and inserts in a 19:1 ratio. The range queries consist of up to 100 keys generated from a Zipfian distribution and are reordered by a random hash function. We drop inserts from our benchmark suite as CARP and TritonSort are not online stores but are instead transient

services with different insert and query pathways. We define the workload's ranges in terms of fully ordered SSTs. We interpret YCSB query ranges as SST numbers in TritonSort's output and translate them into equivalent key ranges that we use for both CARP and TritonSort. The range SST# $(251, 350)$ would therefore query the key range stored in TritonSort SSTs numbered 251 to 350. Since KoiDB SSTs are smaller and more fragmented, the same query with CARP would result in more SSTs read.

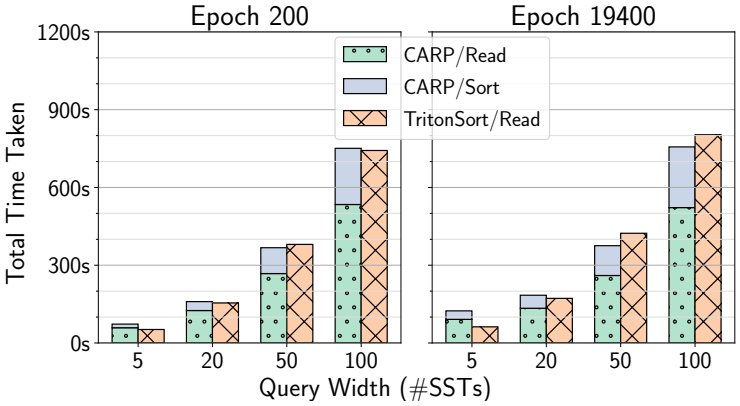


Figure 3.9: *Time taken for a YCSB query suite, demonstrating similar trends to the left-hand figure. CARP is slower for highly selective queries but comparable/better for larger queries despite the sorting overhead.*

We construct 4 query batches of different fixed widths (5, 20, 50, and 100 SSTs) corresponding to query selectivity of 0.03%–0.6%. The starting index is drawn from YCSB's Zipfian distribution in the range SST# (0, 18266), where 18266 is the number of SSTs in each timestep. We use 16 I/O threads to fetch SSTs for each query in parallel. For each width, we construct a batch of 1000 queries per timestep with the order randomized by YCSB's hash function (`fnvhash`). fig. 3.9 compares the total time taken by a batch of queries for two different timesteps and 4 query widths for each timestep, and it reaffirms the trends from fig. 3.8. CARP's median selectivity of 0.07% (with 4-way KoiDB subpartitioning enabled) enables us to be competitive for queries with a width of 20 SSTs and greater. For larger queries, the cost of sorting starts to become a greater proportion of CARP's query latency, but the aggregate latency is still close.

A surprising takeaway from these latency experiments is that CARP's partial ordering seems to result in the most efficient storage layout. CARP reads are faster than the fully sorted layout, and query performance is similar despite us incurring the extra overhead of merging SSTs. We attribute this to our layout being amenable to parallel reads from different storage nodes of a parallel filesystem — it has enough contiguity to be read efficiently vs small random I/Os, but is distributed enough to allow for parallel processing of a query.

3.6.2 Write Path

Observation 3: *CARP is 2.8–4.9× faster than post-processing approaches with no overhead on application performance.*

Fig. 3.10 measures the effective I/O throughput of four different index building approaches ingesting 188 GB of VPIC data (corresponding to 3.5B particles or one timestep at 512 ranks).

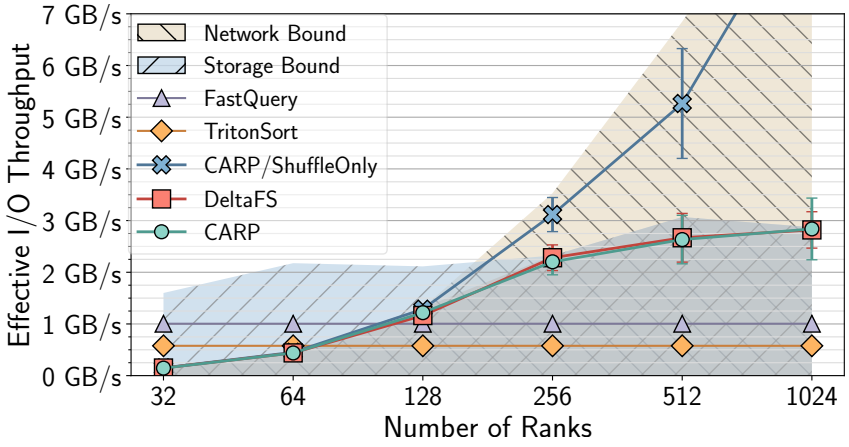


Figure 3.10: *I/O Throughput. CARP incurs no overhead over unpartitioned I/O and is 3-5× faster than post-processing.*

The effective I/O throughput is computed by dividing the application data volume by the total runtime. For in-situ approaches, their runtime overhead is included in the application runtime, while for post-processing approaches, it is obtained by adding the post-processing time. Each run is repeated 9 times and averaged.

To measure I/O throughput, we keep the total amount of data generated constant across different scales as the write performance of SSDs decreases as they fill up. Allowing the amount of data written to scale up with the number of ranks results in higher scales unfairly getting a lower storage throughput. The achievable storage bandwidth from our cluster (*Storage Bound* in fig. 3.10) increases with the number of application ranks: from 1.6 GB/s at 32 ranks to saturating the storage with 3 GB/s at 512 ranks. At 1024 ranks, the increased contention from a large number of parallel writers causes a small dip in achievable throughput. For 1024 ranks, we divide the 512-rank trace into two halves to keep the total data constant.

We compare CARP with DeltaFS, FastQuery, and TritonSort. CARP and DeltaFS are embedded within the application and hence are evaluated at different scales. FastQuery and TritonSort are post-processing approaches and are run after VPIC completes. For the post-processing approaches, effective throughput is obtained by dividing the total data written by the total time taken (VPIC time and post-processing time). For the in-situ approaches VPIC time includes the online index building time.

FastQuery and TritonSort. As shown in fig. 3.10, the additional time taken by post-processing approaches (FastQuery and TritonSort) significantly reduces the effective application bandwidth from 3 GB/s to ~1 GB/s and ~0.6 GB/s respectively. This represents a slowdown of 2.8× for FastQuery, and 4.9× for TritonSort. TritonSort incurs a much larger slowdown as it creates a clustered index, which requires four passes over all data for out-of-core sorts. FastQuery scans the data once, creates index structures, and writes them to storage, but takes an additional 24% space to store indexes for a single attribute.

DeltaFS. As discussed in §3.1, DeltaFS embeds with VPIC and reorganizes data via an all-to-all shuffle while it is in transit from the application to storage. DeltaFS uses the same 3-hop shuffle used by CARP (§3.4.1). DeltaFS uses a hash of the particle ID to partition incoming

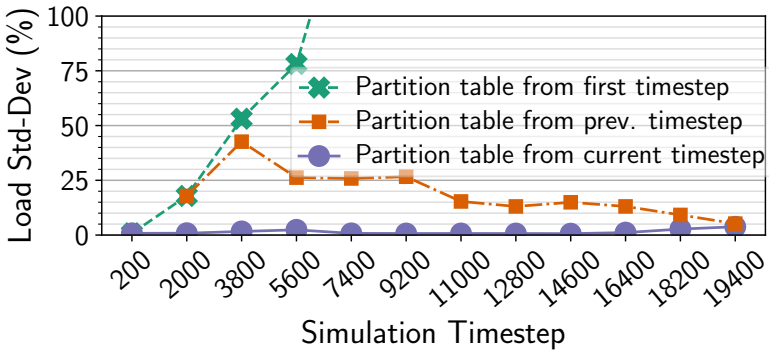


Figure 3.11: *Simulated performance of different static partitioning schemes in generating balanced partitions (lower load std-dev implies better partition balance). Reusing a partition table from the first timestep results in the worst load-balance. Tables from the previous timestep fare better, but perform poorly when the simulation is the most active. Tables from the current timestep fit the best (true by definition).*

data. At smaller scales, DeltaFS is bound by the available shuffle throughput (*Network Bound* in fig. 3.10). As the aggregate shuffle throughput increases with scale, DeltaFS is able to fully saturate the storage layer, incurring no overhead over raw VPIC throughput.

CARP. We measure CARP network overhead (*CARP/ShuffleOnly*) and end-to-end performance (*CARP*) separately in fig. 3.10. For CARP/ShuffleOnly, we drop data once it is received at a shuffle receiver so as to not be bound by storage performance. We see that CARP/ShuffleOnly scales with the available shuffle bandwidth, incurring a small overhead for renegotiation rounds and the residual load imbalance. However, this overhead does not matter for the end-to-end performance as CARP quickly becomes bound by the available storage bandwidth. When the network bandwidth exceeds storage bandwidth, CARP has no impact on end-to-end performance. By avoiding post-processing, CARP enables indexes to be built 2.8–4.9× faster.

Observation 4: *Static partitioning can quickly devolve to reading orders of magnitude more data than CARP by not adapting to key distribution drift.*

Fig. 3.11 is a study of how frequently partition tables need to be recomputed for our VPIC trace to generate balanced partitions. We construct partitions from a perfect knowledge of different simulation timesteps and measure how well they fit subsequent timesteps. The green line simulates a static partitioning approach where partitions are computed from a perfect knowledge of the first timestep but are not changed afterwards. The load balance of a static partition scheme worsens as the key distribution drifts over time. Next, we measure how well a partition table from the previous timesteps fits the current timestep. This works better but still demonstrates a significant load imbalance around timestep 3800 when simulation entropy is high. As the simulation converges and the entropy between adjacent timesteps reduces, reusing older partition tables fares better. Finally, we see that partition tables obtained from the current timestep fit that timestep very well. This is true by definition, and the minor load imbalance arises from the lossiness of the histogram approach to capture distributions.

We conclude that any partitioning approach should recompute its tables at least once every timestep, more if there is intra-timestep entropy. We emphasize that the purple line in fig. 3.11 represents an upper bound on CARP's load balance, as the partitions used for this benchmark are *oracle partitions*. Since CARP partitions data as it arrives, it does not have the required information to compute these.

3.6.3 Sensitivity Analyses and Microbenchmarks

3.6.3.1 Renegotiation Protocol Scalability

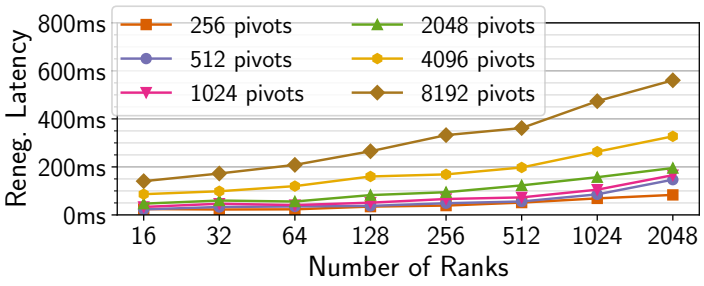


Figure 3.12: *Renegotiation scalability at different pivot counts. Renegotiation scales logarithmically with the number of ranks and takes longer if more pivots are exchanged.*

Fig. 3.12 shows the time taken by a single renegotiation round across different scales and pivot counts. As described in §3.5, we implement renegotiation using a reduction tree and hence expect it to scale logarithmically. Fig. 3.12 shows the logarithmic scalability of our implementation from 16 to 2048 ranks for 6 different values of the pivot count parameter. Increasing the number of pivots computed increases the size of the messages exchanged. This results in proportionally higher round latency. 512 pivots are sufficient for CARP to accurately track distributions (discussed below), and the latency of a single round is only 150 ms even at 2048 ranks.

The absolute values of renegotiation latency are handicapped by us using an emulated network stack (IPoIB) for a fair comparison with baseline approaches that use sockets. We expect these numbers to be much lower if run natively on a modern interconnect.

3.6.3.2 Impact of Pivot Count

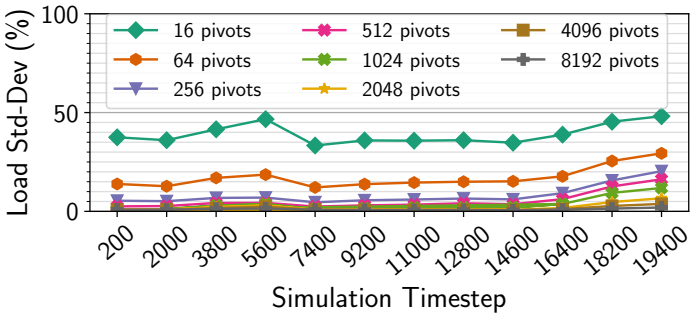


Figure 3.13: *Pivot Count vs Histogram Fidelity. 512 pivots enable reasonably accurate reconstruction for even highly skewed distributions, with diminishing returns after.*

Fig. 3.13 shows the results from a micro-benchmark that tests the lossiness of our pivot calculation scheme. Pivots are an approximate representation of a distribution. The higher the

pivot count, the better the approximation. We compute oracle pivots from a full key distribution of each of the 12 different timesteps for different pivot counts and check how well the partition table calculated from those pivots fits that timestep's keys. We use standard deviation in partition load (`std-dev`) as a proxy for the lossiness of the pivot calculation. A lossless scheme would result in perfect partitions, resulting in zero `std-dev`. Higher `std-dev` implies more lossiness.

In fig. 3.13 we see that higher pivot counts lead to lower load imbalance with diminishing returns beyond 256 pivots. We also see that the last two timesteps are harder to reconstruct than the others. This is because of the extremely skewed nature of those timesteps' distributions (figs. 3.1 and 3.2). While all VPIC timesteps have long tails, more pivots help when they become extremely long as the simulation progresses. Sketch-based quantile estimation methods can provide more robustness for such distributions [106].

3.6.3.3 Impact of KoiDB

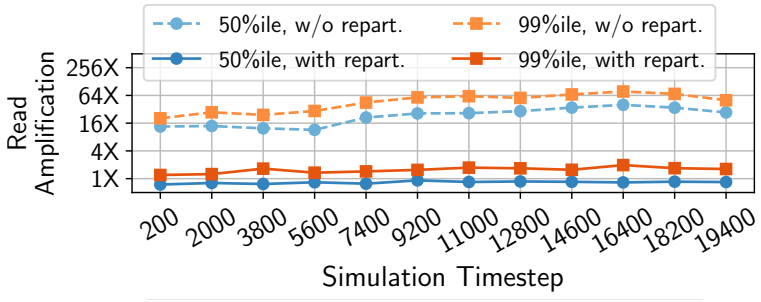


Figure 3.14: *Impact of repartitioning on 50th and 99th percentile RAF. Both median and tail RAFs are significantly lowered by KoiDB partitioning, from 16-64× to 1-2×, across all timesteps.*

We now summarize how KoiDB (§3.4.4) improves partition quality. We introduce a measure called *Read Amplification (RA)* to measure CARP partition quality. We define read amplification as the ratio of the size of actual CARP partitions vs (hypothetical) perfectly balanced partitions. An RA of 1× for all partitions is ideal, as balanced partitions are better on both the write and the query paths. It is also unachievable as CARP tries to predict future key distributions and partition in a single pass, which is inevitably an imperfect operation.

We find that repartitioning, by separating out stray keys, reduces the average RA by up to 48× (fig. 3.14). For *subpartitioning*, we find that 2-way and 4-way subpartitioning improve average latencies for highly selective queries by 28% and 43% respectively with no observable runtime overhead. We omit detailed results for subpartitioning as we have observed its impact to be largely predictable, as described in §3.4.4.

3.6.3.4 Tuning CARP

We now summarize key takeaways on the performance impact of CARP's two main tunable parameters: renegotiation frequency and pivot counts. We vary renegotiation frequencies between 2×/epoch to 26×/epoch and pivot count from 64 to 2048. We then measure CARP's partition load

balance and application runtime performance. Note that it is important to optimize partition imbalance even if it does not affect runtime to provide more predictable and uniform query performance.

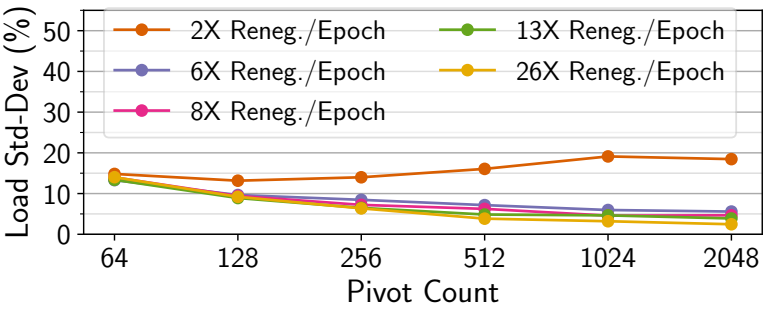


Figure 3.15: *Impact of CARP parameters on partition load balance. Renegotiating multiple times within a timestep is necessary for load-balance, but provides marginal benefit beyond a point. More pivot counts produce a better load balance, with diminishing returns.*

We find that these parameters have a noticeable impact on partition load balance but not on CARP's shuffling runtime (fig. 3.15). At the lowest values of these parameters (64 pivots and 2× renegotiations per epoch), CARP partitions have a normalized standard deviation of 14%. At the highest values (2048 epochs and 26×/epoch) this drops down to 2%. More pivots provide noticeable load balance improvement up to 512 pivots and then returns diminish. It is beneficial to increase the renegotiation intervals from 2×/epoch to 6×/epoch, but we get minimal gains beyond that interval.

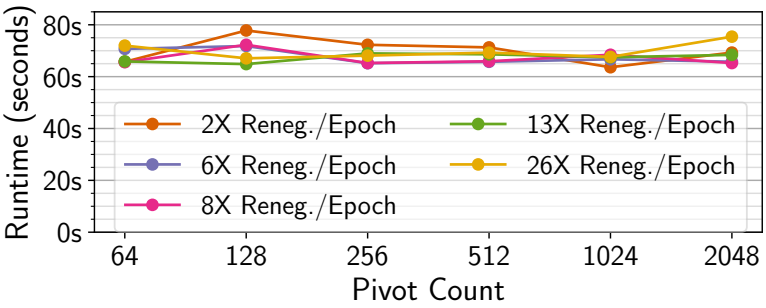


Figure 3.16: *Impact of CARP parameters on write performance. Up to 1024 pivots, increasing the pivot count leads marginal improvement in runtime. Renegotiating frequently does not penalize runtime — the cost of frequent renegotiations is made up by the gains in load balance.*

Surprisingly, none of these parameters seem to impact runtime in any measurable way (fig. 3.16). We attribute this to CARP's imbalance-tolerant implementation (idle ranks can cede CPU time to straggling ranks, §3.5).

Tuning Conclusion. While CARP can be tuned for incremental performance benefits, it provides stable performance across a wide range of parameters. Even with the worst performing parameters, CARP partitions only deviate by 14% on average. CARP will still provide excellent query performance for queries not reading from larger partitions. A moderate number of pivots (~512) can represent even extremely skewed distributions with long tails, and renegotiation frequency has marginal impact on partition quality as it only addresses intra-epoch drift.

3.7 Discussion: Extensibility and Applicability

We now discuss how CARP can be adapted for different query types, analyses, and architectures.

Multi-attribute Queries. In this paper, we focus on using CARP to build a sorted, clustered index on a preferred attribute. For applications such as VPIC, indexing on a single attribute is sufficient to identify and retrieve points of interest (high-energy particles), and apply subsequent transformations in memory. However, CARP can be extended to build sorted, auxiliary attributes on additional attributes. This would require CARP to track distributions of all configured attributes and use a two-stage shuffling pipeline as described below:

- First, the entire row is shuffled using the primary attribute as usual. The receiver serializes all rows, assigns them a unique `row_id`, and writes them to the local storage backend. Each receiver then computes a `(key, partition_id, row_id)` tuple for each additional indexed attribute and shuffles it via the same pipeline.
- Each receiver of the auxiliary attribute tuples writes them to separate storage backend instances, where each row points to the full row on the primary key partition.

The additional attributes will not have the query performance of the primary attribute but will still benefit from the space efficiency and index lookup performance of sorted indices (vs bitmap indexes).

Applications and architectures. CARP can be used to ingest data from any parallel application where storage bandwidth is the bottleneck. The assumptions we make are typical of such analysis workflows:

1. Data is not modified once ingested.
2. Spare network bandwidth is available for shuffling. This is true by definition for any storage-bound workflow and holds for most HPC clusters. The Aurora cluster [107] at ANL, for example, has a bisection bandwidth of 690 TB/s, more than 20× the storage bandwidth (31 TB/s).
3. Write performance is paramount, and the goal is to provide acceptable query performance without compromising write performance. Where the goal is optimal query performance (as with a web service), a distributed database is more appropriate.

3.8 Related Work

Data structures such as LSM-Trees [108], B+ Trees [109], and their variants [110] are employed by online databases to order data. These data structures heavily reorganize data internally and are more inefficient at writes than bulk post-processing approaches [97]. Multiple in-situ analysis frameworks have been proposed including PreDatA [111], GLEAN [112, 113], NESSIE [114], DataSpaces [67], and ADIOS [115]. These systems are designed such that auxiliary nodes are used to perform analysis tasks. ADIOS supports in-situ generation of range query indices using FastBit [116] (the same bitmap index used by FastQuery) using auxiliary nodes. In-situ generation of bitmap indices would be faster than post-processing via FastQuery, but at the cost of dedicated resources, and the space overhead and query performance limitations of bitmap indices would still remain.

Usher et al. [117] describes an in-situ indexing system that aggregates application data into spatial locality-preserving data layouts. Their approach requires provisioning dedicated aggregator nodes which collect spatially-partitioned particle data from the application, add a bitmap index to the data, and write it to storage. Particle codes already have spatial partitions because of how they decompose the problem. This system only preserves these pre-existing partitions. CARP is able to create partitioning along arbitrary dimensions using all-to-all shuffling. Further, CARP does not require dedicated resources, introduces more powerful distribution monitoring constructs, overlaps indexing and I/O for extremely high storage utilization, and confers maximum benefits of an in-situ partitioning approach. Since CARP does not require any dedicated resources, these two approaches can be composed together for richer partitioning capabilities.

Slalom [118] and VETI [119] describe raw-data indexing approaches. Raw data indexing works on the query path to exploit pre-existing order in data and adaptively reorder data in response to query patterns. Scalable variants of these techniques can complement approximate indexes on the write-path to further optimize query performance for exploratory queries, but they are not a substitute for in-situ indexing on the write path. WiscKey [120] introduces the notion of separating keys and values in LSM-Trees for lower write amplification, similar to our discussion on clustered vs auxiliary indexes. SuRF [121] can help reduce redundant reads for range queries with sparse keyspaces but can not help speed up queries for data with no keyspace gaps.

3.9 Lessons for Scientific Data Infrastructure

CARP demonstrates that adaptive range partitioning, driven by periodic reconstruction of global histograms, can effectively range-partition scientific data in situ. The resulting layout is partially ordered, so range queries may require some residual compaction at query time. Our evaluation shows that this is the right tradeoff for scientific data: differences in query performance are imperceptible for analytical use cases, while ingestion performance is maximized. Two lessons recur across the AMR study discussed next and inform ORCA's design: the applicability of OLAP techniques to scientific data management, and the limits of fixed-function feedback loops.

Applicability of OLAP techniques. While hash and range-partitioning are standard database techniques, their application to scientific data management is shaped by the environment. These pipelines terminate in parallel filesystems [51, 122] designed to absorb massive, synchronized ingestion from tens of thousands of writers. As a result, these techniques must be embedded into in-situ pipelines that reorganize data in flight to parallel filesystems.

Beyond techniques, OLAP formats also remain relevant for scientific data, at least insofar as data fits the relational model. CARP evolves a custom storage format derived from DeltaFS's [24] flattened LSM tree: disjoint SSTables with contiguous key and value blocks for efficient sequential reads, with per-SSTable statistics maintained in a catalog for query-time pruning. This is essentially identical to Parquet [41] and open table formats like Iceberg [123]. Queryability over other attributes is a natural concern automatically remedied by such formats and planning-based query engines. We revisit this topic in §6.3.

Limitations of fixed-function loops. CARP's feedback loop is fixed-function: it can materialize one view via one reduction type, triggered by a static policy. Additional views may be desirable not only for other workload types but for better range partitioning itself. Sketch-based quantile estimates [106] may provide better distribution fidelity than histograms. Decisions such as tuning renegotiation intervals and pivot counts would benefit from runtime visibility into how well the partitioning is performing—visibility that CARP's own loop cannot provide.

Chapter 4

Lessons from Optimizing Placement in Adaptive Mesh Codes

Contents

The previous chapter demonstrated a feedback loop that works: CARP reconstructs a global view of an evolving key distribution and adapts range partitioning online. This chapter examines what it costs when the corresponding loop is high-latency, using our experience developing telemetry-driven placement policies for block-structured adaptive mesh refinement (AMR) codes as a case study. AMR accelerates simulations by refining the mesh only where needed, but that

adaptivity produces heterogeneous per-block work and shifting communication patterns [26, 124]. At scale, frequent synchronization amplifies this variability into stragglers, so placement and load balancing become first-order performance concerns [26, 27]. Careful end-to-end characterization of these effects in production AMR codes is sparse. The best available evidence from exported traces of large MPI applications [72] reports substantial waiting time at scale and emphasizes that standard trace data is often insufficient for cross-stack diagnosis, leaving algorithmic and infrastructural causes entangled. We therefore study real AMR codes on our own research cluster with full-stack access, collect our own telemetry, and evaluate changes empirically.

Our goal was to replace static placement heuristics with telemetry-driven workload models, but initial experiments were dominated by effects that had nothing to do with placement. Fail-slow hardware [12], fabric pathologies, MPI runtime interactions, and cross-layer tuning issues produced straggler signatures that could not be distinguished from genuine placement effects until they were identified and removed. Localizing these behaviors required fine-grained telemetry that, in practice, was only accessible through traces. Standard tracing tools [28, 30] collected this data as per-rank logs, but the resulting volume was hard to control across thousands of timesteps, and the fixed schemas did not capture the custom fields we needed to interpret AMR behavior. We therefore iterated through a series of bespoke instrumentation and analysis pipelines, repeatedly adjusting what to emit so that post-hoc analysis remained tractable. The result was a slow, manual diagnostic cycle in which time was dominated by collecting, moving, and reshaping telemetry, even when the eventual remediation was simple (for example, pruning bad nodes, correcting configuration, or fixing an obvious tuning mistake).

With a reliable measurement baseline established, we could finally isolate placement effects. Validated telemetry confirmed that synchronization dominated runtime (35–50% of total time, increasing with scale), driven by compute variability across blocks. We found that compute load balance and communication locality are competing objectives: improving one degrades the other. We developed CPLX (§4.4), a hybrid placement policy that exposes this tradeoff through a single tunable parameter X. At $X{=}0$, CPLX prioritizes locality; at $X{=}100$, it reduces to pure load balancing. On a real AMR code across 512–4096 CPU ranks, CPLX improves runtime by up to 21.6% over tuned baselines (§4.5), with the best setting of X varying with the code, problem, scale, and cluster.

The entire exercise—from diagnosing thermal throttling to evaluating CPLX tradeoffs—was one continuous cycle of hypothesis-driven telemetry work. There was no clean boundary between denoising and placement. Every intervention required re-collecting data, re-analyzing it, and re-validating that the change produced the effect we believed it did. The cost was establishing measurement fidelity, attributing variability to root causes across the stack, and iterating toward a reliable baseline from which placement effects could be studied. Developing and evaluating CPLX required the same pattern, including custom instrumentation for per-block costs, post-hoc analysis, and repeated checks that apparent improvements reflected placement effects rather than residual noise in the measurement and execution environment. Every finding was specific to one

code on one cluster, and the conditions that made the process expensive—including cross-stack confounders, context-dependent tradeoffs, and reshaping telemetry for each new question—are structural properties of large-scale parallel systems. Because the dominant effects and the best tradeoffs depend on the code, problem, scale, and environment, translating a one-off win into a repeatable practice requires repeating much of the same process.

This chapter traces that process end to end: the confounders we encountered and how we diagnosed them (§4.2), the evolution of our telemetry and analysis workflow (§4.3), and the design and evaluation of CPLX (§4.4, §4.5). It concludes with lessons for BSP diagnostic workflows drawn from this process in §4.7.

