Lessons from Profiling and Optimizing Placement in AMR Codes

Ankush Jain[†], Charles D. Cranor[†], Qing Zheng[‡], Dominic Manno[‡*], George Amvrosiadis[†], Gary A. Grider[‡]
[†]Carnegie Mellon University, [‡]Los Alamos National Laboratory
{ankushj, chuck, gamvrosi}@cmu.edu, {qzheng, ggrider}@lanl.gov, dmanno@nvidia.com

***Abstract*—Block-structured Adaptive Mesh Refinement (AMR), while essential for improving efficiency in large-scale irregular and dynamic simulations, poses unique optimization challenges. Previous work has identified load imbalance and synchronization overhead as key obstacles to performance, but the deep understanding of complex runtime behavior needed to systematically address them remains elusive.**

In this paper, we integrate telemetry collection, analysis, and intervention to bridge this understanding gap. Establishing reliable, actionable telemetry required systematic tuning to eliminate cross-stack performance anomalies. Leveraging this foundation we design CPLX, a tunable placement policy balancing compute load and communication locality, improving runtime by up to 21.6% over optimized baselines. Our experience highlights the empirical nature of placement optimization