4.1 Why Telemetry-driven Placement?

Block-structured AMR simulations running on O(100K) cores face significant performance challenges due to computational variability [26, 125, 126]. Frequent mesh refinement necessitates periodic redistribution of work across compute resources. Stragglers, if left unaddressed, are particularly expensive due to frequent global synchronization operations—a single straggler can block the entire simulation at synchronization points, leading to poor cluster utilization [10].

Two complementary approaches exist for computational variability—balancing work among ranks to reduce variability, and asynchronous execution strategies to mask residual variability. While asynchronous runtimes [127–130] are commonly used to mask variability, significant time is lost waiting within MPI routines despite their widespread use (>60% at 1,000 ranks [72]). Addressing variability via placement is difficult—frameworks often lack meaningful per-block cost inputs, as predictive models are hard to create and codes often assume uniform costs. This gap motivates leveraging runtime telemetry to directly measure workload behavior.

This section provides background for the rest of this paper, beginning with an overview of AMR codes and current placement techniques in §4.1.1, followed by computational characteristics of AMR operations in §4.1.2.

4.1.1 Block-based AMR: Infrastructure and Execution

Structured AMR Codes. Structured AMR implementations organize computational grids hierarchically while maintaining a logically Cartesian structure. Implementations typically fall into two categories: *block-based* and *patch-based*. Block-based AMR partitions the domain into uniform sized blocks at each refinement level, managed using octree data structures. This approach enables efficient parallel implementations through simplified mesh management and communication patterns. Examples include Parthenon [85], Enzo-E/Cello [131], and ALPS [27]. Patch-based AMR, in contrast, refines regions using arbitrary rectangular patches rather than fixed blocks, allowing finer control over refinement but requires more complex load-balancing (e.g. AMReX [86], SAMRAI [132]).

Execution Model. Block-based AMR codes operate on structured meshes that are decomposed into mesh blocks, with multiple blocks typically assigned to each simulation rank. As the simulation tracks evolving physical phenomena, blocks may be refined or coarsened to adapt mesh resolution, requiring periodic redistribution across ranks. Computationally, operations on each block are structured as a *directed acyclic graph* (DAG) of tasks—including physics computations, mesh management operations, and inter-block communication. Beyond these per-block operations, AMR codes also invoke global operations for redistribution and synchronization, whose characteristics we detail in §4.1.2.

AMR codes use task-based runtimes to execute DAGs using asynchronous execution strategies. Some frameworks like Parthenon implement these abstractions directly, while others use general-purpose runtimes such as Charm++ [127], Uintah [128], HPX [129], and Legion [130]. Our work incorporating telemetry in work placement complements execution strategies masking residual imbalance.

Placement Mechanisms. When a redistribution is invoked, placement mechanisms compute new block-rank assignments, and blocks are migrated accordingly. Modern block-based AMR codes rely on octree mesh structures combined with space-filling curves (SFCs) to enable efficient parallel mesh management while approximately preserving communication locality. SFC-based partitioning balances uniform block counts effectively but does not account for computational variability between blocks—a gap we address in this work. Octree structures, SFC ordering, and their implications for placement flexibility are discussed further in §4.4 where we present our placement policies.

4.1.2 Characteristics of Key AMR Operations

Compute Tasks. Compute tasks perform the core physical and mesh operations of the simulation. While operations like refinement tagging often have predictable costs, physics kernels are a major source of variability. For instance, regions with steep gradients may require more solver iterations, increasing compute cost. Importantly, this cost is not directly correlated to spatial area, as each block contains the same number of computational points (*cells*) regardless of refinement level. Ultimately, the execution time of these compute tasks represents the core computational cost to advance the simulation state, establishing the fundamental lower bound on runtime.

Communication Tasks. Communication tasks create complex inter-block dependencies through *boundary exchanges*, where each block communicates with up to 26 neighbors in 3D (faces, edges, and vertices). Communication volume depends primarily on the number of physical variables exchanged and neighbor relationships instead of the refinement level of the blocks. These exchanges typically use non-blocking MPI operations and are latency-sensitive due to small message sizes and the potential for blocking downstream tasks. *Flux correction* follows a similar small-message, peer-to-peer pattern for ensuring consistency of conserved quantities.

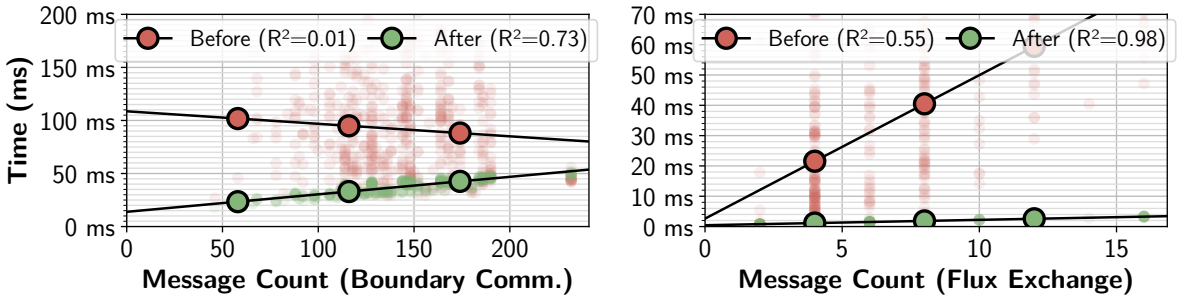


Figure 4.1: *An example of telemetry challenges in AMR codes, showing poor correlation between work (message counts) and communication time. Tuning for system-level issues improves correlation and telemetry reliability.*

Redistribution. Redistribution rebalances blocks across ranks as the simulation evolves, adapting to changes from mesh refinement or coarsening. It is triggered when blocks are tagged for refinement based on physical criteria, like solution gradients exceeding a threshold. Each redistribution computes new block-to-rank mappings and migrates data. Because redistribution may be invoked frequently and must complete within tight performance budgets, placement policies must be fast and capable of handling complex scheduling constraints. The frequency depends on the underlying physics—some problems require frequent adaptation, others are more stable.

Synchronization. Synchronization operations can be explicit (MPI barriers) or implicit (blocking collectives). They inherently expose performance variability by forcing all ranks to wait until the last rank reaches the synchronization point. This waiting time often increases with scale as the likelihood of encountering a straggler grows. While asynchronous collectives are suggested as an alternative [133], synchronous operations remain necessary for certain correctness guarantees, like ensuring that all ranks execute the same timestep.

4.2 Challenges of Telemetry-Driven Placement

In principle, fine-grained telemetry should enable placement decisions to be grounded in observed workload behavior. However, our early attempts revealed two fundamental difficulties. First, the telemetry itself was unreliable: metrics such as communication time showed poor correlation with predictors like message volume (fig. 4.1), making it impossible to model baseline runtime. Second, the magnitude of this variability also changed unpredictably as placement properties were modified. For example, optimizing for communication locality did not reliably improve end-to-end performance initially. These observations forced us to treat placement not as a theoretical optimization problem, but as a systems engineering task grounded in empirically observed runtime behavior. The following challenges structure the empirical and algorithmic contributions of this work.

Challenge 1: Cross-stack Performance Anomalies. When initial telemetry proved inconsistent, showing poor correlation between workload metrics and runtime (fig. 4.1), our first challenge

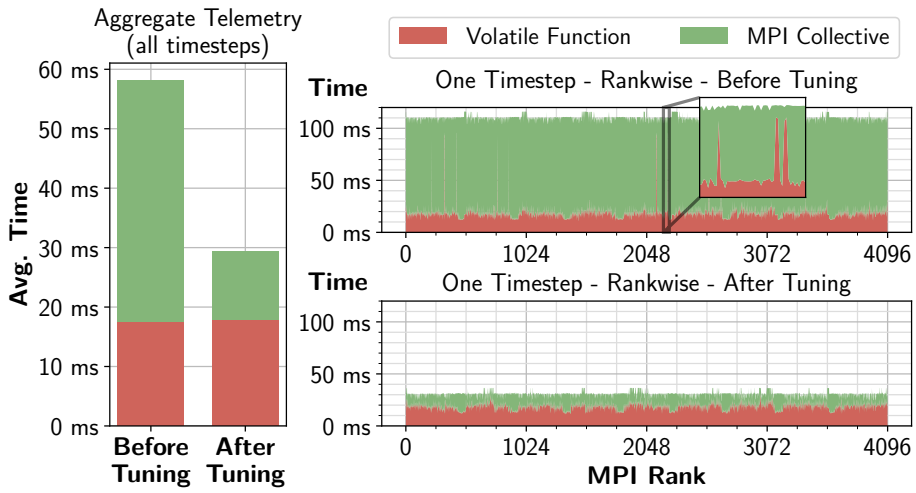


Figure 4.2: *Coarse and fine-grained views of telemetry from a volatile function, followed by a collective. Aggregate telemetry (left) averages out random spikes in the volatile function, attributing the bulk of the runtime to the collective. Fine-grained telemetry (right) reveals subtle performance anomalies affecting average collective time by* $3\times$.

was establishing trust in the measurements. In production environments, organizational and operational boundaries both constrain and scope analyses. Our full-stack access eliminated these boundaries: every deviation, across hardware, drivers, MPI, or application layers, became our direct responsibility to diagnose and explain. While this significantly increased diagnostic complexity, it also provided the crucial advantage of unrestricted access to low-level tools like `perf` [134] and eBPF [135], which were necessary to establish the measurement fidelity underlying this work.

Challenge 2: Analyzing Fine-Grained Telemetry. Diagnosing anomalies required analyzing large volumes of per-timestep, per-rank, and per-block telemetry. Anomalies originated on a small subset of ranks but propagated globally through communication, making root cause identification nontrivial (see fig. 4.2). Tracing tools [28, 136] produced unstructured, high-volume output in formats like OTF2 [136] unsuited for query-driven diagnosis. Our analytics workflow evolved organically, beginning with standard tooling such as TAU [28] and progressing towards custom query-driven workflows capable of surfacing low-level system effects at scale.

Challenge 3: Effective, Low-Overhead Placement. Reliable telemetry exposed genuine load imbalance that could be targeted by placement, but incorporating this information imposed strict new constraints. Placement decisions had to be computed within tight time budgets, under 50 ms per redistribution for our target codes, to avoid measurable overhead on simulation runtime. Even simplified placement objectives, such as minimizing makespan, are NP-hard [137], and additional objectives further complicate optimization. Placement mechanisms had to encode relevant tradeoffs while remaining fast and robust enough for dynamic AMR workloads.

We address these challenges sequentially: §4.3 describes the tuning and analytics methods used to establish reliable telemetry, and §4.4 presents placement policies enabled by this exercise.

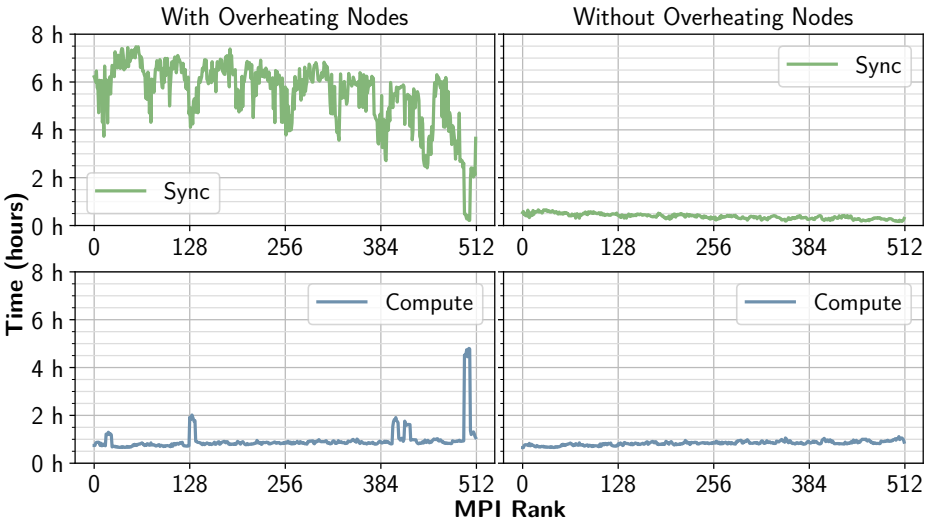


Figure 4.3: *Profiling data from early runs affected by CPU throttling. Compute times on impacted ranks were inflated by up to 4× and appeared in clusters of 16, leading to elevated synchronization delays across the system. Pruning affected nodes reduced overall runtime by 3× by lowering synchronization overhead*

4.3 Denoising Telemetry For Placement

This section describes our methodology for establishing reliable telemetry, addressing the first two challenges from the previous section. We begin with mitigating variability from fail-slow hardware (§4.3.1), followed by examples from different layers of the software stack in (§4.3.2). We then describe the evolution of our analysis workflow in (§4.3.3), followed by a critical path model to formally analyze the impact of task dependencies and scheduling on runtime in §4.3.4. While the specifics of our interventions are necessarily empirical and context-dependent, they serve to illustrate recurring failure modes across the stack and the analytical methods required to uncover them.

Hardware. All experiments ran on a research cluster with 600 compute nodes, each node having Intel Xeon E5-2670 16-core processors, 64GB RAM, and 40 Gbps Qlogic 7340 NICs.

Software. Codes using Parthenon [85], Phoebus [87], and AthenaPK [85], built against MVAPICH2 [138] and the PSM fabric library [139].

4.3.1 Eliminating Faulty and Fail-Slow Hardware

Hardware problems surfaced early in our study and had to be resolved before any software-level conclusions were meaningful. Initially collected profiling data showed more than 70% of runtime being spent in global synchronization (fig. 4.3). Per-rank telemetry showed four-fold compute slowdowns on clusters of 16 ranks—an unmistakable sign of thermal throttling, confirmed by syslogs. Excluding those nodes immediately cut total runtime from 10 h to 2.5 h.

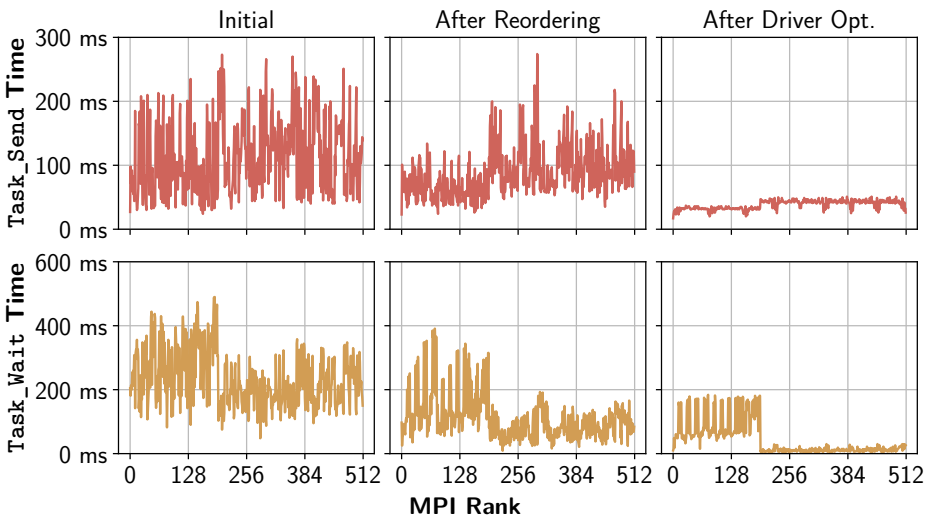


Figure 4.4: *Rankwise boundary communication performance before and after two optimizations. Prioritizing sends in the task schedule reduced noise, revealing faint performance trends. Subsequent queue size tuning further reduced variance, clarifying the underlying telemetry structure.*

While such severe throttling is less likely to persist undetected in production clusters, it serves as an archetype for hardware-related fail-slow behaviors that often manifest more subtly. Studies report transient degradation from throttling, memory errors, or flaky links commonly impacting production workloads [12, 140]. Such faults can present as legitimate stragglers, but must be identified and addressed separately to avoid misdirected tuning efforts.

To ensure measurement integrity, our launch workflow overprovisioned nodes and ran pre/post-job health checks (syslog and other hardware indicators). Failing nodes were automatically pruned from runs and blacklisted.

4.3.2 Tuning The Software Stack

We found interactions within the software stack—application logic, MPI runtime, and network drivers—to be the primary source of performance variability impacting telemetry reliability. While tools exist to check for basic configuration errors or known MPI misuses [141], they cannot capture the dynamic, workload-dependent performance artifacts arising from suboptimal tuning of parameters like queue depths, scheduling priorities, or internal thresholds. We present three representative examples of mitigations applied across the software stack.

Application-Level Example: `MPI_Wait` Spikes. In one code, we traced occasional spikes in `MPI_Wait` following `MPI_Isend` to fabric driver behavior: missing acknowledgments (ACKs) triggered a recovery path that unnecessarily blocked the sender. Since receivers had already acknowledged the messages, blocking the senders on acknowledgments was avoidable. We implemented a drain queue that transparently allocated new MPI requests and drained the blocked requests in the background. fig. 4.2 shows the spikes and the impact of our mitigations.

Application-Level Example: Task Reordering. In another AMR code, MPI send tasks were scheduled after compute/wait tasks, causing cascading delays on CPUs (masked on GPUs where developed) Prioritizing sends (fig. 4.4, middle) unblocked dependent ranks without affecting senders, reducing wait times and jitter.

Network-Level Example: Queue Size Tuning. Persistent variance, contributing to poor correlation between communication time and work (fig. 4.1), was traced using `perf` [134] to contention in the MPI library shared-memory path due to an insufficient preconfigured queue size. Increasing this value reduced MPI wait time variance and significantly improved the correlation between measured communication time and message volume (fig. 4.4, right).

Implications. Mitigating the anomalies described above required analytical capabilities we could not replicate with standard tools. Transient spikes would be obscured by aggregation (profiling), while the sheer volume and complexity of fine-grained data made manual trace inspection infeasible. Systematically identifying root causes—correlating events across ranks, time, application phases, and system layers—necessitated programmable low-level interfaces like eBPF [135], complemented by flexible, query-driven analysis. Our struggles with existing approaches directly motivated the evolution of our analysis workflow, detailed next, towards techniques better suited for this challenge.

4.3.3 Evolution of our Analysis Workflow

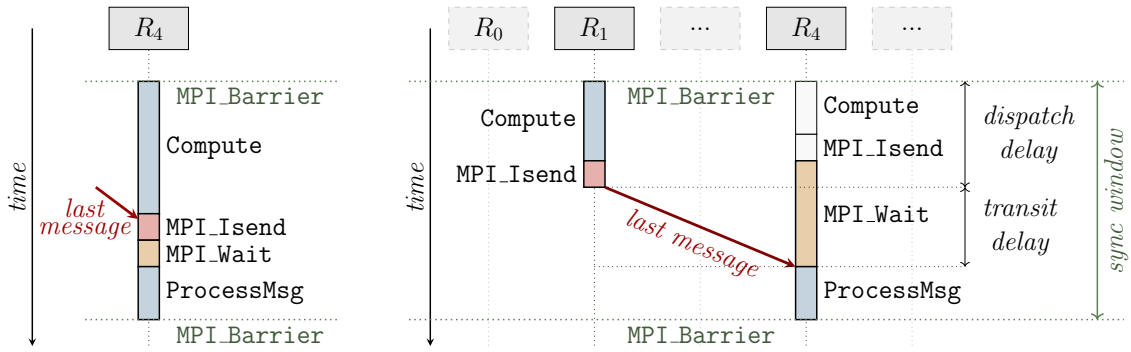
Addressing the analytical limitations described above required evolving our workflow substantially beyond standard tools: TAU [28] provides useful coarse summaries via profiling, but finer-grained telemetry is only accessible via traces, which were impractical to analyze at scale. We wrote TAU plugins to emit CSVs which we analyzed with `pandas` in python. As we scaled up, parsing time became a bottleneck, and we switched to custom binary formats for efficiency.

As our needs evolved, we wanted programmable telemetry triggers based on reconstructed application state. We implemented our own telemetry collection using the MPI [43] and Kokkos [142] profiling interfaces. Eventually, we outgrew pandas' analytical bandwidth and found our telemetry queries naturally mapped to SQL over data ingested into ClickHouse [143], a columnar analytics database.

In hindsight, our pipeline mirrored modern observability stacks: structured schemas, binary formats, and relational queries. Though still ad hoc, it provided the low-latency, queryable insight needed to support targeted diagnosis and hypothesis-driven exploration, essential for tuning and placement at scale.

4.3.4 A Critical Path Model of Execution

Before turning to placement, we introduce a critical path analysis framework to identify optimal task ordering strategies under a fixed placement. This not only grounds the reordering optimization described in §4.3.2, but also helps formally identify other avenues for optimization.



(a) *Single-rank critical path* ***(b)*** *Two-rank critical path*

Figure 4.5: *Critical path examples involving one rank (left) and two ranks (right). Critical paths can be purely local, as in the single-rank case, or involve a maximum of two ranks assuming one P2P communication round.*

We define the *critical path* as the chain of dependent tasks that determines the runtime of the straggler at the next synchronization point. Reduction in critical path length is then modeled as minimizing `MPI_Wait` time on that path, as it is the only flexible-duration component in the execution path. Compute kernels have fixed runtimes, and other communication operations such as `MPI_Isend` and `MPI_Irecv` are similarly fixed, as they simply post buffers to the fabric. We now establish a key principle:

Given a single round of concurrent P2P communication between two synchronization points, at most two ranks can be implicated in the critical path, regardless of scale.

This follows from Lamport's *happened-before* [144] relation — only dependent operations can create causal relationships. If the critical path is local to a rank, it reflects compute imbalance and involves minimal `MPI_Wait` time. More interesting are two-rank paths, where one rank is stalled waiting on a message from another. These cases are illustrated in fig. 4.5b. This model reveals two strategies to shorten such paths:

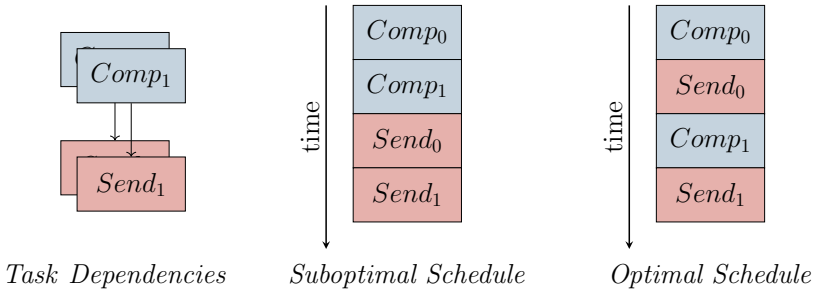


Figure 4.6: *Impact of send ordering on task scheduling. A task schedule for two blocks demonstrating the impact of ordering. Prioritizing $Send_0$ reduces its dispatch time without affecting $Send_1$, minimizing its potential to create a critical path.*

Operation ordering to reduce wait stalls. The most direct way to shorten the path is to ensure the remote message is sent as early as possible. This is precisely what our task reordering optimization accomplished: by prioritizing sends, we minimized the dispatch delay for messages that were on the critical path.

Overlapping computation to hide wait stalls. Asynchronous runtimes discussed in §4.1.1 aim to hide `MPI_Wait` stalls by overlapping them with independent work. However, this requires both a non-zero wait time and the availability of independent tasks. Within a single mesh block, tasks are often data-dependent. While multiple blocks on the same rank can provide independent work, this creates a counterintuitive tension: a strict locality-preserving placement may be detrimental, as all blocks on a rank could end up waiting for the same remote straggler, limiting opportunities for independent work.

This analysis demonstrates the complexity of reasoning about fine-grained execution in partially asynchronous codes, where runtime behavior emerges from a large number of complex and subtle interactions. It also highlights the multiple, sometimes conflicting, dynamics introduced by the locality-reducing placements we discuss next.

4.4 Designing a Placement Policy

With the reliable and interpretable telemetry foundation established in §4.3, we turned to the final challenge: using this runtime information to design effective placement policies. Validated measurements confirmed that MPI collective synchronization dominated runtime, accounting for 35%–50% of total time and increasing with scale (512–4096 ranks), while direct communication and redistribution overhead remained below 10%. This cost was clearly a downstream effect of compute time variability between blocks. Mitigating the impact of this measured variability became the primary placement objective.

Guided by our codes—which, in the worst case, trigger refinement every five 250 ms timesteps—we set a 50 ms placement computation budget to cap overhead at 5% of the total runtime. This constraint ruled out complex optimization objectives such as maximizing communication-compute overlap or encoding network topology, due to limited expected payoff and high constraint complexity. Instead, we focused on two key optimization dimensions:

Compute Load Balance. Directly minimizing the variance in per-rank compute load on the basis of measured block costs was the obvious objective to reduce straggler delays. This problem—formally known as *makespan minization*—is NP-hard [137], necessitating efficient heuristics.

Communication Locality. Baseline SFC-based placements (§4.4.1) preserve spatial locality but hinder load balance flexibility. The actual runtime benefit of this locality had to be evaluated empirically, motivating exploring locality-aware strategies.

We now describe the baseline infrastructure used and augmented by our policies in §4.4.1, followed by load-centric and locality-preserving policies in §4.4.2 and §4.4.3. We conclude with CPLX (§4.4.4), a hybrid policy that enables tunable control over the tradeoff between locality and load balance. All policies were implemented in the Parthenon AMR framework [85] and are directly available in Parthenon-based codes such as Phoebus [87] and AthenaPK [85].

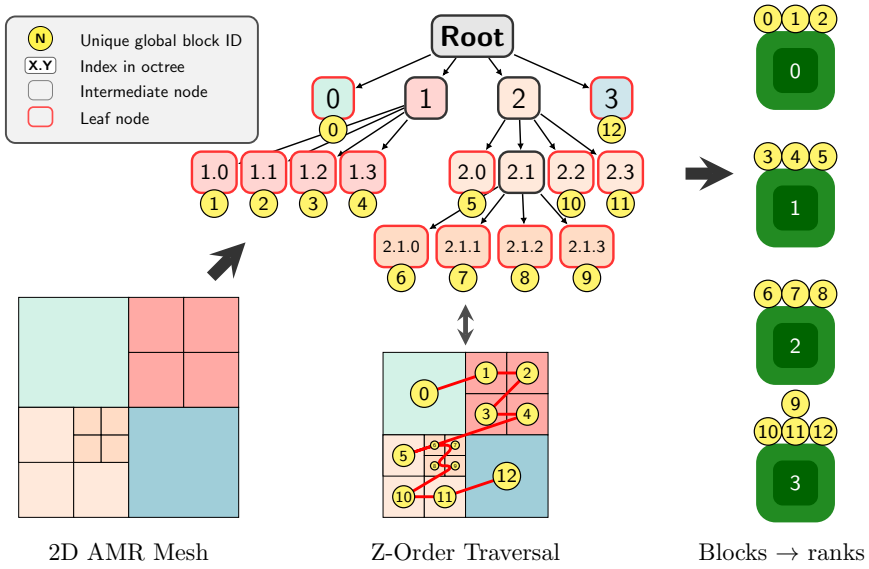


Figure 4.7: *Example of an adaptively refined mesh, octrees, and Z-order Space-Filling Curves (SFCs) in two dimensions. Mesh blocks correspond to octree nodes with refinement creating child nodes. Sequential block IDs are assigned to leaf blocks using a depth-first traversal, equivalent to a Z-Order SFC. Contiguous block ID ranges are then assigned to simulation ranks to balance block counts while preserving locality.*

4.4.1 Existing Placement Infrastructure

Modern block-based AMR codes use octrees and space-filling curves (SFCs) as foundational data structures for mesh management. We now describe these building blocks and the baseline placement policy as a precursor for more sophisticated policies.

4.4.1.1 Octrees and Space-Filling Curves

Octrees provide an efficient hierarchical representation of the mesh structure. When a block needs to be refined, it is split into 8 (2^3) child blocks (in 3D). Only leaf blocks of the octree participate in the simulation. Octrees serve multiple functions in AMR codes—from mesh management to neighbor tracking to block placement—with the latter enabled through easy construction of Z-order space-filling curves via a depth-first traversal of the tree, as shown in fig. 4.7. Blocks are assigned sequential *block IDs* in order of this traversal. This ordering approximately preserves spatial locality, as blocks with nearby IDs are more likely to be spatial neighbors. Some locality, however, is inevitably lost as dimensionality reduction is inherently lossy.

4.4.1.2 Baseline Placement Policy

Placement computation is a component of the *redistribution* process, invoked if the mesh structure changes because of (de)refinement operations. Redistribution proceeds in three steps: blocks are

first assigned block IDs using Z-order SFCs, the placement policy computes new block-rank mappings for all blocks, and finally blocks are migrated to their new ranks using MPI P2P operations.

The baseline policy simply orders blocks by block ID and assigns contiguous ranges of $\lceil n/r \rceil$ or $\lfloor n/r \rfloor$ blocks to consecutive ranks, where n is the total number of blocks and r is the number of ranks. While better assignments are theoretically possible, this approach provides a reasonable balance between compute load and communication locality by balancing block counts across ranks while co-locating spatial neighbors.

4.4.1.3 Augmenting Existing Infrastructure

Octrees and space-filling curves are fundamental, well-studied components of modern AMR codes. Rather than replace these proven building blocks, we augment them with mechanisms to handle variable block costs and flexible allocation patterns. While frameworks like Parthenon provide hooks for specifying per-block costs, these are typically initialized to 1 in practice—treating all blocks as computationally equal. This simplified model persists partly because domain scientists rarely provide cost estimation models for their simulations, and partly because the baseline policy's strict contiguous allocation constraint limits its ability to accommodate variable costs. Our changes to the infrastructure are minimal but enable significantly more flexible placement policies:

1. We populate the existing cost specification hooks with actual computation costs measured via telemetry
2. We modify the block management infrastructure to support arbitrary (non-contiguous) block-to-rank mappings

The second change required minor modifications to data structures but has no correctness implications, as logical dependencies between blocks remain intact. This allows for non-contiguous placements, but communication costs increase only if policies choose to break locality.

4.4.2 LPT: Prioritizing Load Balance

We first investigated whether pure load balancing, ignoring communication locality entirely, could provide meaningful runtime improvements. The *Longest-Processing-Time First* (LPT) algorithm offers a simple but powerful approach—sort blocks by compute cost and assign each to the least loaded rank. LPT is a classical greedy algorithm for makespan minimization with strong theoretical guarantees—the makespan of an LPT solution is provably at most 4/3 times the optimal makespan [145]. In practice, LPT performs remarkably well; we could not obtain better solutions from a commercial ILP solver [146] despite letting it run for 200 s.

LPT demonstrated surprisingly strong performance despite completely ignoring communication locality. While it did incur higher communication costs than the baseline approach, the improvements in load balance more than compensated for this overhead. This suggested that the value of locality preservation might be lower than initially assumed, motivating us to explore this tradeoff further.

4.4.3 CDP: Preserving Locality

After observing LPT perform well despite its locality costs, we wanted to explore the potential of a locality-maximizing load-balanced placement. To this end, we developed *Contiguous-DP* (CDP), which uses dynamic programming to select contiguous block ranges to improve load balance while preserving the locality patterns of the baseline.

Consider a simulation with 10 blocks and 4 ranks. With an average of 2.5 blocks per rank, CDP explores allocations like `[2,2,3,3]` or `[2,3,2,3]`, finding one that minimizes makespan while maintaining contiguous allocations. As a result, CDP has identical locality-preserving properties as baseline.

Formal Description. CDP solves the contiguous partitioning problem: given block costs $w_1, \ldots, w_n$ (in SFC order), partition them into r contiguous segments to minimize the maximum segment sum (makespan). Let $W[i] = \sum_{j=1}^{i} w_j$. The minimum makespan $\text{DP}[i][k]$ for partitioning i blocks among k ranks follows ($\text{DP}[0][0] = 0$):

$$\text{DP}[i][k] = \min_{0 \le j < i} \max\left(\text{DP}[j][k-1], W[i] - W[j]\right)$$

The algorithm has complexity $O(n^2 r)$ in general, where n is the number of blocks and r is the number of ranks. However, we optimize it to $O(nr)$ by considering only two contiguous chunk sizes: $\lceil n/r \rceil$ and $\lfloor n/r \rfloor$. This optimization maintains solution quality while making CDP practical for AMR timescales. This DP formulation guarantees an optimal makespan-minimizing placement within the explored chunk sizes.

In our experiments, CDP improved upon baseline but only reached parity with LPT. While CDP showed lower communication times, its higher synchronization times suggested that perfect locality preservation might be unnecessarily restrictive, motivating our development of our hybrid policy called CPLX, described next.

Scaling CDP With Chunking. The overhead of computing placements with CDP became noticeable at 4096 ranks, prompting us to develop a parallel implementation using hierarchical chunking. This approach divides blocks into c contiguous chunks of approximately equal cost, then applies CDP independently to each chunk using a subset of ranks. At 4096 ranks with chunk size 512, this creates 8 parallel-processed chunks.

While chunking may not find the globally optimal CDP solution, this approximation has minimal impact since CDP's output serves only as an intermediate step for CPLX. The approach reduces placement overhead while retaining CDP's solution quality.

4.4.4 CPLX: Combining the Best of Both

Our experiments with LPT and CDP revealed a clear tradeoff—LPT achieved better load balance but higher communication costs, while CDP preserved locality at the expense of some load imbalance. Neither policy consistently outperformed the other in end-to-end runtime, suggesting the potential for a hybrid approach that could combine their strengths.

RANKS	MESH SIZE	$\mathbf{t_{total}}$	$\mathbf{t_{lb}}$	$\mathbf{n_{initial}}$	$\mathbf{n_{final}}$
512	128^3	30,590	1,213	512	2,080
1024	$128^2 \times 256$	43,088	4,576	1,024	3,824
2048	128×256^2	43,042	4,699	2,048	4,848
4096	256^3	53,459	9,392	4,096	8,968

t_{total}: total timesteps, t_{lb}: timesteps invoking load-balancing
n_{init}: initial block count, n_{final}: final block count

Table 4.1: *Problem configurations for Sedov Blast Wave 3D experiments. All configurations use 16^3 block size, initializing with one block per rank to ensure meaningful placement impact at all scales. The number of blocks increases through refinement as the shock wave propagates, with final block counts shown in the rightmost column.*

Developing such a hybrid proved challenging. Our initial attempts to blend the policies produced unpredictable results—small sacrifices in load balance did not translate to proportional gains in locality, and vice versa. We eventually realized that it was easier to selectively break locality in a contiguous placement than to restore locality in an arbitrary one. This insight led to the CPLX design principle: start with a locality-preserving solution from CDP and strategically apply LPT to address the most significant imbalances.

CPLX begins by computing an initial CDP placement that reuses the chunking mechanism for scalability. It then assesses the work allocated to different ranks by sorting them in descending order of load. The policy selects X% of ranks from either end of this sorted list—the most overloaded and underloaded ranks—for rebalancing via LPT. Including both ends is crucial as rebalancing needs both source and destination ranks to effectively redistribute work.

The parameter X, ranging from 0 to 100%, provides precise control over this locality disruption. At X=0 (`CPL0`), no ranks participate in rebalancing, preserving the locality characteristics of CDP. At X=100 (`CPL100`), all ranks undergo LPT rebalancing, effectively becoming pure LPT. Intermediate values create hybrid behaviors, only disrupting locality within the X% most imbalanced ranks while preserving it elsewhere. This enables CPLX to systematically explore the complete locality-balance tradeoff space while maintaining the performance requirements of AMR codes.

4.5 Evaluation of Placement Policies

In this section, we evaluate CPLX under real AMR workloads and synthetic microbenchmarks. While `CPL100` (pure LPT) and the baseline policy represent known quantities, our key question is whether intermediate values of X can provide meaningful control over the locality-balance tradeoff. For CPLX to be successful, it must achieve three goals: intermediate values should demonstrate measurable control over both communication patterns and runtime performance, achieve minimum runtime if the tradeoff is worth balancing, and maintain low runtime overhead

even in setups with rapid refinement. We evaluate these aspects by comparing different values of X against `CPL100` as our reference point.

We primarily evaluate CPLX using the Sedov Blast Wave 3D problem in Phoebus [87], a hydrodynamics code built on top of the Parthenon [85] framework. While we also studied placement in other codes, such as a galaxy cooling setup in AthenaPK [85], results were directionally similar: codes with high compute variability benefit more from better placement, and vice-versa. We describe our hardware setup in §4.5.1, present Sedov results in §4.5.2, and use microbenchmarks to evaluate CPLX's load balance and locality properties under different synthetic regimes in §4.5.3.

4.5.1 Experimental Setup

We evaluate four configurations of the Sedov problem, as detailed in Table 1. Each run begins with one block per rank, increasing as refinement proceeds, and finally settling at ∼two blocks per rank. The table reports initial and final block counts, total steps, and load-balancing invocations.

All experiments ran on the tuned Emulab-backed cluster described in §4.3, with 16 ranks per node, at scales from 512–4096 ranks. Telemetry was collected using the custom profiling stack described in §4.2. We compare the tuned baseline against CPLX with X $\in \{0, 25, 50, 75, 100\}$, and only report runtime gains from placement. Runtime gains from tuning are omitted because of variable impact across different codes and scales.

Hardware Checks. To ensure experimental integrity, all reserved nodes were scanned for hardware issues—thermal throttling, memory errors etc.—before every individual run in each parameter suite using the process described in §4.3. All reported results are from runs where no participating node showed issues either before or after execution.

Software Stack. All experiments used MVAPICH2 v2.3.7 [138], compiled against a tuned version of PSM [139], the userspace library for our QLogic fabric.

4.5.2 Sedov Blast Wave Results

We evaluate five placement policies on the Sedov Blast Wave problem across four scales (512–4096 ranks), comparing baseline performance against CPLX, as X is varied from 0–100. The analysis focuses on overall runtime behavior (fig. 4.8), the synchronization—communication tradeoff (fig. 4.9), and P2P communication patterns (fig. 4.10).

Finding 1: *Baseline: synchronization reaches 50% of total runtime at scale.*
Fig. 4.8 shows total runtime across all policies, decomposed into computation, communication, synchronization, and rebalancing phases. With baseline placement, computation and synchronization dominate the profile, jointly accounting for over 90% of execution time at all scales. Communication and rebalancing remain minor components, contributing approximately 7% and 3% respectively. Synchronization overhead grows sharply with scale—from 35% at 512 ranks to 50% at 4096 ranks, despite the tuned environment. These results demonstrate the challenge of

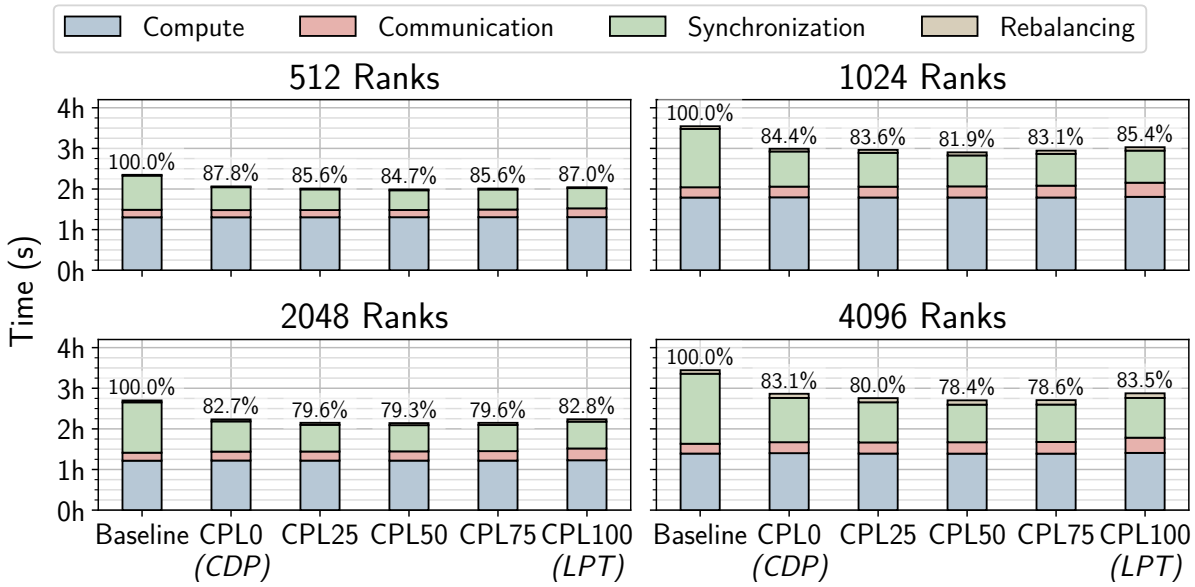


Figure 4.8: *Sedov evaluation: performance breakdown by execution phase and policy. Total runtime across scales (512–4096 ranks), classified into phases. CPL50 performs best, with up to 21.6% runtime reduction over baseline.*

managing dynamic behavior in bulk-synchronous parallel applications. Unless mitigated, runtime is dictated by stragglers, the likelihood of which only goes up with scale.

Finding 2: *CPLX achieves up to 21.6% overall runtime reduction, with impact increasing with scale.*
Fig. 4.8 also shows that all CPLX configurations outperform the baseline, with runtime improvements that grow more pronounced at larger scales. At 4096 ranks, the best-performing variant (`CPL100`) reduces total runtime by up to **21.6%** relative to baseline. Even at 512 ranks, CPLX achieves a 15.3% reduction, and all values of X tested yield improvements above 12%.

Runtime follows a clear U-shaped curve as a function of X: starting from `CPL0` (locality-preserving), decreasing to a minimum with intermediate values, and rising again toward `CPL100` (pure LPT). This shape reflects the ability of CPLX to control the load-locality tradeoff in a manner that yields meaningful overall runtime gains.

Crucially, compute time remains flat across all policies, confirming that the total amount of work is invariant to placement. The observed improvements arise entirely from reductions in synchronization overhead, which dominate non-compute runtime in the baseline configuration. Measured as reduction in non-compute time, the impact of CPLX becomes even more pronounced—up to 36% reduction at 4096 ranks.

Finding 3: *CPLX enables consistent control over the load-locality tradeoff, and tradeoff impact increases with scale.*
Fig. 4.9 isolates the core tradeoff by showing P2P communication and synchronization times, normalized against the baseline, for all policy configurations at 512 and 4096 ranks. Intermediate scales are omitted for compactness, but we find them to demonstrate similar trends. Increasing X reduces synchronization time, as expected from better load balance, while increasing communication time due to loss of locality.

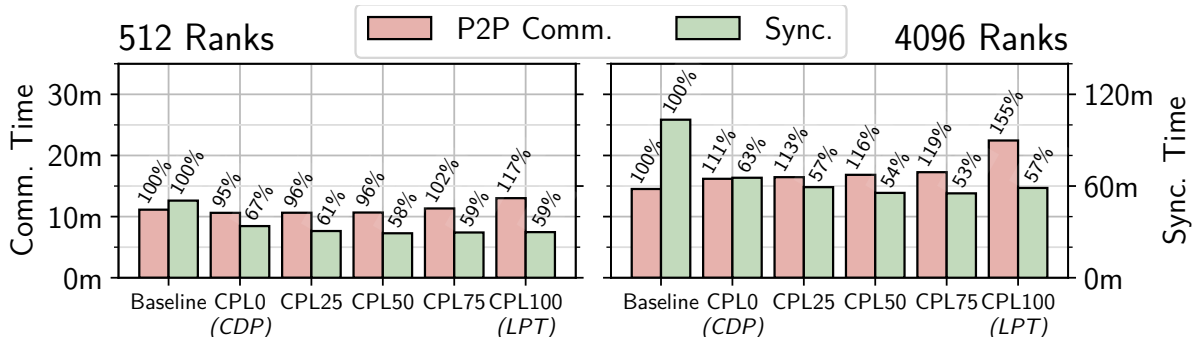


Figure 4.9: *Sedov evaluation: communication and synchronization time tradeoffs. Clear tradeoff emerges between communication time (increasing with X) and synchronization time (decreasing with X), demonstrating how CPLX enables controlled balance between these competing factors.*

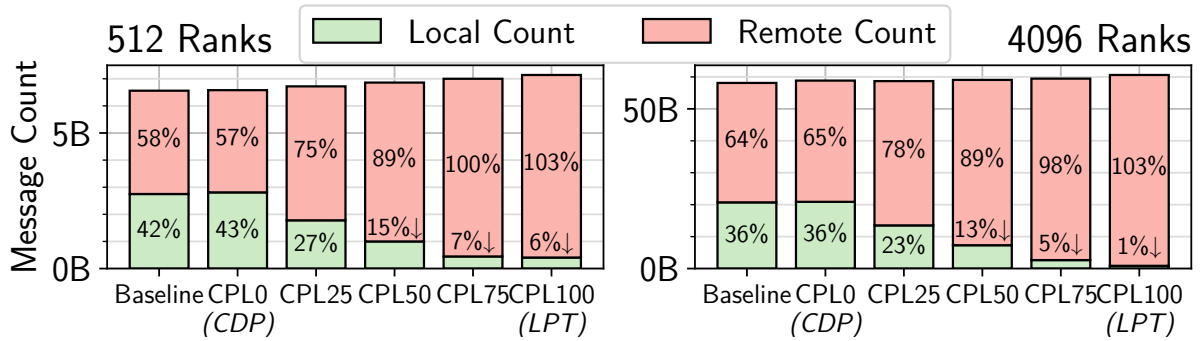


Figure 4.10: *Sedov evaluation: locality analysis by message volume. Message patterns show that increasing X reduces local in favor of remote communication, providing evidence for the locality-performance tradeoff.*

We find that modest values of X (25–50) capture almost all of LPT load-balancing benefits, without incurring the commensurate locality cost. While LPT incurs a relative increase of 55% in locality cost, its impact on the overall runtime remains modest as even at 4096 ranks, communication is only 13% of the overall runtime. The increasing spread between intermediate X values and LPT with scale suggests that the runtime impact of balancing the two increases with scale. Given the strong performance of LPT vs commercial ILP solvers (§4.4.2), this is likely close to the practically achievable optimum under our model of AMR placement.

At 512 ranks, some CPLX configurations even reduce communication time relative to baseline—this is explained by the compounding effects captured by boundary communication time. High variance in the preceding compute kernels results in more time spent in `MPI_Wait` by some ranks, and improving load-balance has a positive downstream impact that overrides the locality cost. This is an illustration of higher-order placement effects that we discard for computational tractability.

Finding 4: *P2P message volume statistics confirm locality degradation with increasing X.*

fig. 4.10 shows the breakdown of local (intra-node shared memory) and remote (inter-node MPI) P2P messages, normalized to the total baseline message volume, for CPLX variants at 512 and 4096 ranks. As X increases, remote message counts rise while local messages fall, reflecting the

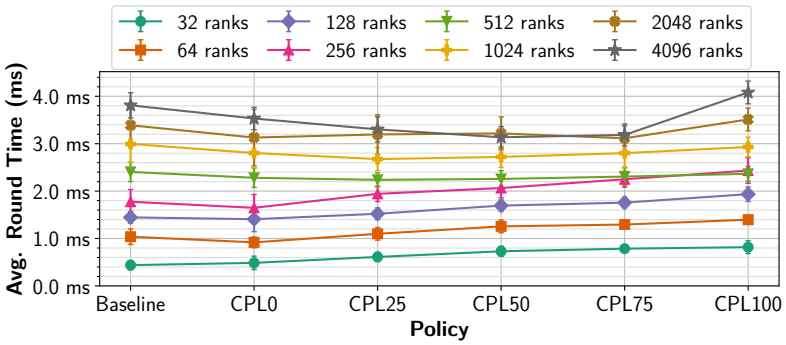


Figure 4.11: *Microbenchmarks: boundary communication latency vs. policies (`commbench`). Boundary communication round latency vs. locality at different scales. Locality modestly affects round latency ($\pm 0.5ms$). At larger scales, strict locality preservation surprisingly becomes counterproductive, as it results in communication hotspots relative to hybrid placements.*

fact that CPLX progressively expands LPT coverage at the expense of locality. This tradeoff is enacted mechanically by CPLX, independent of the underlying application or mesh—only the runtime impact of this shift depends on workload characteristics. We explore workload variations further in §4.5.3 using synthetic microbenchmarks.

A subtler effect visible in fig. 4.10 is the growth in total MPI-visible message volume with increasing X. This occurs because intra-rank communication—handled via `memcpy` when blocks are co-located—is replaced by MPI messages when placement breaks locality. This shift explains part of the communication time increase observed in fig. 4.9.

Finally, even the baseline placement routes a majority of messages across nodes: at 4096 ranks, **64%** of messages are already remote. This is an intrinsic property of space-filling curves—dimensionality reduction to 1-D preserves only partial spatial locality. CPLX does not introduce a new communication cost class, but extends an existing placement pattern and exposes it as a tunable parameter.

4.5.3 Microbenchmarks

To complement the application-level Sedov results (§4.5.2) and enable controlled evaluation across a broader range of conditions, we developed two microbenchmarks. `commbench` isolates P2P communication under varying placements, while `scalebench` evaluates the effectiveness and computational cost of placement policies under synthetic compute imbalance.

Commbench: Simulating Boundary Communication Patterns.

`commbench` is a synthetic microbenchmark that isolates boundary communication and evaluates how placement locality affects end-to-end round latency. It constructs octree-based AMR meshes with realistic refinement, deriving P2P patterns from geometric neighbor relationships (face, edge, vertex) across refinement levels. While inspired by the Halo3D-26 fixed-grid benchmark [147], `commbench` simulates a full AMR placement pipeline and accepts custom placement policies as drop-in modules.

Each communication round emulates a boundary exchange, with blocks exchanging messages based on neighbor type. Message sizes are realistic: face-neighbor exchanges are proportionally larger than edge or vertex ones. Meshes are refined to yield 1–2 blocks per rank, and each

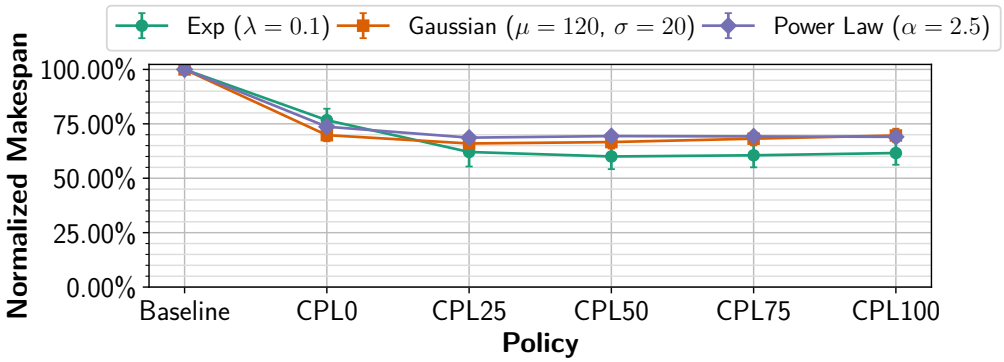


Figure 4.12: *Microbenchmarks: load balance quality vs. policies (`scalebench`). Average makespans from CPLX placements with three synthetic distributions. While LPT performs best across all distributions, the bulk of the benefits are realized with $X = 25$.*

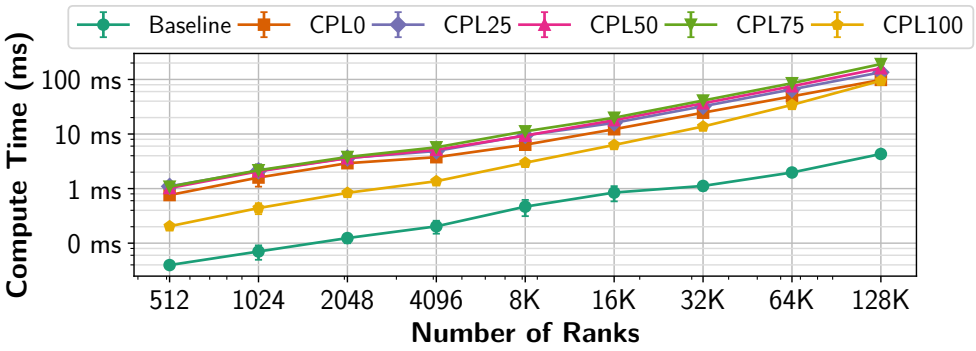


Figure 4.13: *Microbenchmarks: policy compute cost vs. simulated rank count (`scalebench`). Compute time for CPLX as a function of scale, averaged across three synthetic distributions. Compute time stays below 10ms until 16K ranks, and can be reduced beyond that scale using smaller parallel instances.*

round includes all neighbor types. Timing is captured using MPI barriers before and after each round. Results are averaged over 100 rounds and 10 random meshes per policy. We discard cold-start rounds and those with latency above 10 ms, which we found to reflect fabric-level recovery behavior unrelated to placement.

Fig. 4.11 shows average round latency across placement policies as locality decreases from `CPL0` to `CPL100`. While higher locality yields lower latency at small scales (512 ranks and below), a surprising U-shaped trend emerges at larger scales: intermediate X values outperform both extremes. This inversion is driven by face-neighbor communication, which increases as meshes get denser. Locality-preserving policies cluster high-traffic neighbors unevenly, increasing per-rank load. Intermediate CPLX settings diffuse this clustering without fully randomizing placement, leading to more uniform traffic patterns. Though the latency differences are modest ($\pm$0.5 ms), the reversal of expected trends underscores how observed system behavior can defy theoretical intuitions. `commbench` provides a practical mechanism for empirically selecting X in different environments.

Scalebench: Evaluating Imbalance and Placement Overhead. `scalebench` evaluates different policies at scales from 512 to 128K ranks, with 1–2 blocks per rank to enable flexible load-balancing. Block costs are drawn from three representative distributions—exponential, Gaussian, and power-law—with variability bounds chosen to create meaningful balancing opportunities while remaining within realistic AMR ranges.

Fig. 4.12 plots the normalized makespan across placement policies (lower is better). We see that `CPL100` (LPT) achieves the lowest makespan across distributions, but `CPL0` and `CPL25` capture the bulk of the benefits with much higher locality retention. Fig. 4.13 reports placement computation overhead as a function of scale. Compute costs increase with scale, but remain tractable (~10 ms) up to 16K ranks, rising to 100 ms at 128K ranks. At the largest scales, zonal placement architectures can be adopted to mitigate placement overhead—dividing ranks into k zones to compute placement independently and in parallel [148].

4.6 Related Work

Variability and Runtime Analysis. Performance variability in HPC environments is well documented [10, 11, 125, 149, 150], with root causes ranging from tuning to contention. Separately, fail-slow hardware has been reported as a source of performance degradation [12, 140]. Similar variability challenges have been identified in the context of collective performance in hyperscale ML clusters [58]. Varbench [151] is a framework to characterize computational variability using a suite of compute kernels. Klenk et al. [72] analyze proxy MPI traces and report significant MPI overheads (60% at 1024 ranks) and attribute them to load imbalance, while also noting limitations of trace analyses. While prior work documents separate causes of stragglers, an end-to-end treatment combining tuning and placement optimization has been missing—a gap addressed by our work.

Performance Telemetry and Tooling. Established performance tools such as TAU [28] and Score-P [136] collect profiles and traces (e.g., OTF2), but lack the queryability required for scalable analytics, a limitation previously noted by Klenk et al. [72]. Caliper [35] supports structured telemetry and programmable aggregation, but relies on plaintext data formats. Parquet [41] provides a compact, binary columnar format with embedded statistics for efficient analytics, but is not widely adopted in performance tools. Hatchet [40] builds a structured in-memory representation from external telemetry sources but assumes prior parsing. Metric frameworks like LDMS [68] offer low-overhead system-level metrics, but often lack application context needed for bottleneck attribution. Yang et al. [152] describe a general critical path analysis framework, while our framework in §4.3.4 is tailored for AMR codes.

AMR and Communication Models. Modern AMR codes either use MPI or asynchronous runtimes like Charm++ [127] and Legion [130]. Neighborhood collectives [149, 153, 154] for boundary communication have been proposed as an alternative to point-to-point messaging, but are currently not used in the AMR codes we evaluated.

Placement and Load Balancing. Graph-based partitioners such as parMETIS [155] and Zoltan [156] handle complex meshes but incur high overhead. Sasidharan et al. [157] propose a SFC-based partitioner with lower overhead than METIS. All graph-based approaches model communication as edge cuts, which we find poorly correlated with runtime communication overhead. MetaBalancer [158] focuses on rebalancing triggers, while Zheng et al. [148] propose hierarchical schemes to scale load balancing. Solver-based approaches are used to optimize placement in cloud datacenters [159], but are computationally expensive (multi-hour) and impractical for AMR.

4.7 Lessons for BSP Diagnostic Workflows

This chapter traced the end-to-end process of developing telemetry-driven placement policies for AMR codes. It illustrates the fragility of the BSP paradigm: a single straggler at any synchronization point idles all other ranks, and the likelihood of encountering one grows with scale. The technical outcome of this study was CPLX, a placement policy that reduced runtime by up to 21.6% over tuned baselines by exposing the tradeoff between load balance and communication locality. But the dominant effort was not placement design—it was the diagnostic process: identifying and eliminating confounders across the stack, evolving our telemetry and analysis workflow through several iterations, and repeatedly validating that observed effects were genuine. The following lessons distill what this process revealed about the requirements for effective BSP diagnostic workflows, and inform the design of ORCA in the next chapter.

Lesson 1: *Straggler attribution requires fine-grained telemetry.*
Distinct straggler mechanisms are indistinguishable without access to fine-grained telemetry. Load imbalance, hardware variability, scheduling effects, and tuning artifacts all manifest as late arrival at a barrier. Aggregate timing alone cannot distinguish between them—attribution requires per-rank, per-timestep visibility into the events leading up to the stall. Codes with peer-to-peer message patterns add further complexity. Asynchronous execution introduces scheduling and ordering as additional sources of variability, and the messages themselves create indirect paths through which delays propagate before surfacing at synchronization. Fine-grained runtime telemetry is a prerequisite for disentangling these causes.

Lesson 2: *Attribution requires live runtime context.*
The predominant mechanism for fine-grained collection in BSP is traces captured as per-rank logs and analyzed post-hoc. Traces are effective for general characterization of workload properties, but root-cause localization often requires additional telemetry and hardware and runtime context that is no longer available once the trace has left the execution environment [72]. Many performance pathologies can only be diagnosed and resolved in the production environment where they manifest, which requires that the relevant signals be observable in situ.

Lesson 3: *Mitigations are context-specific and require continuous visibility.*
Runtime unknowns such as tunable parameters encode context-specific decisions and do not

admit universal fixes. Our placement policy, CPLX, improved runtime by up to 21.6%, but its optimal tradeoff between load balance and communication locality varies with workload, cluster, and scale. Such conditions can be mitigated but not eliminated. Continuous visibility is needed not only to guide initial parameter selection, but also to detect regressions as conditions change.

Lesson 4: *Fine-grained telemetry collection must be hypothesis-driven and guided by causal structure.* All static tracing profiles cut off at arbitrary instrumentation depths. Execution has practically infinite fidelity: every function call can be traced further. Tracing deeper exhaustively is both unviable and unnecessary. Stragglers in BSP, for example, can be identified from collective latency distributions across ranks. Only the anomalous ranks warrant deeper inspection, and only the functions contributing to their delays. Effective diagnosis must therefore be hypothesis-driven: it follows causal structure downward in response to observed anomalies, instead of attempting exhaustive collection in advance.

Lesson 5: *Hypotheses require extensible telemetry collection and programmable analytics.* Fine-grained telemetry must be assembled into views that preserve causal relationships before hypotheses can be validated. This places requirements on both the collection and analytics layers.

- Collection infrastructure must permit flexible telemetry schemas. In our study, while standard tracing tools [28] captured per-task telemetry via Kokkos [142] instrumentation, they did not capture application-level *block IDs* that we needed to connect timing data to meshblocks for placement analyses. Telemetry from different runtime sources may require different schemas — no static schema can anticipate every analytical need.
- Analytics must support programmable views assembled on demand. eBPF demonstrated this paradigm during our investigation: its mechanisms enable selective instrumentation of arbitrary points in the execution stack, activated on demand rather than compiled in. This was decisive for localizing latency spikes when other approaches failed. The limitation is that eBPF is node-local—events are indexed by hostname and process identifier, probes must be manually activated on each node, and results must be stitched across nodes and resolved into ranks.

At BSP scales, predominant event trace formats [31, 32, 160] serve neither timeline visualization nor analytics. The size of these traces precludes interactive exploration via timeline-based trace viewers [161], and their structure and formats impose prohibitive ETL (*extract-transform-load*) costs before dataframe style queries can be computed. Columnar and queryable representations [162] are the right foundation for managing BSP telemetry.

Chapter 5

ORCA: Steerable BSP Observability

Contents

In the previous chapters, we discussed two instances of feedback loops over distributed BSP state. CARP implements an online but fixed-function loop, reconstructing the global key distribution to drive adaptive, in-situ range partitioning. The AMR study implements an offline loop, where trace analysis informs reconfiguration and reruns. In both cases, the relevant state is not knowable a priori, and making progress requires iterating between measurement and action. The AMR experience highlights the practical limitation of doing so offline: iteration latency becomes the bottleneck, and the resulting insights and mitigations are often deployment-specific rather than readily transferable. These properties motivate generalizing the pattern

into a substrate that can materialize feedback via in-situ reductions of application state and support online interventions in response. Such a capability directly addresses the visibility requirements identified in §4.7: always-on, in-production visibility that surfaces high-level symptoms of pathologies and enables online steering. It also subsumes CARP's renegotiation loop, which becomes a policy running on the substrate rather than a bespoke mechanism. We call the resulting paradigm *steerable observability*.

Steerable observability requires two capabilities: feedback, materializing views over distributed state, and control, coordinating interventions across ranks. The loop may be steered by human operators, automated agents, or policies. No existing tool provides these as an always-on, BSP-aware capability. Metrics and logs provide baseline visibility but rarely enough workload structure to attribute causes. BSP traces are emitted as per-rank logs and analyzed post-hoc. Distributed tracing as evolved through X-Trace [163] and Dapper [164] captures causality in asynchronous microservices by propagating request IDs, but the model does not fit BSP. Timesteps separated by global synchronization, not requests, are the primary unit of work, and even identifying the straggler at a single barrier requires aggregating telemetry from all ranks participating in that barrier. Control poses a coordination challenge: interventions must be applied with well-defined consistency and atomicity across ranks. Parallel debuggers [80] provide interactive control but are designed for debugging sessions, not always-on, in-production observability.

Steerable observability requires a programmable interface for streaming reductions over distributed telemetry. Dataframe-style trace analytics are increasingly the norm in both HPC and ML training workflows [34–36, 71] A streaming, in-situ variant is the logical online counterpart. We propose *timestep dataframes* as the core abstraction for BSP telemetry: a logical representation comprising all telemetry emitted during a single timestep across all ranks. Because BSP synchronization causally partitions execution, each timestep dataframe is a natural unit of analysis: it summarizes what happened at workload scale and provides the local context needed to attribute causes. External stream processing systems can theoretically window event streams [37, 38], but they decompose this structure into individual events that must be reaggregated, cannot leverage RDMA fabrics, and cannot provide coordinated control semantics. An on-fabric tree-based overlay over columnar data is the natural architecture: Arrow's contiguous memory layout enables both efficient in-situ reductions and efficient transfers over RDMA, while the same tree topology supports broadcast for coordinated control. The result is observability infrastructure that leverages a supercomputer's resources rather than requiring a second one to process its output.

We present a *tree-based overlay network* (TBON) for real-time observability and control of large-scale BSP applications, called ORCA (*Observability with Real-time Control and Aggregation*). ORCA treats telemetry as columnar streams, transformed via a SQL-based analytics interface called *OrcaFlow* en route to *sinks* such as dashboards or parallel filesystems. An RDMA-based TBON-optimized transport (§5.3.2) enables low-overhead collection of timestep dataframes. They can then be transformed via SQL-based analytics, routed to a dashboard, or persisted as partitioned Parquet for zero-ETL post-mortems. This accelerates observability workflows across

the spectrum, ranging from real-time monitoring over application-specific metrics, to offline trace analytics. Distributed query planning with operator pushdown minimizes data movement across the overlay (§5.3.3). A control plane (§5.3.4) provides atomic and timestep-consistent control semantics, guaranteeing correctness for interventions such as new analytics, probes, and parameter updates. ORCA combines a modern columnar analytics stack [39, 41, 42] with RDMA and BSP-aligned data and control paths to realize low-overhead, real-time, closed-loop feedback. This accelerates observability workflows across the spectrum, from application-level metrics monitored in real time, to post-hoc trace analyses backed by in-situ transforms and timestep-partitioned Parquet.

We evaluate ORCA on a real AMR code at up to 4096 CPU ranks.

- In conventional tracing workflows, ORCA incurs ∼2% runtime overhead (vs 56–81% for state-of-the-art tracers), produces up to 42× smaller traces for equivalent telemetry, and enables up to 6900× faster analyses than existing trace formats (§5.5.1).
- In-situ analytics enable low-overhead needle-in-the-haystack workflows—outlier `MPI_Wait` calls are isolated in real-time at sub-2% runtime overhead, while discarding up to 99.999% of data at MPI ranks (§5.5.2).
- In an agentic evaluation of its closed-loop capabilities using an LLM agent as a proxy for a human operator, ORCA enables localizing a performance anomaly—the same bug from our AMR study that took months of manual investigation—in 27 minutes, while reducing data volume by up to 144× (§5.5.3).

More broadly, ORCA demonstrates that declarative SQL analytics can be embedded on a supercomputer fabric alongside a live MPI application, enabling streaming analytics workflows beyond observability. ORCA is open source [165].

5.1 Background and Motivation

With the motivation from CARP and the AMR study established, this section surveys existing approaches and identifies gaps that inform ORCA's design. We use the pathology in Fig. 5.1 — observed and mitigated during the AMR study — as a running example. Fig. 5.1 shows volatile `MPI_Wait` durations preceding a collective at 4096 ranks. Continuous collection of this single function produces approximately 4 M events/s at this scale. The root cause lies somewhere in the application's call graph—not only is it unknown which functions under `MPI_Wait` matter, but whether `MPI_Wait` is the right place to look at all.

The ideal workflow is a feedback-driven drill-down. Collective times can be aggregated to produce simple, actionable signals in the common case. In this example, the 1st-percentile collective duration is sufficient to reveal the presence of stragglers. From there, a simple count of outlier `MPI_Waits` (duration > 10 ms) would confirm the source. Additional probes from within this function's call graph would further isolate the root cause. A feedback loop that materializes compact aggregates in-situ would enable a spectrum of workflows, from lightweight common-

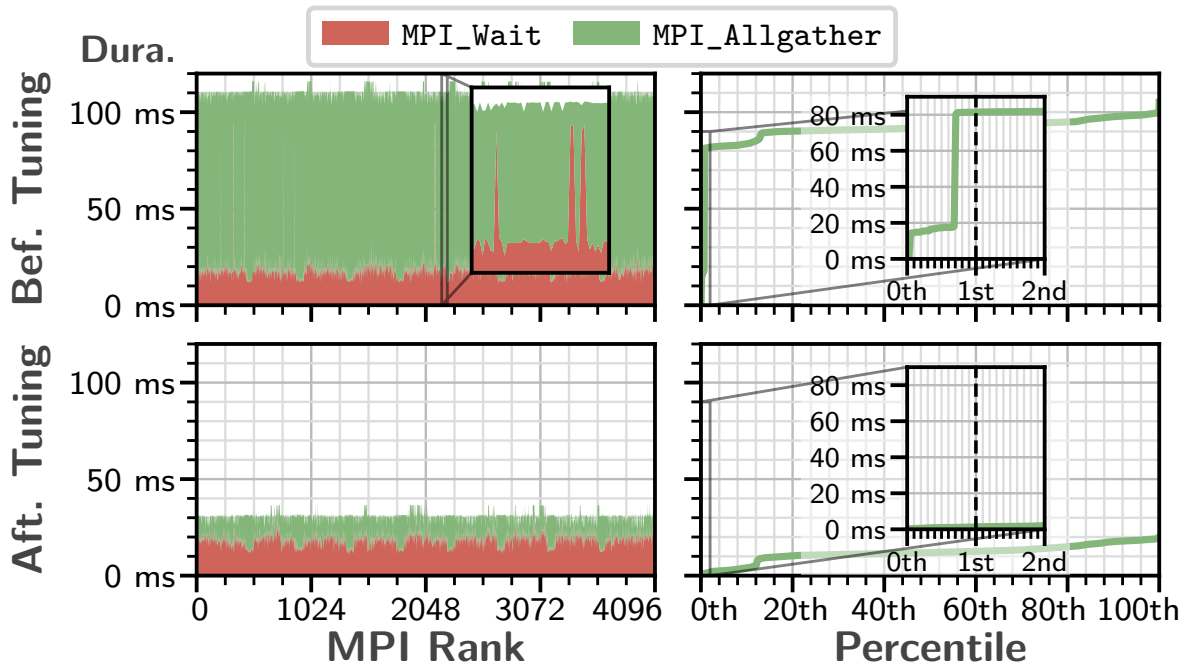


Figure 5.1: *Telemetry from Fig. 4.2 showing volatile `MPI_Wait` preceding `MPI_Allgather`, before and after tuning. (Left) Identifying stragglers requires fine-grained per-rank telemetry joined by timestep. (Right) Percentiles over per-timestep collective durations can be converted into lightweight metrics for continuous monitoring. Here, 1^{st}-percentile collective durations would reveal stragglers.*

case monitoring over application-derived metrics to detailed investigation. This requires in-situ reduction, runtime control over what is collected, and BSP awareness — computing collective percentiles requires joining O(100K) event streams by timestep.

The rest of this section examines solutions from the PyTorch ecosystem for similar pathologies in ML training (§5.1.1), surveys existing tool categories against this workflow (§5.1.2), and describes the template that informs our approach (§5.1.3).

5.1.1 Parallels from the Pytorch Ecosystem

The PyTorch [166] ecosystem has independently converged on a similar diagnostic workflow, with the same structural split: programmable observability is offline and ETL-bound, while online observability is fixed-function over selected metrics. Its diagnostic stack consists of Kineto [29] for fine-grained trace capture as JSON, Dynolog [167] for node-local control to trigger capture, and HTA [168] for derived views such as straggler detection. Together these form a closed loop, but HTA cannot scale to cluster-wide traces without expensive ETL, and Dynolog provides best-effort actuation with no consistency guarantees. Recent ML training systems increasingly report the need for real-time mechanisms to address straggler localization [14–19], fault diagnosis [20, 21], and training correctness [22], but each rebuilds subsets of collection, aggregation, and control from scratch. A common substrate would allow detection strategies to be expressed as analytics queries over telemetry, composable and deployable without rebuilding the infrastructure beneath

them. We compare these systems in more detail in §5.6.

5.1.2 Existing Approaches and Gaps

Monitoring. Monitoring enables basic visibility via metrics captured at sampled intervals, but lacks the workload context for causal attribution. Prometheus [169] is a time-series database for system counters, while LDMS [68] extends coverage for application-level statistics such as Kokkos [170]. They can not support synchronized on-demand collection necessary for views such as Fig. 5.1.

Profiling and Tracing. Tools such as TAU [28], Score-P [30], and Kineto [29] can provide both high-level aggregates or fine-grained telemetry via traces, but their workflows are post hoc by design. Traces can characterize short runs, but cannot provide always-on visibility over long-running jobs. In the absence of any real-time feedback to guide collection, the post hoc paradigm also incentivizes overcollection, leading to large datasets and long delays. Formats like OTF2 [32] are not designed for analytics, and visual inspection for anomalies is impractical at scale with billions of events.

Dataframe Analytics. Recent tools such as Caliper [35], DFTracer [34], Pipit [83], and Pytorch HTA [168] have adopted Pandas [171] and Dask-based [172] dataframe analytics interfaces to analyze telemetry, but require expensive ETL [36] to parse entire traces written as per-rank logs in plaintext formats — the working set for which scales with *ranks* $\times$ *timesteps*. Columnar formats like Parquet [41] offer efficient post-hoc analytics, but do not by themselves address overcollection, working set scaling, or real-time feedback, and remain largely unadopted in tracing workflows. In a simple internal port of HTA analyses from JSON to Parquet, we observed 12$\times$ faster data loading and 8$\times$ faster end-to-end analysis.

Streaming and In-situ Analytics. ADIOS [82] enables fixed-function in-situ workflows [66] on scientific data, typically run as another MPI application. MRNet [78, 173] is a tree-based overlay used as a reduction backend for commercial debuggers, but provides a compiled packet-based programming model. Neither enable runtime-programmable dataframe-style analytics needed for real-time BSP observability. Cloud streaming systems such as Flink [38] and Kafka [37] are designed for independent event streams over IP networks. They lack fabric-aware transport, BSP-aligned coordinated collection, and control semantics, precluding views like Fig. 5.1 without expensive off-fabric aggregation.

5.1.3 A Template for Dynamic Observability

In our AMR study, eBPF [135] was decisive for diagnosing the pathology in Fig. 5.1 after conventional tools were exhausted. Hypotheses about interrupts and scheduler effects were tested and discarded in hours through dynamic probes with programmatic aggregation and low overhead. But eBPF is node-local, and our workflow involved ad-hoc orchestration, uncoordinated

collection, and manual post-hoc correlation. Extending this to workload scale requires a substrate for real-time *feedback*, treating observability not as distributed logging, but as materializing views directly from running applications. But visibility alone is insufficient for steerability — interventions, whether additional views or parameter adjustments, require a *control plane* back to the running application. The next section states the requirements such a substrate must satisfy.

5.2 Requirements

Our goal is to accelerate the full spectrum of observability workflows, from offline trace analysis to real-time diagnosis. This requires analytics to transform fine-grained telemetry in situ, terminating in aggregated views or detailed traces but reorganized for efficient offline analyses. Additionally, control semantics are needed for interventions ranging from steering application views to steering the application itself. We model BSP as consisting of timesteps punctuated by synchronous collectives [43, 44], with timesteps containing one or more collectives. Steerable observability for this model must support three key requirements.

BSP-Aware Semantic Guarantees. Both the data and control paths must be synchronization-aware. On the data path, both transport and analytics must offer consistency guarantees, as all telemetry for a timestep must be collected and analyzed as a single unit. Control inputs must similarly be applied consistently and atomically at timestep boundaries across all ranks. While uncoordinated updates may be tolerable in some cases, they break correctness for interventions such as hyperparameter changes — a timestep-consistent control plane generalizes across use cases.

Low-Cost, Low-Latency Feedback. For always-on view-driven observability to be viable, overhead must be low and proportional to view complexity. Induced jitter must be minimized, so that continuously monitored aggregates incur minimal cost, while detailed traces remain available on demand. Analytics must execute in real-time, as feedback latency bounds intervention latency.

Declarative and Flexible Analytics. No predefined set of analyses can anticipate every pathology. Operators must be able to specify views and collection policies via a programmable interface at runtime, without recompilation or restart. Collection and analytics must be schema-agnostic, enabling visibility into any layer of the runtime stack — from MPI and fabric libraries to the application itself.

At scale, both data and control paths necessitate distributed execution, requiring a careful balance of consistency guarantees with asynchrony for low overhead. Our design leverages modern columnar primitives (Arrow [39], Parquet [41]), an extensible query engine called DataFusion [42], and RDMA for low-overhead transport, with a tree-based topology for reductions and broadcast.

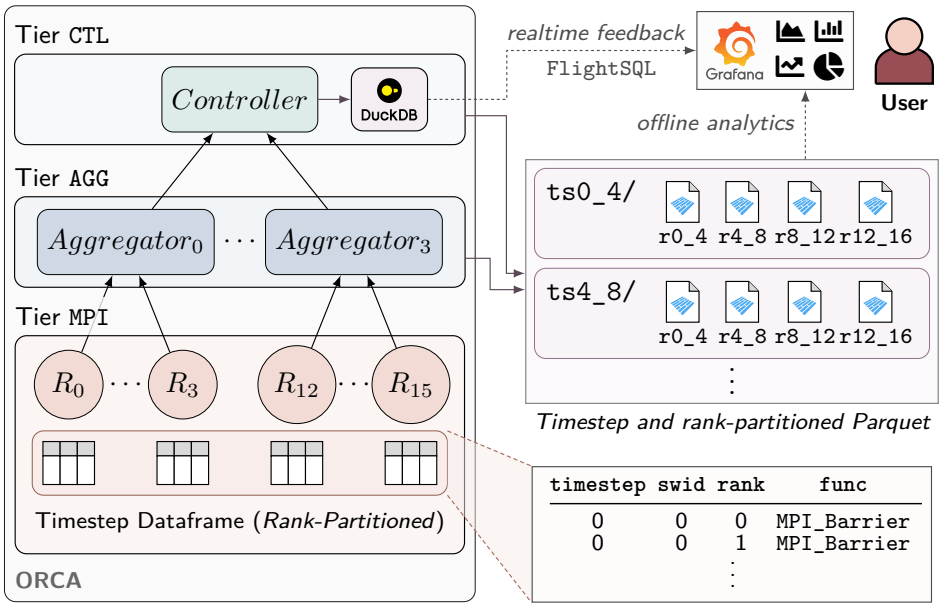


Figure 5.2: *ORCA architecture. A three-tier TBON with application ranks (MPI) at the leaves, intermediate aggregators (AGG), and a controller (CTL) at the root. Telemetry originates as columnar timestep dataframes, is transformed by SQL plan operators at each tier, and persisted via DuckDB and Parquet sinks — enabling workflows from real-time feedback via lightweight metrics to zero-ETL post-hoc analytics.*

5.3 Design

ORCA (Fig. 5.2) is a programmable *tree-based overlay network* (TBON) with a controller (CTL) at the root and aggregators (AGG) as the intermediate scale-out tier, attached to application ranks (MPI) as the leaf tier. CTL and AGG run on dedicated nodes outside the MPI domain and communicate with the application via on-fabric RPCs. The TBON provides asynchronous data and control planes. Data originates as columnar Arrow [39] streams at leaf ranks, flows through the TBON, and is persisted via sinks. Two sinks are currently implemented: Parquet [41] for high-volume telemetry, and DuckDB [174] for real-time metrics streamed to dashboards. We present ORCA as a progression of capabilities that successively shorten the observability feedback loop:

- Timestep-consistent, low-overhead collection is provided via a timestep-based data model (§5.3.1) along with an RDMA-based transport (§5.3.2). By themselves, they write timestep-partitioned traces, enabling a workflow similar to conventional tracers, but more efficient to collect and analyze.
- A SQL-based interface called *OrcaFlow* (§5.3.3) provides declarative in-situ analytics over data en-route to sinks, with automatic operator pushdowns to minimize data movement.
- Timestep-consistent online interventions are provided via a speculative two-phase commit variant called TS2PC (*timestep-linked two-phase commit*), ensuring atomic commit across all ranks at the same timestep (§5.3.4).

5.3.1 Data Model: Timestep Dataframes

ORCA workloads emit telemetry via named and *typed* streams generating *timestep dataframes*, which are the logical unit of collection, analysis, and persistence. Applications can have any number of streams, each generating events with a fixed schema. Streams are flushed at the end of each timestep to form these dataframes, comprising all events from that timestep. Timestep dataframes are logical constructs consisting of rank-partitioned shards that are transported over the TBON asynchronously, but analyzed and persisted as a single unit. Timesteps can contain multiple collectives, dividing execution further into causally-independent *synchronization windows*, assigned unique monotonic identifiers called `swid`s.

Timestep dataframes enable efficient analytics by capturing the causal structure of BSP execution. At blocking collectives, all ranks wait for stragglers and execution resets. Each timestep is therefore, naturally, an independent analysis unit. Pathologies are first attributed to a straggler rank that delayed a collective, then to events in the dataframe that led to that straggler. This mirrors the evolution of distributed tracing in loosely-coupled microservices — from treating traces as bags of events [175] to capturing causal edges between events via *request IDs* [163, 164]. For BSP workloads, this timestep-centric dataframe abstraction confers two analytical advantages:

- In transit, data can be represented as Arrow [39], whose contiguous columnar layout enables efficient serialization into RDMA-registered memory, zero-copy deserialization and in-situ analytics, while types such as dictionary arrays map naturally to repeated symbols that occur frequently in trace data.
- At rest, data is persisted as *timestep-partitioned* Parquet [41] (see Fig. 5.2), a columnar analytics-oriented format and a natural counterpart to Arrow. It not only enables zero-ETL analytics, but also reduces data read via rowgroup statistics and operator pushdowns.

Conventional tracers [28–30, 34, 35] not only employ ETL-heavy formats [31, 32], but create rank-major layouts that fragment a single timestep across thousands of per-rank logs. ORCA's partitioning aligns the primary key of the storage layout with the causal structure of BSP telemetry. Among prior work, Hatchet [40] postprocesses traces to build callflow graph indexes. Callflow graphs are a secondary index for BSP telemetry, complementary but optional when the primary key (`timestep, rank`) is effective.

5.3.2 TBON-Optimized RDMA Transport

ORCA's transport layer is designed to move Arrow dataframes asynchronously over an RDMA fabric across TBON tiers. Application overhead is minimized via an incast-optimized, receiver-driven mechanism that avoids synchronized pushes and overlaps data movement with application execution. As the CTL and AGG nodes are fully asynchronous and throughput-optimized, fabric QoS features [176] can be used to relegate ORCA traffic to a lower priority class, relying on idle fabric capacity for low-perturbation transfer.

The transport layer is agnostic to how telemetry is generated. Different profiling mechanisms may accumulate telemetry into per-stream *collectors* — local buffers that emit Arrow dataframes when flushed. After each timestep, all collectors on each rank are flushed to generate per-stream Arrow dataframes, which are then serialized into RDMA-registered memory. Ranks then send an RPC to their aggregators containing RDMA descriptors for the serialized data. Ranks then resume execution, while aggregators collect these dataframes via asynchronous RDMA GETs. After multiple iterations, we developed three key mechanisms for stable TBON operation:

- Aggregators must issue RDMA GETs at controlled concurrency to prevent catastrophic throughput collapse, as NICs have limited resources to track active RDMA operations (1–2 on older fabrics). The pull-based model, which resembles receiver-driven flow control, lets aggregators stay in control of data volume and is essential for stable incast operation.
- Because the collection pipeline yields dataframes in timestep order, it is vulnerable to head-of-line blocking from missing shards, risking *out-of-memory* errors. We address this with a priority queue at aggregators that prioritizes lower timesteps.
- While the system must be provisioned with adequate bandwidth to ensure minimal backpressure, flow control is essential for stability. Aggregators communicate backpressure by withholding RPC ACKs, and ranks block if outstanding RPC count exceeds a configured limit.

For offline analyses, aggregators compress and persist data at preconfigured timestep intervals. As the incoming data is already partitioned by timestep and rank, the TBON simply preserves this partitioning.

5.3.3 In-Situ Analytics for Real-time Feedback

The timestep dataframe abstraction also enables zero-copy in-situ SQL analytics at every TBON tier, enabling low-latency materialization of views and drastically reducing persisted data volume. OrcaFlow (Fig. 5.3) is a SQL-based programming interface that turns the TBON into a programmable reduction tree, transforming dataframes en route to sinks — currently partitioned Parquet for offline analysis and DuckDB for real-time metrics accessed by dashboards [177]. For analyses difficult to express in SQL, *map* operators allow invoking arbitrary precompiled kernels.

OrcaFlow is backed by a TBON-aware distributed planner and query engine built on Apache DataFusion [42]. Each flow compiles into *tier plans* describing transforms to apply, data to forward, and sinks to execute locally. The planner automatically places both transforms and sinks, pushing operators that reduce data volume as close to the source as possible. MPI and AGG operate on rank-partitioned shards and execute per-partition operators: selection, projection, partial aggregation. Cross-partition operations like joins must run on CTL, the root aggregator. Sinks are placed on the lowest tier that produces their stream (no sinks on MPI, DuckDB only on CTL). Fig. 5.3 illustrates with two flows:

- *(Left)* A selective tracing flow for `MPI_Waits` $> 10\,\text{ms}$ pushes filters all the way to the MPI tier. Results are coalesced at AGG and written as Parquet. Most telemetry for pathologies like Fig. 5.1 gets discarded at source.

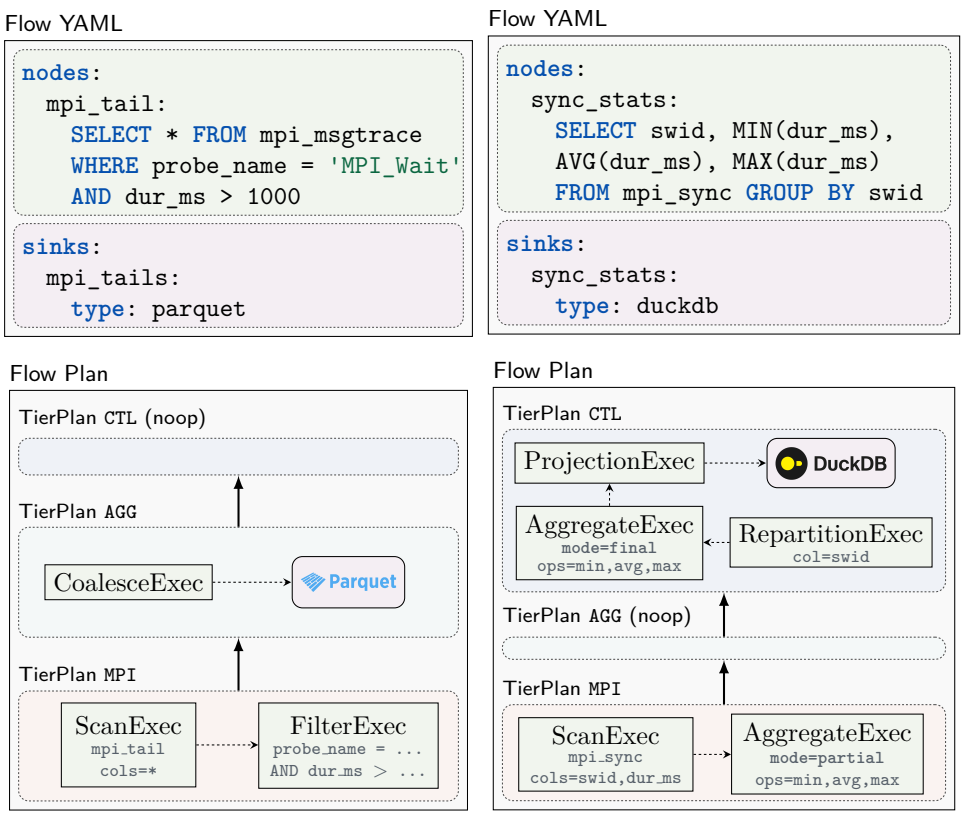


Figure 5.3: *OrcaFlow compilation showing flows and their compiled tier plans. Flows chain SQL transforms over telemetry streams en route to configurable sinks. Both operators and sinks are placed automatically. (Left) A selective filter pushed to MPI, discarding most data at source. (Right) Aggregations also reduce data volume — pushed-down partial aggregations emit fixed-size intermediates regardless of input volume.*

- *(Right)* An aggregation computing per-collective statistics splits into a partial aggregation at MPI and a final merge at CTL. Each leaf emits one row per group regardless of input volume — sketches and quantile digests fit the same model, producing fixed-size intermediates.

The programming model resembles batch analytics frameworks like Spark [178], but targets a multi-tier TBON, remains in-memory until sink ingestion, and avoids distributed shuffling to maintain predictable throughput. All tiers collect dataframes asynchronously but analyze them in timestep order. Per-partition compute budget scales with MPI and AGG, while cross-partition work is limited to a single node (CTL). This suffices for residual work after filtering and pre-aggregation. Post-mortems remain the preferred workflow for expensive cross-partition analyses, but still benefit from ORCA's in-situ filtering, preaggregation, and timestep-partitioned Parquet.

5.3.4 Control Plane for Steerable Observability

The control plane enables users to intervene in response to feedback, closing the loop between observation and action. Interventions range from steering collection to steering the application itself in response to observed behavior. Prior work has explored auto-tuning for specific use

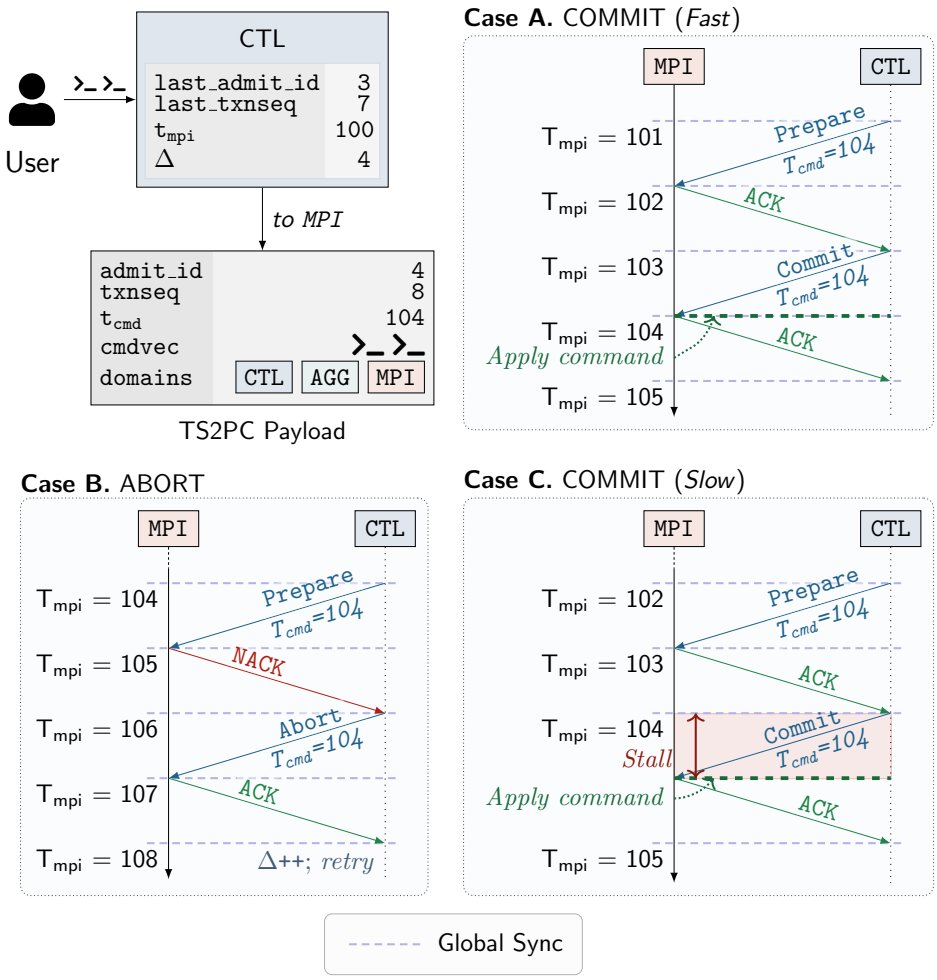


Figure 5.4: *Timestep-linked two-phase commit (TS2PC). Commands are proposed for a future timestep $T_{cmd} = T_{mpi} + \Delta$. (A) Normal case: ranks agree and apply the command before executing T_{cmd}. (B) Abort: ranks have passed T_{cmd} and vote to abort, command is retried with a larger Δ. (C) Stall: ranks have prepared but not received commit/abort. Execution blocks until resolution to preserve correctness.*

cases [179]. ORCA instead provides general mechanisms for timestep-consistent feedback and control, on which such policies can be built.

Requirements. The control plane must be asynchronous, as controllers — humans or automated agents — are not a part of the collective domain. It must also be atomic and timestep-consistent, i.e. commands must be acknowledged by all ranks before being executed, and they must execute on all ranks at the same timestep for correctness.

The TS2PC Protocol. ORCA implements the control plane as a replicated state machine [180] using a two-phase commit [181] variant called *timestep-linked two-phase commit* (TS2PC). Commands are opaque payloads — the protocol only decides when to commit and apply them in a timestep-consistent manner. As the controller is asynchronous with respect to the application,

it speculates on the timestep before which the command can be safely committed at all ranks by using a lagging view of application timestep (T_{app}) and adding an automatically learned margin called Δ. Commands are proposed for $T_{cmd} = T_{app} + \Delta$ and are buffered on all ranks to be executed at the beginning of T_{cmd}, if committed. The TBON assists the protocol by broadcasting requests and reducing responses — any `NAK` in the subtree results in a single `NAK` at the controller. Fig. 5.4 illustrates the protocol via three cases.

- *Case A (Commit):* Ranks agree to commit if they have not yet reached T_{cmd}, and apply the command just before T_{cmd}.
- *Case B (Abort):* If ranks are past T_{cmd}, they vote to abort. The controller processes the abort, increments Δ, and automatically retries the transaction. As a result, Δ quickly converges to the natural latency of the TBON.
- *Case C (Stall):* A subtlety arises when ranks have prepared but not yet received commit/abort by T_{cmd}. Correctness requires stalling execution until commit/abort is received, so the prepared command can be applied before T_{cmd}.

More implementation-specific details are discussed in §5.4.2.

5.4 Implementation

The core ORCA framework consists of 40K lines of code (80% C++, 20% Rust). ORCA uses the Mercury [101] RPC library for on-fabric communication via libfabric [182] and UCX [183] providers. The Rust portion consists of OrcaFlow's DataFusion-based query planning and execution on each node, while the bulk of the functionality — bootstrap, transport, and control — lies on the C++ side.

5.4.1 Implementation Overview

Integration. Two integration surfaces exist: the *OrcaClient* shim for applications, and collectors for profiling interfaces.

- *OrcaClient:* The *OrcaClient* shim links with the application and exposes a `PostTimestepAdvance()` hook, called after every timestep. ORCA can be activated at runtime by preloading the full library, otherwise these hooks become no-ops. `PostTimestepAdvance()` captures all data path work, including draining collectors, executing tier plans, dispatching data, and handling control messages.
- *Collectors:* Streams are implemented via collectors that accumulate events locally and emit Arrow dataframes when drained. At the end of each timestep, ORCA drains all collectors and enqueues the output for analysis. Our implementation currently packages basic collectors for MPI [43] and Kokkos [170]. Additional collectors may be defined by clients. Collectors can also implement dynamic instrumentation via interfaces like eBPF [135] and DynInst [184].

Overlay. The overlay bootstraps via a TCP-based protocol and switches to on-fabric RPCs after the initial handshake. While it supports additional topologies for use cases like intermediate fan-in reduction, only one is discussed for simplicity.

Data Path. The data path is completely asynchronous but timestep-ordered. Each tier executes its tier plans on dataframes in timestep order independently, and forwards only the relevant data to the next tier. Sinks are also a part of the tier plans generated by the controller, and are executed on the corresponding tiers after the plan nodes. While they may be scheduled on any TBON tier, current placement rules only permit placing sinks on CTL and AGG. Despite its asynchronous nature, flow updates across tiers also guarantee timestep-consistency.

5.4.2 TS2PC: Implementation Details

TS2PC is implemented as separate *inner* and *outer* layers. The inner layer enables a single transaction with a unique monotonic ID called `txnseq`, targeting a particular T_{cmd}. The outer layer adds features like automatic Δ-learning (discussed in §5.3.4) and *control domains* for cross-tier consistency.

Concurrency. Correctness necessitates that only one transaction be in flight at any given time, as intermittent failures and automatic retries can result in commands being committed out of order. However, a single transaction can contain an arbitrary number of commands applied as a batch.

Handling Pauses. The control plane supports pausing and resuming the application between timesteps for basic lifecycle management, and as primitives for more complex workflows. An interesting case emerges from the interaction of the paused state with TS2PC — resume commands in this state will be committed for a future timestep never reached by a paused application, creating a deadlock. Our solution recognizes that the paused state makes the application synchronous with the user and renders TS2PC unnecessary for consistency. Commands in this state circumvent the protocol and are executed directly.

Cross-tier Consistency. Commands such as flow updates trigger tier plan updates at all tiers, and must be applied at a consistent timestep across the entire TBON. This is challenging as the higher tiers are asynchronous and observe application timesteps at different times. A tier is defined to observe a timestep when it processes its dataframes.

Cross-tier consistency is supported via *control domains*, which make each overlay tier independently addressable by the control plane in a timestep-consistent manner. Each tier corresponds to a domain, and each transaction carries a bitmask specifying which domains it targets — flow updates, for example, target all domains. The inner protocol remains unchanged, with MPI making all commit decisions as the single source of truth on the application timestep, regardless of the domain. Higher tiers snoop on protocol traffic and apply commands only if their domain bit is set.

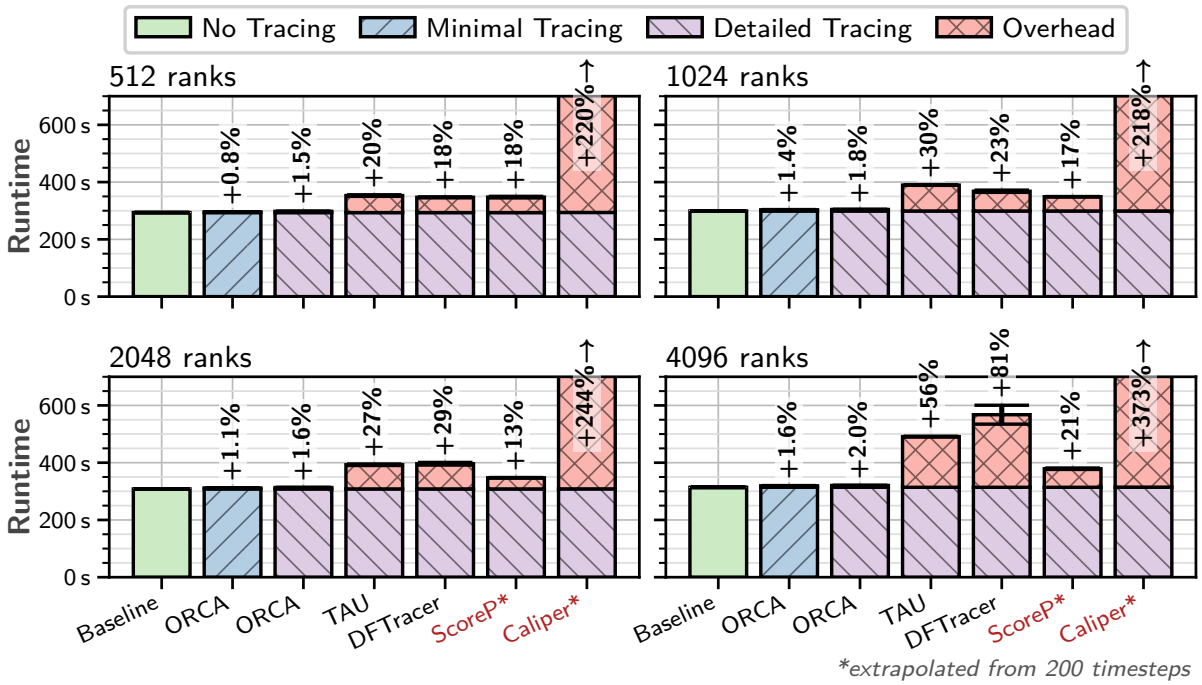


Figure 5.5: *Runtime overhead for a detailed tracing profile on an AMR code at 512–4096 ranks. An additional minimal profile (collective-only) for ORCA shows always-on overhead. ORCA maintains 0.8–2% overhead across both profiles, while TAU and DFTracer degrade from 18–20% at 512 ranks to 56–81% at 4096 ranks.* *extrapolated from 200 timesteps due to stability issues.

5.5 Evaluation

We evaluate ORCA by progressively enabling its capabilities. As a passive telemetry collector, we compare runtime overhead (§5.5.1.1), storage efficiency (§5.5.1.2), and analytics performance (§5.5.1.3) against state-of-the-art tracers. We then demonstrate online analytics (§5.5.2) and control workflows (§5.5.3) — capabilities with no existing counterparts.

Hardware Setup. Experiments run on a 600-node research cluster with QLogic Truescale fabric (Intel Xeon E5-2670, 16 cores/node, 64 GB RAM, 40 Gbps QLogic 7340 NICs) and a 20-node Lustre filesystem with 3GB/s aggregate bandwidth.

Software Setup. We use the Sedov Blast Wave 3D [185] AMR code from the Parthenon [85] and Phoebus [87] frameworks at 512–4096 CPU ranks, with 16 ranks/node. ORCA uses one aggregator node per 1024 ranks plus one controller node. All experiments use MVAPICH2 v2.3.7 [138] with a tuned PSM [186] library for our fabric. While simulation nodes use PSM, ORCA communication uses Mercury [101] over verbs because of fabric restrictions.

This setup represents a strenuous observability workload: CPU codes make every cycle visible as runtime overhead, the legacy fabric lacks QoS support and forces ORCA onto a non-native interface (verbs), and the code's multiple communication patterns — multiple collectives, halo exchanges, and frequent load balancing — make it highly jitter-sensitive.

5.5.1 Tracing Overhead and Efficiency

We compare ORCA in conventional tracing workflows with four state-of-the-art tracing tools (TAU [28], Score-P [30], Caliper [35], and DFTracer [34]) along three axes: runtime overhead, storage efficiency, and analytics performance.

Collection Profiles. ORCA is set up with two tracing profiles: a lightweight profile that only captures MPI collectives, and a detailed profile that also captures all MPI point-to-point messages and Kokkos events. Extremely high-volume events like `MPI_Test` and `MPI_Iprobe` are excluded. For ORCA, the lightweight profile measures the common-case overhead of always-on observability, while the detailed profile measures the worst-case overhead of capturing all MPI and Kokkos events. All other tracing tools are configured with the same detailed collection profile. Unless otherwise specified, all experiments run for 2000 timesteps, and we report the average runtime from three runs. Unlike tracing tools, ORCA is deployed with additional resources: one aggregator node per 1024 ranks, plus one controller node.

5.5.1.1 Runtime Overhead

Fig. 5.5 compares the runtime overhead of ORCA and other tracing tools for 512–4096 ranks.

Takeaway: *ORCA incurs an overhead of 0.8–2% across all scales and profiles, vs up to 56–81% for TAU and DFTracer.*

- ORCA remains within 0.8–2% overhead across all scales and profiles, while that of TAU and DFTracer degrades, from 18–20% at 512 ranks to 56–81% at 4096 ranks (Score-P and Caliper discussed separately below).
- At 4096 ranks, the 2% overhead is despite sharing the fabric with a real AMR code with no QoS support, transporting 1.8 GB/s of telemetry (550 GB on-wire, 111 GB after compression, over 308 s).
- Overhead increases only 0.4% between ORCA's lightweight and detailed profiles at 4096 ranks, despite 200× more telemetry (524 MB vs 111 GB), enabled by Arrow's serialization efficiency and a TBON-optimized RDMA transport.

Why ORCA Works. TAU and DFTracer degrade steadily because they flush trace buffers to storage as they fill, which produces uncoordinated flushing patterns. This, coupled with a kernel-based data path for storage, introduces jitter that compounds at scale. ORCA, in comparison, uses an inherently efficient format, serializes synchronously into RDMA-registered buffers, and transports via efficient RDMA GETs.

Caveats on Score-P and Caliper. Because of difficulties running them for the full 2000 timesteps, all numbers for Score-P and Caliper are extrapolated from 200 timesteps. Both tools are designed for short characterization runs where data is buffered in memory and flushed once at the end. Score-P also requires an additional workload pass to estimate trace buffer sizes, and frequently crashed with out-of-memory errors despite generously configured buffer sizes. While Caliper

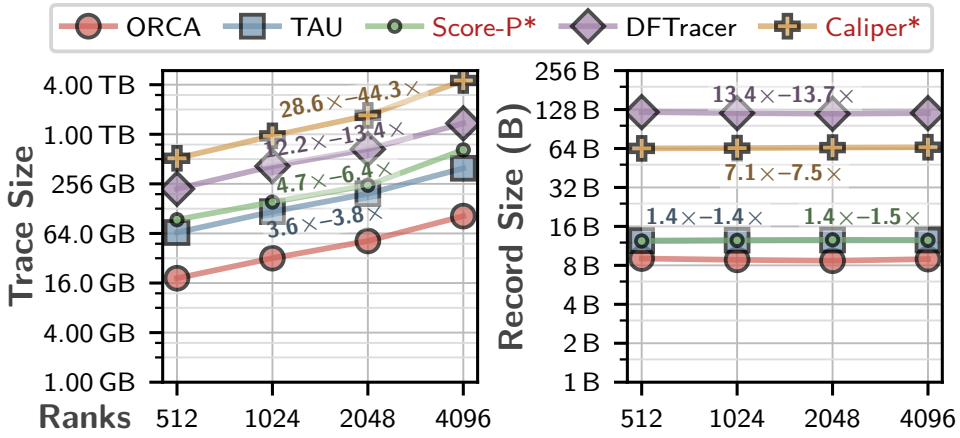


Figure 5.6: *Trace size (left) and per-record size (right) comparison for detailed profile. ORCA traces are 3.6–44× smaller than other formats. Per-record size isolates format efficiency from record amplification — other tracers emit 2.5–3.3× more records than ORCA for the same collection profile.*

supports periodic flushing, the path was buggy and performance was significantly degraded even after patches.

5.5.1.2 Storage Efficiency

Fig. 5.6 compares the total size, and the per-record size of the traces emitted by ORCA and other tracers. ORCA writes Parquet, TAU and Score-P are configured for OTF2, DFTracer writes `jsonl`, and Caliper writes its custom plaintext `cali` format. Key findings:

Takeaway: *ORCA's Parquet traces are up to 6.4× and 44.3× smaller than OTF2 and plaintext formats, respectively.*

- ORCA consistently emits the smallest traces, 3.6–3.8× smaller than TAU, and up to 44.3× smaller than Caliper.
- Differences can be partly explained by format efficiency. Per-record footprint of OTF2, being a binary format, is only 1.4× larger, while that of plaintext formats is 7–13× larger.

Record Amplification. Accounting for format efficiency, other tracing tools emit 2.5–3.3× more records than ORCA, despite the same collection profile.

1. Most tracers generate separate entry and exit records per traced function call, while ORCA and DFTracer generate one record containing both start time and duration. Latter is both more space-efficient and allows duration-based filters without additional preprocessing.
2. TAU incurs additional amplification from hardcoded derived events: for every MPI send and receive, it emits a function event, a message event, and an event for messages received during certain phases such as `MPI_Wait`.
3. Flush-induced jitter with Score-P and Caliper creates stragglers, resulting in additional wait events from other ranks.

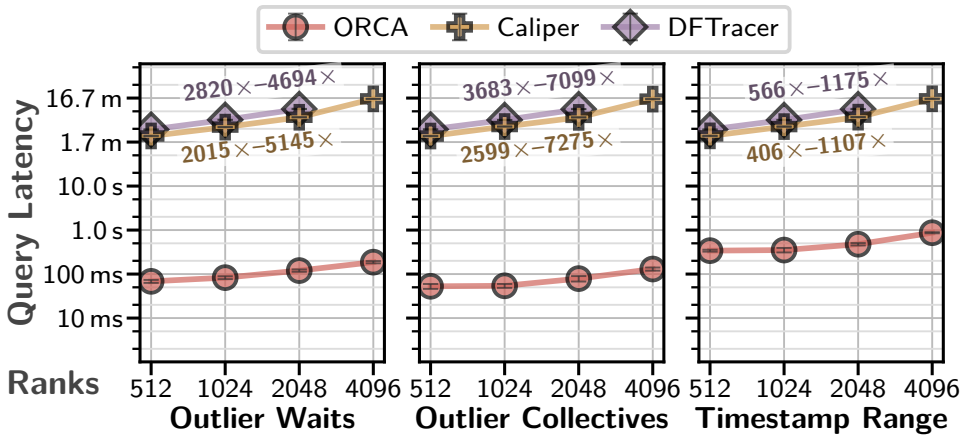


Figure 5.7: *Post-mortem query performance. ORCA completes queries in hundreds of milliseconds, 3–4 orders of magnitude faster than DFTracer and Caliper. Performance for plaintext formats is dominated by parsing, not query complexity. TAU and Score-P are omitted as OTF2 is not a queryable format.*

Why ORCA Works. ORCA's storage efficiency stems from two design choices. First, Parquet yields an inherently compact representation through binary columnar layout, dictionary encoding, and Snappy [187] compression. Second, separating derived events from collection — computing them via OrcaFlow rather than baking them into the tracer — prevents record amplification from creeping in as features accumulate. We discuss compression further in §5.5.1.4.

5.5.1.3 Query Performance

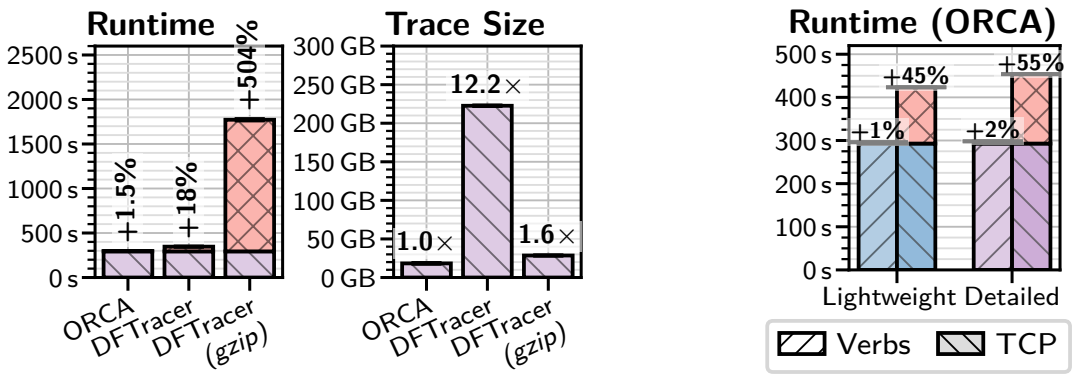
We evaluate analytics performance using a suite of post-mortem queries (Fig. 5.7). All queries run on traces from a smaller 20-timestep run, executed cold with caches cleared, latencies averaged over 3 runs.

Takeaway: *ORCA traces enable 3–4 orders of magnitude faster analytics vs. DFTracer and Caliper.*

Queries. We use three queries of increasing complexity. *Outlier Waits* is a simple filter identifying `MPI_Wait` events exceeding 1 ms. *Outlier Collectives* aggregates collective events by `swid` and filters for outliers (max duration $>$ 10 ms). *Timestamp Range* returns all events within the first second of execution—a query that might power a visualization frontend, and notably not a dimension ORCA partitions on.

Analytics Tools. All queries execute on a single 16-core node. DFTracer and Caliper queries use their bundled Python-based analytics libraries. ORCA queries use an off-the-shelf query engine [188] with no partition awareness for simplicity. TAU and Score-P are excluded as OTF2 does not support dataframe analytics. Findings:

- ORCA queries complete in hundreds of milliseconds, 3–4 orders of magnitude faster than DFTracer and Caliper. The gap widens with scale.
- Query times for DFTracer and Caliper do not vary with query complexity. This is because performance is bound by parsing, not computation — entire plaintext traces need to be parsed and loaded as dataframes for queries.



(a) *Compression impact.* ***(b)*** *Transport impact.*

Figure 5.8: *(Left) Compression reduces trace size but induces severe jitter on simulation ranks. ORCA avoids this by offloading compression to dedicated aggregators. (Right) TCP transport increases ORCA overhead from 1–2% to 45–55%. Arrow-RDMA co-design is essential for low overhead.*

Why ORCA Works. Plaintext formats used by analytics-oriented tracers [34, 35] require expensive ETL to materialize dataframes before any analyses, while Parquet's columnar layout enables zero-ETL analytics. Additionally, ORCA's layout enables both cross-file and intra-file pruning via partitioning and rowgroup statistics. The outperformance in Fig. 5.7 is despite the query suite not exploiting the former — unmodified Polars with lazy execution must scan all footers. Open table metadata formats [123, 189] would accelerate queries further by adding a catalog layer over Parquet, enabling standard query engines [174, 188] to automatically leverage ORCA's partitioning. Conventional formats offer no such benefits, requiring larger analytics clusters for ETL as traces grow.

5.5.1.4 Tradeoff Analysis

We now isolate the impact of key architectural decisions using a 512-rank run (Fig. 5.8).

Takeaway: *Architectural choices — dedicated nodes and RDMA — are key to compression and low runtime overhead.*

Compression. DFTracer supports gzip compression, which reduces its trace size from $12.2\times$ that of ORCA to $1.6\times$ (fig. 5.8a), but at the cost of increasing runtime overhead from 18% to 504%. Compression on MPI ranks is unviable because it competes with the application for CPU, amplifying jitter. ORCA offloads compression to dedicated aggregator cores, keeping it off the critical path. Additionally, most compressed formats cannot be queried without full decompression, while ORCA queries only require decompressing the files and rowgroups that match query predicates.

Transport Layer. Switching from `verbs` to libfabric's `TCP` provider increases ORCA's overhead from 1–2% to 45–55% (fig. 5.8b). Even with the same Arrow-based data path, TCP suffers because retrieval incurs CPU overhead on simulation ranks, introducing jitter. These results underscore the

Flow	Query
Slow Wait Counts	`SELECT COUNT(*) FROM mpi_wait` `WHERE duration > 1ms` → DuckDB
Slow Wait Traces	`SELECT * FROM mpi_wait` `WHERE duration > 1ms` → Parquet

Load Imbalance	`SELECT rank, SUM(duration) FROM kokkos`
	`WHERE name IN <compute kernels>`
	`GROUP BY rank` → `stats` → Parquet
	`PERCENTILES(stats, [50,90,99])` → DuckDB

Table 5.1: *In-situ workflows evaluated in Section 5.5.2, ranging from lightweight metrics and selective tracing on the communication path to aggregates over compute kernels.*

cost of TCP-based transport for tightly-coupled BSP workloads. While GPU codes may have more tolerance for the transport overhead of conventional event-based streaming systems [37, 38], they still lack the columnar, causally-partitioned data path that underpins ORCA's analytical efficiency.

5.5.2 ORCA-Native In-Situ Workflows

While it can enable conventional tracing workflows, ORCA is fundamentally designed for in-situ analytics — directly materializing views from a running application. We now evaluate these capabilities, measuring the runtime overhead of operator pushdown, and its impact on data movement.

Takeaway: *In-situ analytics and operator pushdown reduce data movement by 98.5–99.999%, at sub-2% overhead.*

Workflows. Table 5.1 shows three flows of increasing complexity, designed for AMR codes. The first demonstrates a monitoring workflow: measuring the count of `MPI_Wait` exceeding 1 ms each timestep, while the second persists outlier events to Parquet. The third flow tracks compute load imbalance in AMR codes — it filters for known compute kernels, computes duration aggregates by rank, and streams details to Parquet and percentiles to DuckDB for real-time dashboards.

Observations. Fig. 5.9 shows the runtime overhead and the data reduction at MPI for the three flows.

- All workflows incur an overhead of $\leq 2\%$ across all scales despite executing operators on CPU ranks, illustrating the efficiency of Arrow-based in-situ analytics. Computation budget for pushdown only grows with modern GPU codes.
- Data movement from MPI ranks is reduced by $98.5\% - 99.999\%$. Designing observability around materialized views reduces data movement and improves feedback latency. Most telemetry is redundant for monitoring and debugging — a small fraction of aggregated and filtered data suffices to materialize the required views.
- The `MPI_Wait` flows do not sample—they perform continuous predicate evaluation for every single `MPI_Wait`, emitting only outliers. Comprehensive continuous observability of this type is impractical without pushdown.

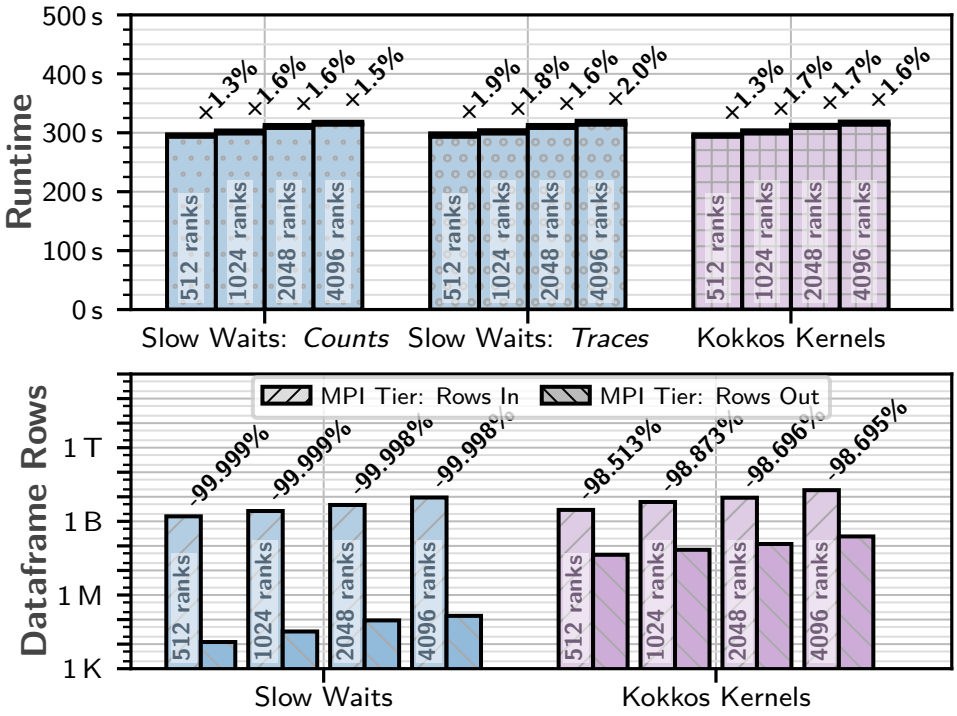


Figure 5.9: *Runtime overhead (top) and MPI tier data reduction (bottom) for in-situ analytics with operator pushdown. Flows range from monitoring communication outliers to compute load balance. Pushdown discards 98.5–99.999% of data at sub-2% overhead, despite operators executing on MPI ranks.*

5.5.3 Closing the Loop: Agentic Debugging

While ORCA was designed for human operators, the emergence of LLM-based coding agents, capable of issuing commands and analyzing outputs, offers a way to validate the steerability paradigm. We task an LLM agent with localizing the 4096-rank AMR anomaly shown in Fig. 4.2.

Takeaway: *Steerable collection and in-situ analytics compress a months-long manual investigation into 27 minutes.*

Setup. Claude Code (Opus 4.5) [190] ran on the head node with access to the running application instance via ORCA, and a copy of the codebase. The agent was asked to monitor only collective statistics initially, collecting additional Kokkos/MPI telemetry only if outliers were detected. The LLM wrote and installed all ORCA flows and used Polars [188] on the head node for all analyses.

Timeline. Fig. 5.10 shows the debugging session: user and LLM interactions at top, per-timestep event volume at bottom. After a warmup phase, the LLM began the localization exercise at 272 s by monitoring collectives for outliers (①).

1. At 384 s, it analyzed collective percentiles, identified stragglers, and installed a flow for Kokkos events > 1 ms (②).
2. Between 510–1000 s, it used the Kokkos events to identify a `map.clear()` in AMR redistribution as the culprit.

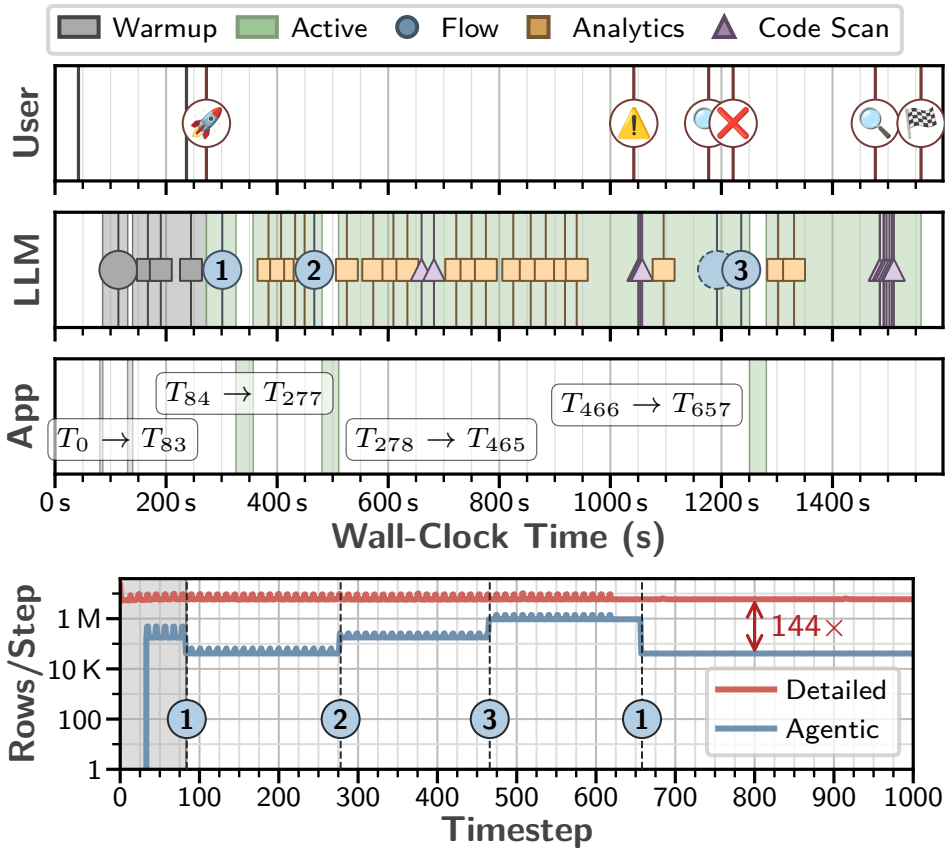


Figure 5.10: *Visual transcript of an agentic debugging session demonstrating closed-loop capabilities. An LLM agent is tasked with localizing the anomaly from fig. 5.1 using high-level metrics, progressively collecting targeted telemetry to identify the source. The agent operates largely autonomously, producing the complete call path in 27 minutes. Reverting to baseline monitoring reduces data volume by 144×.*

3. At 1042 s, user rejected this as inadequate given the magnitude of the anomaly (50–100 ms). LLM then found a `MPI_Wait` in a buffer destructor triggered by the `clear()`.
4. At 1191 s, LLM proposed capturing all `MPI_Wait` events to confirm, user rejected and asked for a duration-based filter. LLM then installed a flow for `MPI_Waits` > 1 ms.
5. At 1335 s, telemetry confirmed waits as the root cause. LLM reported the complete call flow at 1605 s. Application then resumed with baseline collective monitoring (①).

Observations. The agent autonomously correlated high-level telemetry with code, requested targeted detail, and traced the root cause to where instrumentation ended — completing in 27 minutes what took months of manual investigation. Reverting to baseline monitoring afterwards reduced data volume by 144×. User interventions were minor: rejecting an incomplete explanation and asking for basic filtering. While the exercise relied on pre-existing annotations in relevant regions, dynamic instrumentation [135, 184] would enable investigations to continue into the MPI and fabric layers. Beyond localization, the same anomaly signature can be deployed as an always-on metric, allowing regressions to be detected immediately.

5.6 Discussion and Related Work

5.6.1 Discussion: Guarantees, Perturbation Bounds, and Tradeoffs

We now revisit ORCA's design to assess its guarantees, costs, and tradeoffs.

Rationale for Design Guarantees. ORCA is best understood as a programmable network for BSP-aware in-situ trace analytics, not as a conventional database for time-series data [169, 191]. While databases typically persist before analytics, ORCA enables in-situ workflows over a tree-based overlay before ingesting processed and reduced output. The closest analogs are *Data Acquisition* (DAQ) pipelines in particle accelerators [192, 193], where FPGAs discard the bulk of sensor data at source, retaining only high-signal telemetry. Its data and control plane guarantees are directly informed by observability requirements given BSP execution.

- On the data path, the goal is to enable causal analyses in Lamport's happens-before sense [144], from symptoms to root cause. Symptoms, whether correctness or performance, materialize at global synchronization boundaries where global state becomes meaningfully defined. Understanding the root cause of behavior surfacing at these boundaries requires traversing causal edges backwards, from symptoms derived from global state to local events capturing their root causes. A single timestep dataframe captures the complete set of events needed to undertake these analyses — those derived from global state surface symptoms, while local events surface the root causes. Note that the timestep dataframe abstraction by itself does not guarantee adequate instrumentation depth, it only provides a generalized windowing construct for correctness and performance analyses.
- Control semantics similarly do not create new guarantees, but only preserve pre-existing ones in BSP execution. Hyperparameters within a BSP workload are strongly consistent across ranks. Updates to these hyperparameters are typically coordinated by the lead rank via `MPI_Bcast()`: a synchronous collective that blocks execution until all ranks receive updates. However, human operators and other agents attempting to interactively modify these parameters are asynchronous entities outside the application domain. Any steering inputs that affect execution must be propagated in a timestep-consistent manner, a guarantee ORCA provides with TS2PC.

Perturbation Sources and Impact. All measurement necessarily imposes some perturbation, and is useful insofar as the induced noise does not obscure the observed signal. Perturbation in ORCA arises from either instrumentation, transport, or analytics. It manifests either as easily measured foreground costs within the `PostTimestepAdvance()` hook, or asynchronous work during application execution, which is harder to observe directly. As its magnitude is highly context-dependent, we now discuss the circumstances under which each source materially perturbs execution.

- **Instrumentation.** At its core, ORCA is a streaming analytics framework and agnostic to instrumentation techniques. Different instrumentation techniques offer different tradeoffs — compiled tracepoints are cheap but lack runtime flexibility, while eBPF `uprobes` enable

dynamic instrumentation but incur a relatively expensive kernel trap per invocation [194]. Costs are dependent on the technique used and the functions instrumented, and tracing hot inner-loop functions may impose unbounded overhead incurred inline during the execution. ORCA's online control capabilities enable deactivating expensive probes instantaneously if found to be destabilizing.

- **Transport.** Transport effects may show as Arrow serialization costs or backpressure-driven stalls within the hook, or `RDMA GET`-induced jitter mid-timestep. Arrow serialization is efficient as it involves contiguous buffer `memcpy`'s, and backpressure does not trigger if aggregators are adequately provisioned for workload requirements. `RDMA GET` jitter is low as our experiments show, and can be lower with modern fabrics' QoS capabilities [176].
- **Analytics.** Analytics costs manifest as pushed-down kernels executed on ranks within the hook, and are proportional to kernel complexity. Our experiments show that these costs are negligible for a large class of observability workflows even with highly contended CPU ranks, and the headroom only increases with GPU-based codes with relatively idle CPU cores. However, there is no upper bound on perturbation for arbitrary pushed-down kernels. The planner can be augmented to reactively adjust placement in response to observed performance impact.

Overall, ORCA's architectural and workflow capabilities enable realizing any given observability task with costs approaching fundamental hardware limits. Architecturally, Arrow enables highly efficient columnar analytics with SIMD kernels and cheap (de)-serialization, RDMA enables low-overhead transport, while pushdowns minimize data movement required to materialize views. At a workflow level, steerability enables dynamically instantiating only the workflows necessary at any given moment.

Architectural Limits of Auxiliary Analytics Tiers. While elastic, higher TBON tiers typically have lower total I/O bandwidth compared to the application. ORCA currently never places sinks on MPI — while the right default for observability, this may be suboptimal when the higher I/O bandwidth of the application tier is genuinely necessary, such as during checkpointing. For observability, offloading sinks minimizes application jitter during the compute phase, but during synchronous checkpoints, ranks are idle, jitter is not a concern, and maximizing I/O bandwidth is more important. ORCA's planner can be extended to be phase-aware to accommodate streaming analytics use cases beyond observability, and allow placing sinks on MPI when I/O bandwidth becomes a bottleneck.

Columnar Transpose Tradeoffs. Unlike event-based streaming systems, ORCA maintains a columnar structure end-to-end from generation to persistence, but this may not be optimal for some high-volume, cache-sensitive functions. Traces are intrinsically events, which ORCA transposes into per-column buffers at generation time. As opposed to `memcpy`-ing an entire struct into a *row-major* layout, this involves small random writes to multiple cache lines to produce a *column-major* layout, which may perturb some cache-sensitive kernels. The alternative is a bulk transpose, where events are accumulated in a row-major format, and transposed in a bulk

System	Diagnostic Task	Their Approach	ORCA Equivalent
Greyhound [15] *(Fail-slows)*	• Iteration boundaries • Fail-slow detection • Straggler localization • Cause identification • Online interventions	• Autocorrelation over collectives • Track iteration times • Aggregate NCCL events (off-fabric) • Microbenchmarks • Tune micro-batch distribution • Tune parallelism	• `PostTimestepAdvance` • Track iteration times • `GROUPBY swid` (on-fabric) • Microbenchmarks or trace events • Supported via control plane
Flare [16] *(Training anomalies)*	• Stall detection • Stall disambiguation • Slowdown analysis	• Call stack analysis • Periodic metrics	• In-situ reductions • In-situ trace event aggregation
Holmes [14] *(Fail-slows)*	• Outlier collectives • Localization	• Random forest model • Graph search	• Policy-dependent • Learn collective topology at bootstrap, `GROUPBY swid`, trace critical path in reverse
Mycroft [18] *(Training anomalies)*	• Straggler detection • Straggler localization	• Fine-grained trace NCCL events buffered locally, flushed to Kafka if heuristics trigger	• Aligned dataframes analyzed in-situ
TrainCheck [22] *(Training correctness)*	• Correctness invariants	• Offline trace analysis for invariant generation, online enforcement	• Timestep-aligned traces for offline invariant generation, in-situ enforcement via OrcaFlow
Reveal [19] *(Training anomalies)*	• Anomaly detection • Anomaly attribution	• Time-windowed metrics + unsupervised learning	• Workload-level trace events
Minder[#] [20] *(Fault diagnosis)*	• Fail-slow hardware	• Cluster-level metrics and anomaly detection	• Workload-level trace events
Aegis[#] [21] *(Fault diagnosis)*	• Fail-stop hardware	• Per-node NCCL counters pulled on timeouts	• Workload-level trace events
PerfTracker[#] [17] *(Training anomalies)*	• Anomaly detection • Anomaly attribution	• Always-on iteration monitoring • Trigger-driven detailed capture • In-situ telemetry reductions	• Identical workflow, workload-level

[#] *Minder, Aegis, and PerfTracker target multi-tenant environments. ORCA currently supports a single large workload.*

Table 5.2: *Diagnostic approaches in ML training observability and their ORCA equivalents.*

secondary pass before analytics, at the cost of an additional pass. In our benchmarks, bulk transpose offers no measurable benefits and imposes an additional traversal cost. However, this effect may be architecture- and application-dependent.

5.6.2 Related Work

ML Training Observability. Recent ML training systems build application-specific pipelines spanning straggler localization [14–19], fault diagnosis [20, 21], and correctness enforcement [22]. ORCA generalizes the pattern. Holmes [14] uses graph search over collective telemetry to find anomalies. ORCA's aligned collection makes stragglers surface through simple grouped aggregations, no complex detection needed. Mycroft [18] buffers fine-grained NCCL [44] telemetry locally and flushes to Kafka [37] on trigger, but notes the scalability limits of central logging. ORCA's tree-based overlay scales out naturally. TrainCheck [22] enforces training correctness via offline invariant generation and online checks. ORCA's timestep-aligned columnar analytics accelerate

both stages. Minder [20], Aegis [21], and PerfTracker [17] operate at the cluster infrastructure level, using hardware counters, metrics, and fine-grained trace events to identify faulty nodes across workloads. Their scopes range from identifying bad hardware across many workloads treated as black boxes to providing tuning recommendations to customers. ORCA provides high-fidelity visibility into a single workload but does not yet support cross-workload diagnosis in multi-tenant environments. Table 5.2 summarizes the diagnostic mechanisms implemented in these systems and their ORCA equivalents.

Among these systems, Greyhound [15] and PerfTracker [17] are the closest to ORCA in scope. Greyhound implements a complete feedback loop: iteration times are tracked, NCCL events are collected for localization when outliers are detected, microbenchmarks are dispatched to validate stragglers, and online tuning rebalances work in response. This is a fixed-function version of what ORCA expresses as composable operators. ORCA's timestep dataframe abstraction preserves fine-grained trace events, enabling disambiguation of stragglers without needing microbenchmarks for validation. Greyhound's control semantics require pausing the application, while ORCA implements an asynchronous control plane with atomicity and timestep-consistency guarantees. Additionally, ORCA's on-fabric broadcast tree enables faster message dissemination than a central controller. PerfTracker implements a two-level loop: lightweight always-on monitoring of iteration times, with detailed capture triggered by outliers. Node-level aggregation reduces 300 MB of telemetry to 30 KB of aggregated signatures. Long symbol names are noted as contributing to size amplification, which ORCA avoids via dictionary-encoded Arrow columns.

Tool Infrastructure and Tracing. MRNet [78] introduced TBONs for scalable tool infrastructure, but its compiled C++ filter model has not evolved with modern analytics. Fay [195], Snicket [196], and Pivot Tracing [197] pioneered causality-preserving query-driven tracing in the context of asynchronous services. ORCA extends this paradigm to BSP, adding consistent control, open columnar formats, and an RDMA TBON. Dynamic instrumentation via eBPF [135] and DynInst [184] can extend reach into runtime and kernel layers. Sketch-based telemetry reduction [198] aligns with ORCA's in-situ aggregation model.

Computational Steering. Computational steering—interactive control of running simulations—dates to the early 1990s [74, 75] and continues through modern in-situ visualization frameworks [62, 65], but remains realized as fixed-function pipelines over domain-specific data models. ORCA generalizes this as a workload-agnostic framework using columnar formats and relational operators, with asynchronous and timestep-consistent control. ORCA's motivation is telemetry analytics, but its applicability to broader steering use cases tracks Arrow's adoption in scientific computing. Particle data already maps naturally to Arrow [199, 200], and bridges to array-based formats are emerging [201, 202]. As these ecosystems converge, the same infrastructure subsumes workflows from observability to in-situ visualization and indexing.

Chapter 6

Conclusions and Future Directions

Contents

This thesis argues that a large class of efficiency bottlenecks in BSP systems arise from conditions unknowable before runtime, and that such conditions are systematically addressable through adaptive mechanisms if the requisite feedback loops exist.

- In CARP (Chapter 3), the loop is policy-driven: a static renegotiation policy periodically reconstructs a global view of attribute distributions and uses it to drive adaptive range partitioning as distributions evolve.
- In the AMR study (Chapter 4), the loop was human-driven: placement effects were repeatedly confounded by fail-slow hardware, fabric behavior, and tuning issues, and progress depended on us manually constructing views from fine-grained BSP telemetry to isolate and localize these confounders. The resulting slow, rerun-driven diagnostic process illustrates the difficulty of addressing similar anomalies in production environments.
- ORCA (Chapter 5) enables a programmable feedback loop via a SQL-based on-fabric analytics pipeline that materializes BSP telemetry views through in-situ reduction, coupled with timestep-consistent control semantics for coordinated runtime intervention. The loop driver is programmable and can range from human operators to automated agents.

Beyond observability, ORCA's design reflects a broader convergence between OLAP and HPC: columnar formats [39, 41], SQL planning, and embeddable query engines [42] applied to in-situ HPC analytics. Compared with prior in-situ systems, ORCA differs in two ways. First, instead of placing analytics entirely on application ranks [24, 25] or entirely on a dedicated tier [66], ORCA's planner decomposes them into operators that can be placed on either tier at runtime. Second, unlike ADIOS-style offload as a peer MPI job [66], ORCA's analytics tier is fully asynchronous while still preserving timestep-consistent semantics.

We conclude this thesis by recapping the steerable observability paradigm and its limits (§6.1), synthesizing broader lessons for scientific data management from CARP and ORCA (§6.2), describing extensions within immediate reach (§6.3), and outlining longer-term directions (§6.4).

6.1 Steerable Observability: Retrospective and Synthesis

Steerable observability is the ability to request runtime views of application state, reconstructed in real time, and to change those views on demand as diagnosis progresses. Conventional tracing is locked to a preselected fidelity that both overcollects and undercollects: despite paying for comprehensive collection at one layer, it can still lack the visibility into lower layers necessary for root-cause identification. In contrast, steerable observability materializes relevant views directly from the application in real-time. This drastically reduces collection and storage overhead through predicate and reduction pushdowns. It also shortens feedback latency, enabling drill-downs across arbitrary fidelity insofar as instrumentation permits, while cheap pushdowns enable continuous predicate evaluation for anomalies.

BSP Is Different. BSP workloads run as a single coordinated job, not a web of loosely coupled microservices. Global synchronization defines both the unit of work and the axes of causality and consistency. Attribution of pathologies is first to an anomalous rank, and then to local events within that rank. Root causes lie within a deeper layer of a single workload's runtime stack, not across microservices, requiring analytical substrates to preserve these causal edges. Event-based streaming systems [37, 38] treat events as separate in transit, requiring these dependency edges to be reconstructed after collection. Even with identical collection profiles and data formats, reconstructing these edges after persistence incurs costs from small random reads, analogous to how FastQuery compares with CARP in §3.6.3.3.

ORCA's in-situ timestep alignment not only accelerates offline trace analytics but enables skipping persistence entirely—only the relevant data is materialized for a given view, and queries translate into execution strategies that approach the minimum necessary work. Data movement is minimized via pushdowns that leverage workload resources, and RDMA minimizes the residual data movement overhead. Prior [195–197] work has explored query-driven tracing for asynchronous microservices. ORCA extends this to BSP workloads, adding a control plane for steerability.

Steerability Limits. Feedback-driven mechanisms have an inherent latency floor: collection, reduction, and transport take physical time. ORCA eliminates the artificial costs that dominate today—ETL and unnecessary data movement—but cannot circumvent this floor. Some regimes demand action faster than any global view can be assembled. Network congestion control algorithms, for instance, are designed explicitly around partial observability [203]. When conditions are too transient for online intervention, ORCA's in-situ reductions still capture their structure more efficiently than any post-hoc approach, providing the empirical basis for developing predictive strategies that anticipate rather than react [204].

Applicability Beyond BSP. While this thesis focuses on steerability under BSP constraints, the broader lesson is that observability becomes far more powerful when built over an explicit ontology of system state rather than over fixed telemetry categories such as traces, metrics, and logs. BSP makes this unusually tractable because it defines a regular unit of distributed progress with strong consistency guarantees that applies regardless of scale. For observability beyond BSP, ORCA suggests two directions. Conceptually, it argues for richer ontologies, modeling systems as not just serving requests asynchronously, but as databases, filesystems, and inference engines, extending what can become visible and actionable, with instructive parallels in Palantir [205]. Architecturally, it is a template for observability as tier-aware streaming analytics evaluating invariants over distributed application state, and persisting detail only when those invariants are violated—closer to a distributed core dump than to always-on retention. Extending this model beyond BSP, however, requires solving the harder problem that BSP largely sidesteps: defining the ontology, causal boundaries, and staleness bounds in systems that do not provide a global clock for free, and deriving from them meaningful pushed-down units of work.

6.2 Lessons from OLAP for Scientific Data Management

Having reviewed ORCA as an observability and control substrate, we now turn to broader lessons from CARP and ORCA for scientific data management.

Why HPC Ingestion Is Filesystem-Centric. HPC applications write to parallel filesystems [51, 206–210] and object stores [211] designed to absorb synchronized bursts of data from tens of thousands of ranks, instead of databases. Workflows may additionally interpose intermediate staging tiers [212]. While data management techniques such as partitioning are deeply relevant, they must be realized as in-situ streaming pipelines that reorganize data while minimizing ingestion overhead, terminating in files and objects. The finite nature of HPC jobs, along with the heterogenous and application-specific nature of scientific data formats, resists monolithic databases.

Relevance of Open Table Formats. By decoupling query engines from data at rest stored in standard formats, the modern analytics ecosystem has converged on an architecture well-aligned with HPC workflows. Open table formats such as Iceberg [123] and Delta Lake [189] maintain a lightweight catalog over partitioned columnar files or objects. Query engines attach at read time and use this catalog for pruning, pushdown, and lazy evaluation. CARP and ORCA converge on the same pattern. CARP range-partitions particle data into contiguous key and value blocks with metadata as a separate manifest, structurally identical to Parquet and Iceberg. ORCA writes trace data as timestep-partitioned Parquet but does not currently generate a catalog. Open table formats are immediately applicable to other forms of scientific data, insofar as they fit the relational model.

Challenges for Scientific Catalog Formats. Scientific data formats extend beyond the relational model. While particle data and telemetry streams map cleanly to tables, other simulation

paradigms write multidimensional arrays and custom metadata via formats like HDF5 [213], netCDF [214], Zarr [215], and XDMF [216]. Among these, Zarr is closest to a catalog-backed layout, having evolved for GIS data stored in cloud object stores. Its metadata enables index-based pruning, but it lacks value-level statistics for filter predicates. arrow-zarr [201] is an early attempt to bridge Zarr and Arrow [39]. Despite these gaps, simulation structure already aligns with the broader pattern: problem decomposition intrinsically partitions state in many codes, and the main task is to preserve that partitioning through I/O and capture it in a catalog. On the analytics side, such catalogs would enable query engine-based access to scientific data, extending the benefits of pushdown, pruning, and planning beyond relational workloads. DataFusion's [42] extensibility makes this a viable path.

6.3 Immediate Extensions

Multi-Dimensional Parallelism for ML Training. Large-scale ML training implements multiple levels of parallelism — data, tensor, pipeline, and expert — that compose into complex multi-dimensional topologies [3, 5]. While the execution model remains fundamentally BSP at each level, the nesting creates structure that ORCA does not currently exploit. The TBON topology and the query planner would benefit from alignment to these parallelism dimensions, enabling intra-data-parallel reductions to be pushed deeper in the tree than cross-data-parallel aggregations. The ML training ecosystem currently lacks a standard way to construct and query these topologies at runtime, so this would require integration with training frameworks such as Megatron [217] and DeepSpeed [218] to discover the parallelism layout and map it onto ORCA's overlay.

Shuffling-Based Streaming Analytics. Shuffling-based in-situ indexing systems like DeltaFS [24] and CARP [25] show that hash and range partitioning can reorganize scientific data effectively at scale. ORCA can treat these shuffles as intra-tier operators within its topology, so they compose cleanly with inter-tier reductions in a single plan. Because the MPI tier has substantial egress bandwidth, the planner can be phase-aware, scheduling sinks at the MPI tier in checkpoint phases while avoiding that interference during compute phases.

Instrumentation Sources. Tools like Kineto [29] aggregate telemetry from multiple sources into a unified trace. ORCA supports the same but as real-time aggregations over any interface that emits structured events—CUPTI [219] GPU hooks, MPI runtime probes, hardware counters. eBPF [135] and DynInst [184] are especially appealing as they enable dynamic instrumentation without recompilation, extending visibility into arbitrary functions at runtime.

Approximate Summaries. Mergeable sketches can summarize streams in-flight at far lower cost and fit naturally into ORCA's reduction topology. For example, KLL [106] can estimate latency percentiles, and Count-Min [220] can identify heavy hitters such as straggler ranks or hottest callsites, all with bounded error and without retaining raw events.

Control Interface Regularization. The set of controllable actions is currently defined ad hoc per integration. Regularizing this into typed providers that advertise capabilities, accept structured commands, and return typed acknowledgments makes control capabilities discoverable and composable, analogous to a southbound API in SDN [221].

6.4 Future Directions

ORCA's current query model is relational: SQL over event streams, planned and placed across the overlay. The power of an algebraic approach is that complex pipelines reduce to compositions of well-defined operators, each independently placeable and optimizable. The directions below extend this principle to richer data types, deeper execution tiers, and adaptive placement.

Beyond Relational Algebra. Scientific computing and ML operate primarily on arrays, meshes, and tensors using domain-specific analytics that go beyond the relational model. N-D arrays are strided access patterns over contiguous buffers, requiring only dimensional metadata on top of the columnar representation. More broadly, memory is linear regardless of how data is logically shaped, and columnar formats capture that linearity directly. Everything above the buffer is metadata. Prior work on SciQL [222] and MeshSQL [223] has explored declarative interfaces for these types, and practical convergence is underway through bridges like arrow-zarr [201] and Arrow↔DLPack [202]. DataFusion's extensibility makes it possible to host array and mesh operators alongside relational ones in the same query plan. This also opens a path to expressing HPC visualization workflows [62, 65] as dataflows rather than separate pipelines. A related direction is observability over GPU-resident state such as model weights, where device-side reductions would emit compact summaries without transferring full tensors to the host.

Multi-Tier Compilation. ORCA's overlay currently executes all operators on host-side aggregators. Additional execution tiers are becoming viable: SmartNICs can run Arrow-native reductions on the fabric itself [199, 200], and eBPF can push filter and sketch logic into kernel-side probes. Each tier admits a restricted operator subset under different resource and state constraints. More broadly, unified intermediate representations and layered compilation are emerging as the path toward composing data processing across heterogeneous execution targets [224]. The compilation problem for ORCA — decomposing a high-level query across host, NIC, and kernel tiers — is an instance of this.

Cost-Aware Reactive Placement. ORCA's placement today follows the tree topology: per-partition work is pushed down, while cross-partition work remains at higher tiers. While this works, it is not cost-aware because it does not react to observed placement conditions such as backlogs and kernel costs. A natural extension is to feed these cost signals back into the planner and let it move operators and sinks across tiers online. This effectively makes ORCA self-tuning using the same observability machinery it provides to applications.

Bibliography

[1] Sambit Das et al. “Large-Scale Materials Modeling at Quantum Accuracy: Ab Initio Simulations of Quasicrystals and Interacting Extended Defects in Metallic Alloys”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2023, pp. 1–12. DOI: 10.1145/3581784.3627037. URL: https://dl.acm.org/doi/10.1145/3581784.3627037.

[2] Ryan Stocks et al. “Breaking the Million-Electron and 1 EFLOP/s Barriers: Biomolecular-Scale Ab Initio Molecular Dynamics Using MP2 Potentials”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC ’24. 2024, pp. 1–12. DOI: 10.1109/SC41406.2024.00015. URL: https://dl.acm.org/doi/10.1109/SC41406.2024.00015.

[3] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. DOI: 10.48550/arXiv.2407.21783. arXiv: 2407.21783 [cs]. URL: http://arxiv.org/abs/2407.21783. Pre-published.

[4] *Meta Trains Llama 4 on a 100,000+ H100 GPU Supercluster - NADDOD Blog*. URL: https://www.naddod.com/blog/meta-trains-llama-4-on-a-100-000-h100-gpu-supercluster.

[5] Robi Rahman. *AI Training Cluster Sizes Increased by More than 20x since 2016*. Epoch AI. 2024. URL: https://epoch.ai/data-insights/training-cluster-size.

[6] William M. Jones, John T. Daly, and Nathan DeBardeleben. “Application Monitoring and Checkpointing in HPC: Looking towards Exascale Systems”. In: *Proceedings of the 50th Annual Southeast Regional Conference*. 2012, pp. 262–267. DOI: 10.1145/2184512.2184574. URL: https://dl.acm.org/doi/10.1145/2184512.2184574.

[7] Assaf Eisenman et al. “{Check-N-Run}: A Checkpointing System for Training Deep Learning Recommendation Models”. In: *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 2022, pp. 929–943. URL: https://www.usenix.org/conference/nsdi22/presentation/eisenman.

[8] Xinyu Lian et al. *Universal Checkpointing: A Flexible and Efficient Distributed Checkpointing System for Large-Scale DNN Training with Reconfigurable Parallelis*. Version 3. 2025. DOI: 10.48550/arXiv.2406.18820. arXiv: 2406.18820 [cs]. URL: http://arxiv.org/abs/2406.18820. Pre-published.

[9] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. “Characterizing the Influence of System Noise on Large-Scale Applications by Simulation”. In: *SC ’10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. 2010, pp. 1–11. DOI: 10.1109/SC.2010.12. URL: https://ieeexplore.ieee.org/document/5645455.

[10] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q”. In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. 2003, p. 55. DOI: 10.1145/1048935.1050204. URL: https://dl.acm.org/doi/10.1145/1048935.1050204.

[11] Bilge Acun, Phil Miller, and Laxmikant V. Kale. “Variation Among Processors Under Turbo Boost in HPC Systems”. In: *Proceedings of the 2016 International Conference on Supercomputing*. 2016, pp. 1–12. DOI: 10.1145/2925426.2926289. URL: https://dl.acm.org/doi/10.1145/2925426.2926289.

[12] Haryadi S. Gunawi et al. “Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems”. In: *ACM Transactions on Storage* 14.3 (2018), pp. 1–26. DOI: 10.1145/3242086. URL: https://dl.acm.org/doi/10.1145/3242086.

[13] Abhinav Bhatele et al. “The Case of Performance Variability on Dragonfly-based Systems”. In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020, pp. 896–905. DOI: 10.1109/IPDPS47924.2020.00096. URL: https://ieeexplore.ieee.org/document/9139880.

[14] Zhiyi Yao et al. “Holmes: Localizing Irregularities in LLM Training with Mega-scale GPU Clusters”. In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 2025, pp. 523–540. URL: https://www.usenix.org/conference/nsdi25/presentation/yao.

[15] Tianyuan Wu et al. “GREYHOUND: Hunting Fail-Slows in Hybrid-Parallel Training at Scale”. In: *2025 USENIX Annual Technical Conference (USENIX ATC 25)*. 2025, pp. 731–747. URL: https://www.usenix.org/conference/atc25/presentation/wu-tianyuan.

[16] Weihao Cui et al. “Flare: Anomaly Diagnostics for Divergent LLM Training in GPU Clusters of Thousand-Plus Scale”. In: *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*. 2026. URL: https://www.usenix.org/conference/nsdi26/presentation/cui.

[17] Yu Guan et al. *PerfTracker: Online Performance Troubleshooting for Large-scale Model Training in Production*. Version 2. 2025. DOI: 10.48550/arXiv.2506.08528. arXiv: 2506.08528 [cs]. URL: http://arxiv.org/abs/2506.08528. Pre-published.

[18] Yangtao Deng et al. “Mycroft: Tracing Dependencies in Collective Communication Towards Reliable LLM Training”. In: *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*. SOSP ’25. 2025, pp. 254–269. DOI: 10.1145/3731569.3764848. URL: https://doi.org/10.1145/3731569.3764848.

[19] Ziji Chen et al. *Detecting Anomalies in Machine Learning Infrastructure via Hardware Telemetry*. 2025. DOI: 10.48550/arXiv.2510.26008. arXiv: 2510.26008 [cs]. URL: http://arxiv.org/abs/2510.26008. Pre-published.

[20] Yangtao Deng et al. “Minder: Faulty Machine Detection for Large-scale Distributed Model Training”. In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 2025, pp. 505–521. URL: https://www.usenix.org/conference/nsdi25/presentation/deng.

[21] Jianbo Dong et al. “Evolution of Aegis: Fault Diagnosis for AI Model Training Service in Production”. In: *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*. 2025, pp. 865–881. URL: https://www.usenix.org/conference/nsdi25/presentation/dong.

[22] Yuxuan Jiang et al. “Training with Confidence: Catching Silent Errors in Deep Learning Training with Automated Proactive Checks”. In: *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 2025, pp. 313–329. URL: https://www.usenix.org/conference/osdi25/presentation/jiang.

[23] Alexander Rasmussen et al. “TritonSort: A Balanced Large-Scale Sorting System”. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI’11. 2011, pp. 29–42.

[24] Qing Zheng et al. “Scaling Embedded In-Situ Indexing with DeltaFS”. In: *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2018, pp. 30–44. DOI: 10.1109/SC.2018.00006. URL: https://ieeexplore.ieee.org/document/8665820.

[25] Ankush Jain et al. “CARP: Range Query-Optimized Indexing for Streaming Data”. In: *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. 2024, pp. 1–19. DOI: 10.1109/SC41406.2024.00093. URL: https://ieeexplore.ieee.org/document/10793150.

[26] Anshu Dubey et al. “A Survey of High Level Frameworks in Block-Structured Adaptive Mesh Refinement Packages”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014), pp. 3217–3227. DOI: 10.1016/j.jpdc.2014.07.001. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731514001178.

[27] Carsten Burstedde et al. “Scalable Adaptive Mantle Convection Simulation on Petascale Supercomputers”. In: *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*. 2008, pp. 1–15. DOI: 10.1109/SC.2008.5214248. URL: http://ieeexplore.ieee.org/document/5214248/.

[28] Sameer S. Shende and Allen D. Malony. “The Tau Parallel Performance System”. In: *Int. J. High Perform. Comput. Appl.* 20.2 (2006), pp. 287–311. DOI: 10.1177/1094342006064482. URL: https://doi.org/10.1177/1094342006064482.

[29] *Pytorch Kineto*. pytorch, 2025. URL: https://github.com/pytorch/kineto.

[30] Andreas Knüpfer et al. "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir". In: *Tools for High Performance Computing 2011*. Ed. by Holger Brunst et al. 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7. URL: http://link.springer.com/10.1007/978-3-642-31476-6_7.

[31] *Catapult - The Trace-Event File Format*. URL: https://chromium.googlesource.com/catapult/+/HEAD/docs/trace-event-format.md.

[32] Eschweiler Dominic et al. "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries". In: *Advances in Parallel Computing*. 2012. DOI: 10.3233/978-1-61499-041-3-481. URL: https://www.medra.org/servlet/aliasResolver?alias=iospressISSNISBN&issn=0927-5452&volume=22&spage=481.

[33] Ankush Jain et al. "Lessons from Profiling and Optimizing Placement in AMR Codes". In: *2025 IEEE International Conference on Cluster Computing (CLUSTER)*. –Sept. 30, 2025. DOI: 10.1109/cluster59342.2025.11186462.

[34] Hariharan Devarajan et al. "DFTracer: An Analysis-Friendly Data Flow Tracer for AI-Driven Workflows". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. SC '24. 2024, pp. 1–24. DOI: 10.1109/SC41406.2024.00023. URL: https://dl.acm.org/doi/10.1109/SC41406.2024.00023.

[35] David Boehme et al. "Caliper: Performance Introspection for HPC Software Stacks". In: *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2016, pp. 550–560. DOI: 10.1109/SC.2016.46. URL: https://ieeexplore.ieee.org/abstract/document/7877125.

[36] Izzet Yildirim et al. "IOMax: Maximizing Out-of-Core I/O Analysis Performance on HPC Systems". In: *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*. SC-W '23. 2023, pp. 1209–1215. DOI: 10.1145/3624062.3624191. URL: https://dl.acm.org/doi/10.1145/3624062.3624191.

[37] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A Distributed Messaging System for Log Processing". In: *Proceedings of the 6th International Workshop on Networking Meets Databases*. NetDB '11. 2011, pp. 1–7. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[38] Paris Carbone et al. "Apache Flink™: Stream and Batch Processing in a Single Engine". In: *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* 38.4 (2015), pp. 28–38. URL: https://asterios.katsifodimos.com/assets/publications/flink-deb.pdf.

[39] *Apache Arrow*. Apache Arrow. URL: https://arrow.apache.org/.

[40] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. "Hatchet: Pruning the Overgrowth in Parallel Profiles". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2019, pp. 1–21. DOI: 10.1145/3295500.3356219. URL: https://dl.acm.org/doi/10.1145/3295500.3356219.

[41] *Apache Parquet*. Apache Parquet. URL: https://parquet.apache.org/docs/overview/.

[42] Andrew Lamb et al. "Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine". In: *Companion of the 2024 International Conference on Management of Data*. 2024, pp. 5–17. DOI: 10.1145/3626246.3653368. URL: https://dl.acm.org/doi/10.1145/3626246.3653368.

[43] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.1*. Message Passing Interface Forum, 2023. URL: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[44] *NVIDIA Collective Communications Library (NCCL)*. NVIDIA Developer. URL: https://developer.nvidia.com/nccl.

[45] *TOP500 List - June 2025 | TOP500*. URL: https://top500.org/lists/top500/list/2025/06/.

[46] Gregory F Pfister. "An Introduction to the Infiniband Architecture". In: *High performance mass storage and parallel I/O* 42.617–632 (2001), p. 102.

[47] Duncan Roweth et al. "HPE Slingshot Launched into Network Space". In: *Cray User Group* (2022).

[48] J Metz. "Empowering AI Workloads in Ultra Ethernet Consortium". In: *2024 IEEE Photonics Society Summer Topicals Meeting Series (SUM)*. 2024, pp. 1–2. DOI: 10.1109/SUM60964.2024.10614558. URL: https://ieeexplore.ieee.org/document/10614558/.

[49] Nathan DeBardeleben et al. "GPU Behavior on a Large HPC Cluster". In: *Euro-Par 2013: Parallel Processing Workshops*. Ed. by Dieter an Mey et al. 2014, pp. 680–689. DOI: 10.1007/978-3-642-54420-0_66.

[50] Addison Snell and Laura Segervall. *HPC Application Support for GPU Computing*. Intersect360 Research, 2017. URL: https://www.nvidia.com/content/intersect-360-HPC-application-support.pdf.

[51] Peter J Braam and Philip Schwan. "Lustre: The Intergalactic File System". In: *Ottawa Linux Symposium*. Vol. 8. 11. 2002, pp. 3429–3441.

[52] Michael Hennecke. "Daos: A Scale-out High Performance Storage Stack for Storage Class Memory". In: *Supercomputing frontiers* 40 (2020).

[53] *Hammerspace Data Platform*. Hammerspace. 2024. URL: https://hammerspace.com/hammerspace-global-data-platform/.

[54] *The VAST Platform White Paper*. URL: https://www.vastdata.com.

[55] Glenn Lockwood. *LLM Training without a Parallel File System*. 2025. URL: https://blog.glennklockwood.com/2025/02/llm-training-without-parallel-file.html.

[56] Gary Grider. "LANL Platform Planning and Update" (HPC User Forum 2023). 2023. URL: https://hpcuserforum.com/wp-content/uploads/2023/09/Gary-Grider-LANL-Platform-Planning-and-Update.pdf.

[57] Robert Bird et al. "VPIC 2.0: Next Generation Particle-in-Cell Simulations". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 952–963. DOI: 10.1109/TPDS.2021.3084795. URL: https://ieeexplore.ieee.org/document/9444146/.

[58] Adithya Gangidi et al. "RDMA over Ethernet for Distributed Training at Meta Scale". In: *Proceedings of the ACM SIGCOMM 2024 Conference*. 2024, pp. 57–70. DOI: 10.1145/3651890.3672233. URL: https://dl.acm.org/doi/10.1145/3651890.3672233.

[59] Bing Xie et al. "Characterizing Output Bottlenecks in a Supercomputer". In: *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.28. URL: https://ieeexplore.ieee.org/document/6468446/.

[60] Pandian Raju et al. "PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees". In: *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 2017. DOI: 10.1145/3132747.3132765.

[61] Hank Childs et al. "A Terminology for in Situ Visualization and Analysis Systems". In: *The International Journal of High Performance Computing Applications* 34.6 (2020), pp. 676–691. DOI: 10.1177/1094342020935991. URL: https://doi.org/10.1177/1094342020935991.

[62] Utkarsh Ayachit et al. "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. DOI: 10.1145/2828612.2828624. URL: https://dl.acm.org/doi/10.1145/2828612.2828624.

[63] Utkarsh Ayachit et al. "The SENSEI Generic In Situ Interface". In: *2016 Second Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV)*. 2016, pp. 40–44. DOI: 10.1109/ISAV.2016.013. URL: http://ieeexplore.ieee.org/document/7836400/.

[64] Matthew Larsen et al. "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*. ISAV'17. 2017, pp. 42–46. DOI: 10.1145/3144769.3144778. URL: https://dl.acm.org/doi/10.1145/3144769.3144778.

[65] Matthew Larsen et al. "Ascent: A Flyweight In Situ Library for Exascale Simulations". In: *In Situ Visualization for Computational Science*. Ed. by Hank Childs, Janine C. Bennett, and Christoph Garth. 2022, pp. 255–279. DOI: 10.1007/978-3-030-81627-8_12. URL: https://doi.org/10.1007/978-3-030-81627-8_12.

[66] Greg Eisenhauer et al. “Streaming Data in HPC Workflows Using ADIOS”. In: *Proceedings of the Cray User Group*. CUG ’24. 2025, pp. 31–43. DOI: 10.1145/3725789.3725793. URL: https://dl.acm.org/doi/10.1145/3725789.3725793.

[67] Ciprian Docan, Manish Parashar, and Scott Klasky. “DataSpaces: An Interaction and Coordination Framework for Coupled Simulation Workflows”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. HPDC ’10. 2010, pp. 25–36. DOI: 10.1145/1851476.1851481. URL: https://dl.acm.org/doi/10.1145/1851476.1851481.

[68] Benjamin Schwaller. *Integrated System and Application Continuous Performance Monitoring and Analysis Capability (Final)*. SAND2021-11954, 1822583, 700207. 2021, SAND2021–11954, 1822583, 700207. URL: https://www.osti.gov/servlets/purl/1822583/.

[69] Brian J. N. Wylie et al. “15+ Years of Joint Parallel Application Performance Analysis/Tools Training with *Scalasca/Score-P* and *Paraver/Extrae* Toolsets”. In: *Future Generation Computer Systems* 162 (2025), p. 107472. DOI: 10.1016/j.future.2024.07.050. URL: https://www.sciencedirect.com/science/article/pii/S0167739X24004187.

[70] *Pytorch/Kineto*. pytorch, 2025. URL: https://github.com/pytorch/kineto.

[71] *Holistic Trace Analysis — Holistic Trace Analysis 0.5.0 Documentation*. URL: https://hta.readthedocs.io/en/latest/index.html.

[72] Benjamin Klenk and Holger Fröning. “An Overview of MPI Characteristics of Exascale Proxy Applications”. In: *High Performance Computing: 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18–22, 2017, Proceedings*. 2017, pp. 217–236. DOI: 10.1007/978-3-319-58667-0_12. URL: https://doi.org/10.1007/978-3-319-58667-0_12.

[73] *Characterization of the DOE Mini-apps*. URL: https://portal.nersc.gov/project/CAL/doe-miniapps.htm.

[74] G.A. Geist, James Arthur Kohl, and Philip M. Papadopoulos. “CUMULVS: Providing Fault Tolerance, Visualization, and Steering of Parallel Applications”. In: *The International Journal of Supercomputer Applications and High Performance Computing* 11.3 (1997), pp. 224–235. DOI: 10.1177/109434209701100305. URL: https://journals.sagepub.com/doi/10.1177/109434209701100305.

[75] M. Miller et al. “Simulation Steering with SCIRun in a Distributed Environment”. In: *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*. 1998, pp. 364–365. DOI: 10.1109/HPDC.1998.710032. URL: https://ieeexplore.ieee.org/document/710032.

[76] Daniel Lorenz et al. “RMOST: A Shared Memory Model for Online Steering”. In: *Computational Science – ICCS 2008*. Ed. by Marian Bubak et al. Red. by Hutchison et al. Vol. 5103. 2008, pp. 223–232. DOI: 10.1007/978-3-540-69389-5_26. URL: https://link.springer.com/10.1007/978-3-540-69389-5_26.

[77] Delbert Hart, Eileen Kraemer, and Gruia-Catalin Roman. “Consistency Considerations in the Interactive Steering of Computations”. In: *International Journal of Parallel and Distributed Systems and Networks* 2 (1999), pp. 171–179. URL: https://scholar.google.com/scholar?cluster=12278076233441998448&hl=en&oi=scholarr.

[78] P.C. Roth, D.C. Arnold, and B.P. 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. DOI: 10.1145/1048935.1050172. URL: https://ieeexplore.ieee.org/abstract/document/1592924.

[79] Dorian C. Arnold, Gary D. Pack, and Barton P. Miller. “Tree-Based Overlay Networks for Scalable Applications”. In: *Proceedings of the 20th International Conference on Parallel and Distributed Processing*. IPDPS’06. 2006, p. 223. DOI: 10.1109/IPDPS.2006.1639493.

[80] *TotalView: Advanced Debugging for HPC | Perforce Software*. URL: https://www.perforce.com/products/totalview.

[81] Christopher Kelly et al. “Chimbuko: A Workflow-Level Scalable Performance Trace Analysis Tool”. In: *ISAV’20 In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. ISAV’20. 2020, pp. 15–19. DOI: 10.1145/3426462.3426465. URL: https://doi.org/10.1145/3426462.3426465.

[82] William F. Godoy et al. “ADIOS 2: The Adaptable Input Output System. A Framework for High-Performance Data Management”. In: *SoftwareX* 12 (2020). DOI: 10.1016/j.softx.2020.100561. URL: https://www.softxjournal.com/article/S2352-7110(19)30256-0/fulltext.

[83] Abhinav Bhatele et al. *Pipit: Scripting the Analysis of Parallel Execution Traces*. Version 2. 2024. DOI: 10.48550/arXiv.2306.11177. arXiv: 2306.11177 [cs]. URL: http://arxiv.org/abs/2306.11177. Pre-published.

[84] Surendra Byna et al. “Parallel I/O, Analysis, and Visualization of a Trillion Particle Simulation”. In: *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*. 2012. DOI: 10.1109/SC.2012.92.

[85] Philipp Grete et al. “Parthenon—a Performance Portable Block-Structured Adaptive Mesh Refinement Framework”. In: *The International Journal of High Performance Computing Applications* 37.5 (2023), pp. 465–486. DOI: 10.1177/10943420221143775. URL: https://doi.org/10.1177/10943420221143775.

[86] Weiqun Zhang et al. “AMReX: Block-structured Adaptive Mesh Refinement for Multiphysics Applications”. In: *The International Journal of High Performance Computing Applications* 35.6 (2021), pp. 508–526. DOI: 10.1177/10943420211022811. URL: https://doi.org/10.1177/10943420211022811.

[87] Brandon Barker et al. *Phoebus: Performance Portable GRRMHD for Relativistic Astrophysics*. Version 2. 2024. DOI: 10.48550/arXiv.2410.09146. arXiv: 2410.09146 [astro-ph]. URL: http://arxiv.org/abs/2410.09146. Pre-published.

[88] A. C. Bauer et al. “*In Situ* Methods, Infrastructures, and Applications on High Performance Computing Platforms”. In: *Computer Graphics Forum* 35.3 (2016), pp. 577–597. DOI: 10.1111/cgf.12930. URL: https://onlinelibrary.wiley.com/doi/10.1111/cgf.12930.

[89] Jerry Chou, Kesheng Wu, and Prabhat. “FastQuery: A Parallel Indexing System for Scientific Data”. In: *2011 IEEE International Conference on Cluster Computing*. 2011, pp. 455–464. DOI: 10.1109/CLUSTER.2011.86. URL: https://ieeexplore.ieee.org/document/6061134.

[90] Fay Chang et al. “Bigtable: A Distributed Storage System for Structured Data”. In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*. 2006. URL: https://www.usenix.org/conference/osdi-06/bigtable-distributed-storage-system-structured-data.

[91] MikeRayMSFT. *Clustered and Nonclustered Indexes - SQL Server*. 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16.

[92] Mike Ray. *Clustered and Nonclustered Indexes - SQL Server*. 2023. URL: https://learn.microsoft.com/en-us/sql/relational-databases/indexes/clustered-and-nonclustered-indexes-described?view=sql-server-ver16.

[93] Hari Sundar, Dhairya Malhotra, and George Biros. “HykSort: A New Variant of Hypercube Quicksort on Distributed Memory Architectures”. In: *Proceedings of the International Conference on Supercomputing*. 2013. DOI: 10.1145/2464996.2465442.

[94] Bin Dong, Surendra Byna, and Kesheng Wu. “SDS-sort: Scalable Dynamic Skew-Aware Parallel Sorting”. In: *HPDC 2016 - Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. 2016. DOI: 10.1145/2907294.2907300.

[95] Prasanna Ganesan, Mayank Bawa, and Hector Garcia-Molina. “Online Balancing of Range-Partitioned Data with Applications to Peer-to-Peer Systems”. In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB ’04. 2004, pp. 444–455.

[96] Ioannis Konstantinou, Dimitrios Tsoumakos, and Nectarios Koziris. “Fast and Cost-Effective Online Load-Balancing in Distributed Range-Queriable Systems”. In: *IEEE Transactions on Parallel and Distributed Systems* 22.8 (2011), pp. 1350–1364. DOI: 10.1109/TPDS.2010.200.

[97] Yifan Qiao et al. “Closing the B+-Tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression”. In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. 2022, pp. 69–82. URL: https://www.usenix.org/conference/fast22/presentation/qiao.

[98] K. Mani Chandy and Leslie Lamport. "Distributed Snapshots: Determining Global States of Distributed Systems". In: *ACM Transactions on Computer Systems (TOCS)* (1985). DOI: 10.1145/214451.214456.

[99] Qing Zheng et al. "Software-Defined Storage for Fast Trajectory Queries Using a deltaFS Indexed Massive Directory". In: *Proceedings of PDSW-DISCS 2017 - 2nd Joint International Workshop on Parallel Data Storage and Data Intensive Scalable Computing Systems - Held in Conjunction with SC 2017: The International Conference for High Performance Computing, Networking, Storage a*. 2017. DOI: 10.1145/3149393.3149398.

[100] Robert B. Ross et al. "Mochi: Composing Data Services for High-Performance Computing Environments". In: *Journal of Computer Science and Technology* 35.1 (2020), pp. 121–144. DOI: 10.1007/s11390-020-9802-0. URL: https://doi.org/10.1007/s11390-020-9802-0.

[101] Jerome Soumagne et al. "Mercury: Enabling Remote Procedure Call for High-Performance Computing". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702617. URL: http://ieeexplore.ieee.org/document/6702617/.

[102] Rolf Rabenseifner. "Optimization of Collective Reduction Operations". In: *Computational Science - ICCS 2004*. Ed. by Marian Bubak et al. 2004, pp. 1–9. DOI: 10.1007/978-3-540-24685-5_1.

[103] Rolf Rabenseifner and Jesper Larsson Träff. "More Efficient Reduction Algorithms for Non-Power-of-Two Number of Processors in Message-Passing Parallel Systems". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. 2004, pp. 36–46. DOI: 10.1007/978-3-540-30218-6_13.

[104] Benny Vigil. *Trinity Advanced Technology System Overview*. LA-UR-14-28143, 1160100. 2014, LA-UR-14–28143, 1160100. DOI: 10.2172/1160100. URL: https://www.osti.gov/servlets/purl/1160100/.

[105] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. 2010, pp. 143–154. DOI: 10.1145/1807128.1807152. URL: https://dl.acm.org/doi/10.1145/1807128.1807152.

[106] Zohar Karnin, Kevin Lang, and Edo Liberty. *Optimal Quantile Approximation in Streams*. 2016. DOI: 10.48550/arXiv.1603.05346. arXiv: 1603.05346 [cs]. URL: http://arxiv.org/abs/1603.05346. Pre-published.

[107] Timothy Prickett Morgan. *Aurora Rising: A Massive Machine For HPC And AI*. The Next Platform. 2023. URL: https://www.nextplatform.com/2023/05/23/aurora-rising-a-massive-machine-for-hpc-and-ai/.

[108] Patrick O'Neil et al. "The Log-Structured Merge-Tree (LSM-tree)". In: *Acta Informatica* (1996). DOI: 10.1007/s002360050048.

[109] Douglas Comer. "Ubiquitous B-Tree". In: *ACM Comput. Surv.* 11.2 (1979), pp. 121–137. DOI: 10.1145/356770.356776. URL: https://dl.acm.org/doi/10.1145/356770.356776.

[110] Pandian Raju et al. "PebblesDB: Building Key-Value Stores Using Fragmented Log-Structured Merge Trees". In: *SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles*. 2017. DOI: 10.1145/3132747.3132765.

[111] Fang Zheng et al. "PreDatA – Preparatory Data Analytics on Peta-Scale Machines". In: *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470454. URL: https://ieeexplore.ieee.org/document/5470454.

[112] Venkatram Vishwanath, Mark Hereld, and Michael 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*. 2011, pp. 9–14. DOI: 10.1109/LDAV.2011.6092178. URL: https://ieeexplore.ieee.org/document/6092178.

[113] Venkatram Vishwanath et al. "Topology-Aware Data Movement and Staging for I/O Acceleration on Blue Gene/P Supercomputing Systems". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2011, pp. 1–11. DOI: 10.1145/2063384.2063409. URL: https://dl.acm.org/doi/10.1145/2063384.2063409.

[114] Ron A. Oldfield et al. "Trilinos I/O Support Trios". In: *Scientific Programming* 20.2 (2012), pp. 181–196. DOI: 10.1155/2012/842791. URL: https://dl.acm.org/doi/abs/10.1155/2012/842791.

[115] Junmin Gu et al. “Querying Large Scientific Data Sets with Adaptable IO System ADIOS”. In: *Supercomputing Frontiers*. Ed. by Rio Yokota and Weigang Wu. 2018, pp. 51–69. DOI: 10.1007/978-3-319-69953-0_4.

[116] Kesheng Wu. “FastBit: An Efficient Indexing Technology for Accelerating Data-Intensive Science”. In: *Journal of Physics: Conference Series* 16.1 (2005), p. 556. DOI: 10.1088/1742-6596/16/1/077. URL: https://dx.doi.org/10.1088/1742-6596/16/1/077.

[117] Will Usher et al. “Adaptive Spatially Aware I/O for Multiresolution Particle Data Layouts”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2021), pp. 547–556. DOI: 10.1109/IPDPS49936.2021.00063. URL: https://ieeexplore.ieee.org/document/9460496/.

[118] Matthaios Olma et al. “Slalom: Coasting through Raw Data via Adaptive Partitioning and Indexing”. In: *Proceedings of the VLDB Endowment* 10.10 (2017), pp. 1106–1117. DOI: 10.14778/3115404.3115415. URL: https://dl.acm.org/doi/10.14778/3115404.3115415.

[119] Stavros Maroulis et al. “Resource-Aware Adaptive Indexing for in Situ Visual Exploration and Analytics”. In: *The VLDB Journal* 32.1 (2023), pp. 199–227. DOI: 10.1007/s00778-022-00739-z. URL: https://doi.org/10.1007/s00778-022-00739-z.

[120] Lanyue Lu et al. “WiscKey: Separating Keys from Values in SSD-Conscious Storage”. In: *ACM Trans. Storage* 13.1 (2017), 5:1–5:28. DOI: 10.1145/3033273. URL: https://dl.acm.org/doi/10.1145/3033273.

[121] Huanchen Zhang et al. “SuRF: Practical Range Query Filtering with Fast Succinct Tries”. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD ’18. 2018, pp. 323–336. DOI: 10.1145/3183713.3196931. URL: https://dl.acm.org/doi/10.1145/3183713.3196931.

[122] Sebastian Oeste et al. “An Analysis of the I/O Semantic Gaps of HPC Storage Stacks”. In: *Frontiers in High Performance Computing* 3 (2025). DOI: 10.3389/fhpcp.2025.1393936. URL: https://www.frontiersin.org/journals/high-performance-computing/articles/10.3389/fhpcp.2025.1393936/full.

[123] *Apache Iceberg - Apache Iceberg™*. URL: https://iceberg.apache.org/.

[124] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. “P4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees”. In: *SIAM Journal on Scientific Computing* 33.3 (2011), pp. 1103–1133. DOI: 10.1137/100791634. URL: https://epubs.siam.org/doi/10.1137/100791634.

[125] Brian Van Straalen et al. “Scalability Challenges for Massively Parallel AMR Applications”. In: *2009 IEEE International Symposium on Parallel & Distributed Processing*. 2009, pp. 1–12. DOI: 10.1109/IPDPS.2009.5161014. URL: http://ieeexplore.ieee.org/document/5161014/.

[126] Florian Schornbaum and Ulrich Rüde. “Extreme-Scale Block-Structured Adaptive Mesh Refinement”. In: *SIAM Journal on Scientific Computing* 40.3 (2018), pp. C358–C387. DOI: 10.1137/17M1128411. arXiv: 1704.06829 [cs]. URL: http://arxiv.org/abs/1704.06829.

[127] Bilge Acun et al. “Parallel Programming with Migratable Objects: Charm++ in Practice”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 647–658. DOI: 10.1109/SC.2014.58. URL: https://ieeexplore.ieee.org/abstract/document/7013040.

[128] Qingyu Meng, Justin Luitjens, and Martin Berzins. “Dynamic Task Scheduling for the Uintah Framework”. In: *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*. 2010, pp. 1–10. DOI: 10.1109/MTAGS.2010.5699431. URL: https://ieeexplore.ieee.org/document/5699431/?arnumber=5699431.

[129] Gregor Daiß et al. *Asynchronous-Many-Task Systems: Challenges and Opportunities – Scaling an AMR Astrophysics Code on Exascale Machines Using Kokkos and HPX*. 2024. DOI: 10.48550/arXiv.2412.15518. arXiv: 2412.15518 [cs]. URL: http://arxiv.org/abs/2412.15518. Pre-published.

[130] Michael Bauer et al. “Legion: Expressing Locality and Independence with Logical Regions”. In: *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71. URL: https://ieeexplore.ieee.org/document/6468504.

[131] Andrew Emerick. "Enzo-E/Cello Project: Enabling Exa-Scale Astrophysics". 2019. URL: https://bluewaters.ncsa.illinois.edu/liferay-content/document-library/19symposium-slides/emerick_enzo.pdf.

[132] Brian T.N. Gunney and Robert W. Anderson. "Advances in Patch-Based Adaptive Mesh Refinement Scalability". In: *Journal of Parallel and Distributed Computing* 89 (2016), pp. 65–84. DOI: 10.1016/j.jpdc.2015.11.005. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731515002129.

[133] Torsten Hoefler et al. "A Case for Standard Non-blocking Collective Operations". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Franck Cappello, Thomas Herault, and Jack Dongarra. Vol. 4757. 2007, pp. 125–134. DOI: 10.1007/978-3-540-75416-9_22. URL: http://link.springer.com/10.1007/978-3-540-75416-9_22.

[134] *Perf: Linux Profiling with Performance Counters*. URL: https://perfwiki.github.io/main/.

[135] *What Is eBPF? An Introduction and Deep Dive into the eBPF Technology*. URL: https://ebpf.io/what-is-ebpf/.

[136] Dieter an Mey et al. "Score-P: A Unified Performance Measurement System for Petascale Applications". In: *Competence in High Performance Computing 2010*. Ed. by Christian Bischof et al. 2012, pp. 85–97. DOI: 10.1007/978-3-642-24025-6_8.

[137] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. 1979.

[138] Dhabaleswar Kumar Panda et al. "The MVAPICH Project: Transforming Research into High-Performance MPI Library for HPC Community". In: *Journal of Computational Science*. Case Studies in Translational Computer Science 52 (2021), p. 101208. DOI: 10.1016/j.jocs.2020.101208. URL: https://www.sciencedirect.com/science/article/pii/S1877750320305093.

[139] *Intel Performance Scaled Messaging 2 (PSM2) Programmer's Guide*. H76473-1.0. Intel Corporation, 2015. URL: https://www.intel.com/content/dam/support/us/en/documents/network/omni-adptr/sb/Intel_PSM2_PG_H76473_v1_0.pdf.

[140] Ruiming Lu et al. "PERSEUS: A Fail-Slow Detection Framework for Cloud Storage Systems". In: *Proceedings of the 21st USENIX Conference on File and Storage Technologies*. FAST'23. 2023, pp. 49–63.

[141] Tobias Hilbrich et al. "MUST: A Scalable Approach to Runtime Error Detection in MPI Programs". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. 2010, pp. 53–66. DOI: 10.1007/978-3-642-11261-4_5. URL: https://link.springer.com/10.1007/978-3-642-11261-4_5.

[142] Simon D. Hammond et al. "Profiling and Debugging Support for the Kokkos Programming Model". In: *High Performance Computing*. Ed. by Rio Yokota et al. Vol. 11203. 2018, pp. 743–754. DOI: 10.1007/978-3-030-02465-9_53. URL: http://link.springer.com/10.1007/978-3-030-02465-9_53.

[143] Robert Schulze et al. "ClickHouse - Lightning Fast Analytics for Everyone". In: *Proceedings of the VLDB Endowment* 17.12 (2024), pp. 3731–3744. DOI: 10.14778/3685800.3685802. URL: https://dl.acm.org/doi/10.14778/3685800.3685802.

[144] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[145] R. L. Graham. "Bounds on Multiprocessing Timing Anomalies". In: *SIAM Journal on Applied Mathematics* 17.2 (1969), pp. 416–429. DOI: 10.1137/0117039. URL: https://epubs.siam.org/doi/10.1137/0117039.

[146] Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*. 2026. URL: https://www.gurobi.com.

[147] A. F. Rodrigues et al. "The Structural Simulation Toolkit". In: *SIGMETRICS Perform. Eval. Rev.* 38.4 (2011), pp. 37–42. DOI: 10.1145/1964218.1964225. URL: https://dl.acm.org/doi/10.1145/1964218.1964225.

[148] Gengbin Zheng et al. “Periodic Hierarchical Load Balancing for Large Supercomputers”. In: *The International Journal of High Performance Computing Applications* 25.4 (2011), pp. 371–385. DOI: 10.1177/1094342010394383. URL: https://journals.sagepub.com/doi/10.1177/1094342010394383.

[149] Torsten Hoefler and Timo Schneider. “Optimization Principles for Collective Neighborhood Communications”. In: *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*. 2012, pp. 1–10. DOI: 10.1109/SC.2012.86. URL: http://ieeexplore.ieee.org/document/6468467/.

[150] Orcun Yildiz et al. “On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2016, pp. 750–759. DOI: 10.1109/IPDPS.2016.50. URL: https://ieeexplore.ieee.org/document/7516071.

[151] Brian Kocoloski and John Lange. “Varbench: An Experimental Framework to Measure and Characterize Performance Variability”. In: *Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10. DOI: 10.1145/3225058.3225125. URL: https://dl.acm.org/doi/10.1145/3225058.3225125.

[152] C.-Q. Yang and B.P. Miller. “Critical Path Analysis for the Execution of Parallel and Distributed Programs”. In: *[1988] Proceedings. The 8th International Conference on Distributed*. 1988, pp. 366–373. DOI: 10.1109/DCS.1988.12538. URL: http://ieeexplore.ieee.org/document/12538/.

[153] Torsten Hoefler et al. “The Scalable Process Topology Interface of MPI 2.2”. In: *Concurrency and Computation: Practice and Experience* 23.4 (2011), pp. 293–310. DOI: 10.1002/cpe.1643. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1643.

[154] S Mahdieh Ghazimirsaeed. “Efficient Designs for Distributed MPI Neighborhood Collectives”. In: ().

[155] Dominique Lasalle and George Karypis. “Multi-Threaded Graph Partitioning”. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 225–236. DOI: 10.1109/IPDPS.2013.50. URL: https://ieeexplore.ieee.org/document/6569814.

[156] Umit V. Catalyurek et al. “Hypergraph-Based Dynamic Load Balancing for Adaptive Scientific Computations”. In: *2007 IEEE International Parallel and Distributed Processing Symposium*. 2007, pp. 1–11. DOI: 10.1109/IPDPS.2007.370258. URL: http://ieeexplore.ieee.org/document/4227986/.

[157] Aparna Sasidharan, John M. Dennis, and Marc Snir. “A General Space-filling Curve Algorithm for Partitioning 2D Meshes”. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems*. 2015, pp. 875–879. DOI: 10.1109/HPCC-CSS-ICESS.2015.192. URL: https://ieeexplore.ieee.org/document/7336274/.

[158] Harshitha Menon et al. “Automated Load Balancing Invocation Based on Application Characteristics”. In: *2012 IEEE International Conference on Cluster Computing*. 2012, pp. 373–381. DOI: 10.1109/CLUSTER.2012.61. URL: https://ieeexplore.ieee.org/document/6337800/?arnumber=6337800.

[159] Andrew Newell et al. “RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation”. In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. 2021, pp. 505–520. DOI: 10.1145/3477132.3483578. URL: https://dl.acm.org/doi/10.1145/3477132.3483578.

[160] *Caliper: A Performance Analysis Toolbox in a Library — Caliper 2.11.0-Dev Documentation*. URL: https://software.llnl.gov/Caliper/.

[161] *Perfetto - System Profiling, App Tracing and Trace Analysis*. Perfetto. URL: https://perfetto.dev/.

[162] *Parquet*. Apache Parquet. URL: https://parquet.apache.org/.

[163] Rodrigo Fonseca et al. “X-Trace: A Pervasive Network Tracing Framework”. In: *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. 2007. URL: https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework.

[164] Benjamin H. Sigelman et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Google, Inc., 2010. URL: https://research.google.com/archive/papers/dapper-2010-1.pdf.

[165] Ankush Jain, Sheng Jiang, and Chuck Cranor. *ORCA: Observability with Real-Time Control and Aggregation*. 2026. URL: https://github.com/pdlfs/orca-umbrella.

[166] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Vol. 32. 2019. URL: https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[167] *Facebookincubator/Dynolog*. Meta Incubator, 2025. URL: https://github.com/facebookincubator/dynolog.

[168] *Facebookresearch/HolisticTraceAnalysis*. Meta Research, 2025. URL: https://github.com/facebookresearch/HolisticTraceAnalysis.

[169] *Prometheus - Monitoring System & Time Series Database*. URL: https://prometheus.io/.

[170] Christian R. Trott et al. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283. URL: https://ieeexplore.ieee.org/document/9485033.

[171] *Pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/.

[172] Matthew Rocklin. "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling". In: *Python in Science Conference*. 2015, pp. 126–132. DOI: 10.25080/Majora-7b98e3ed-013. URL: https://doi.curvenote.com/10.25080/Majora-7b98e3ed-013.

[173] William R. Williams et al. "Dyninst and MRNet: Foundational Infrastructure for Parallel Tools". In: *Tools for High Performance Computing 2015*. Ed. by Andreas Knüpfer et al. 2016, pp. 1–16. DOI: 10.1007/978-3-319-39589-0_1.

[174] Mark Raasveldt and Hannes Mühleisen. "DuckDB: An Embeddable Analytical Database". In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1981–1984. DOI: 10.1145/3299869.3320212. URL: https://dl.acm.org/doi/10.1145/3299869.3320212.

[175] Paul Barham et al. "Using Magpie for Request Extraction and Workload Modelling". In: *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*. 2004. URL: https://www.usenix.org/conference/osdi-04/using-magpie-request-extraction-and-workload-modelling.

[176] Daniele De Sensi et al. "An In-Depth Analysis of the Slingshot Interconnect". In: *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1 vols. 2020, pp. 1–14. DOI: 10.1109/SC41405.2020.00039. URL: https://doi.org/10.1109/SC41405.2020.00039.

[177] *Grafana | Query, Visualize, Alerting Observability Platform*. Grafana Labs. URL: https://grafana.com/grafana/.

[178] Matei Zaharia et al. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'10. 2010, p. 10. URL: https://www.usenix.org/conference/hotcloud-10/spark-cluster-computing-working-sets.

[179] Shajulin Benedict, Ventsislav Petkov, and Michael Gerndt. "PERISCOPE: An Online-Based Distributed Performance Analysis Tool". In: *Tools for High Performance Computing 2009*. Ed. by Matthias S. Müller et al. 2010, pp. 1–16. DOI: 10.1007/978-3-642-11261-4_1. URL: https://doi.org/10.1007/978-3-642-11261-4_1.

[180] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167. URL: https://dl.acm.org/doi/10.1145/98163.98167.

[181] Jim Gray. "Notes on Data Base Operating Systems". In: *Operating Systems, An Advanced Course*. 1978, pp. 393–481.

[182] Paul Grun et al. "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*. 2015, pp. 34–39. DOI: 10.1109/HOTI.2015.19. URL: https://ieeexplore.ieee.org/document/7312664.

[183] Pavel Shamis et al. “UCX: An Open Source Framework for HPC Network APIs and Beyond”. In: *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 2015, pp. 40–43. DOI: 10.1109/HOTI.2015.13. URL: https://ieeexplore.ieee.org/document/7312665.

[184] Andrew R. Bernat and Barton P. Miller. “Anywhere, Any-Time Binary Instrumentation”. In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE ’11. 2011, pp. 9–16. DOI: 10.1145/2024569.2024572. URL: https://dl.acm.org/doi/10.1145/2024569.2024572.

[185] Leonid Ivanovich Sedov. *Similarity and Dimensional Methods in Mechanics*. 1959. DOI: 10.1201/9780203739730. URL: http://archive.org/details/l-i-sedov.-similarity-and-dimensional-methods-in-mechanics.

[186] *PSM: Performance Scaled Messaging*. Intel® Corporation, 2025. URL: https://github.com/intel/psm.

[187] *Snappy: A Fast Compressor/Decompressor*. Google, 2026. URL: https://github.com/google/snappy.

[188] *Polars*. URL: https://www.pola.rs/.

[189] Michael Armbrust et al. “Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores”. In: *Proceedings of the VLDB Endowment* 13.12 (2020), pp. 3411–3424. DOI: 10.14778/3415478.3415560. URL: https://dl.acm.org/doi/10.14778/3415478.3415560.

[190] *Claude Code Overview*. Claude Code Docs. URL: https://code.claude.com/docs/en/overview.

[191] Anthony Agelastos 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. DOI: 10.1109/SC.2014.18. URL: https://ieeexplore.ieee.org/document/7013000.

[192] M. Abolins et al. “The ATLAS Data Acquisition and High Level Trigger System”. In: *Journal of Instrumentation* 11.6 (2016). DOI: 10.1088/1748-0221/11/06/P06008. URL: https://www.scopus.com/pages/publications/85019832772.

[193] G. Bauer. “Operational Experience with the CMS Data Acquisition System”. In: *Journal of Physics: Conference Series* 396.1 (2012). DOI: 10.1088/1742-6596/396/1/012007. URL: https://www.osti.gov/biblio/1405171.

[194] Yusheng Zheng et al. “Extending Applications Safely and Efficiently”. In: *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*. 2025, pp. 557–574. URL: https://www.usenix.org/conference/osdi25/presentation/zheng-yusheng.

[195] Úlfar Erlingsson et al. “Fay: Extensible Distributed Tracing from Kernels to Clusters”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 311–326. DOI: 10.1145/2043556.2043585. URL: https://dl.acm.org/doi/10.1145/2043556.2043585.

[196] Jessica Berg et al. “Snicket: Query-Driven Distributed Tracing”. In: *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*. 2021, pp. 206–212. DOI: 10.1145/3484266.3487393. URL: https://dl.acm.org/doi/10.1145/3484266.3487393.

[197] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. “Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems”. In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP ’15. 2015, pp. 378–393. DOI: 10.1145/2815400.2815415. URL: https://dl.acm.org/doi/10.1145/2815400.2815415.

[198] Milind Srivastava and Vyas Sekar. “Circa: Re-imagining Network Telemetry from an Approximation-First Perspective”. In: *Proceedings of the ACM SIGCOMM 2024 Conference: Posters and Demos*. ACM SIGCOMM Posters and Demos ’24. 2024, pp. 57–59. DOI: 10.1145/3672202.3673742. URL: https://dl.acm.org/doi/10.1145/3672202.3673742.

[199] Jianshen Liu et al. “Processing Particle Data Flows with SmartNICs”. In: *2022 IEEE High Performance Extreme Computing Conference (HPEC)*. 2022, pp. 1–8. DOI: 10.1109/HPEC55821.2022.9926325. URL: https://ieeexplore.ieee.org/document/9926325.

[200] Craig Ulmer et al. “Extending Composable Data Services into SmartNICs”. In: *2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2023, pp. 953–959. DOI: 10.1109/IPDPSW59300.2023.00158. URL: https://ieeexplore.ieee.org/document/10196516.

[201] *Datafusion-Contrib/Arrow-Zarr*. datafusion-contrib, 2026. URL: https://github.com/datafusion-contrib/arrow-zarr.

[202] *The DLPack Protocol — Apache Arrow V22.0.0*. URL: https://arrow.apache.org/docs/python/dlpack.html.

[203] V. Jacobson. “Congestion Avoidance and Control”. In: *SIGCOMM Comput. Commun. Rev.* 18.4 (1988), pp. 314–329. DOI: 10.1145/52325.52356. URL: https://dl.acm.org/doi/10.1145/52325.52356.

[204] Otto JM Smith. “Closer Control of Loops with Dead Time”. In: *Chemical Engineering Progress* 53 (1957), pp. 217–219. URL: https://cir.nii.ac.jp/crid/1574231875295965056.

[205] Palantir Technologies. *Overview • Ontology*. URL: https://www.palantir.com/docs/foundry/ontology/overview.

[206] Frank B. Schmuck and Roger L. Haskin. “GPFS: A Shared-Disk File System for Large Computing Clusters”. In: *Proceedings of the Conference on File and Storage Technologies*. FAST ’02. 2002, pp. 231–244.

[207] *VDURA: Velocity Meets Durability*. VDURA. URL: https://www.vdura.com/.

[208] VAST Data. *VAST AI Operating System: Powering the Agentic AI Revolution*. VAST Data. 2025. URL: https://www.vastdata.com.

[209] *NeuralMesh: Accelerated Infrastructure Solutions for Agentic AI*. WEKA. URL: https://www.weka.io/.

[210] *Hammerspace - The Data Platform for AI Anywhere*. Hammerspace. URL: https://hammerspace.com/.

[211] Zhen Liang et al. “DAOS: A Scale-Out High Performance Storage Stack for Storage Class Memory”. In: *Supercomputing Frontiers*. Ed. by Dhabaleswar K. Panda. 2020, pp. 40–54. DOI: 10.1007/978-3-030-48842-0_3.

[212] Ning Liu et al. “On the Role of Burst Buffers in Leadership-Class Storage Systems”. In: *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 2012, pp. 1–11. DOI: 10.1109/MSST.2012.6232369. URL: https://ieeexplore.ieee.org/abstract/document/6232369.

[213] Mike Folk et al. “An Overview of the HDF5 Technology Suite and Its Applications”. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. AD ’11. 2011, pp. 36–47. DOI: 10.1145/1966895.1966900. URL: https://dl.acm.org/doi/10.1145/1966895.1966900.

[214] *NetCDF Users Guide: An Introduction to NetCDF*. URL: https://docs.unidata.ucar.edu/nug/current/netcdf_introduction.html.

[215] *Zarr*. Zarr. URL: https://zarr.dev/.

[216] Jerry Clarke and Eric R. Mark. “Enhancements to the eXtensible Data Model and Format (XDMF)”. In: *2007 DoD High Performance Computing Modernization Program Users Group Conference*. 2007, pp. 322–327. DOI: 10.1109/HPCMP-UGC.2007.30. URL: https://ieeexplore.ieee.org/document/4438005.

[217] Mohammad Shoeybi et al. *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism*. 2020. DOI: 10.48550/arXiv.1909.08053. arXiv: 1909.08053 [cs]. URL: http://arxiv.org/abs/1909.08053. Pre-published.

[218] *DeepSpeed: Extreme Speed and Scale for DL Training*. DeepSpeed. URL: https://www.deepspeed.ai/.

[219] *NVIDIA CUDA Profiling Tools Interface (CUPTI) - CUDA Toolkit*. NVIDIA Developer. URL: https://developer.nvidia.com/cupti.

[220] Graham Cormode and S. Muthukrishnan. “An Improved Data Stream Summary: The Count-Min Sketch and Its Applications”. In: *Journal of Algorithms* 55.1 (2005), pp. 58–75. DOI: 10.1016/j.jalgor.2003.12.001. URL: https://www.sciencedirect.com/science/article/pii/S0196677403001913.

[221] Wolfgang Braun and Michael Menth. “Software-Defined Networking Using OpenFlow: Protocols, Applications and Architectural Design Choices”. In: *Future Internet* 6.2 (2014), pp. 302–336. DOI: 10.3390/fi6020302. URL: https://www.mdpi.com/1999-5903/6/2/302.

[222] M. Kersten et al. “SciQL, a Query Language for Science Applications”. In: *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*. AD ’11. 2011, pp. 1–12. DOI: 10.1145/1966895.1966896. URL: https://dl.acm.org/doi/10.1145/1966895.1966896.

[223] Byung S Lee and Ron Musick. “MeshSQL: The Query Language for Simulation Mesh Data”. In: *Information Sciences* 159.3 (2004), pp. 177–202. DOI: 10.1016/j.ins.2003.02.002. URL: https://www.sciencedirect.com/science/article/pii/S0020025503001981.

[224] Michael Jungmair and Jana Giceva. “Towards Designing Future-Proof Data Processing Systems”. In: *Proc. VLDB Endow.* 18.11 (2025), pp. 3988–3995. DOI: 10.14778/3749646.3749669. URL: https://dl.acm.org/doi/10.14778/3749646.3749669.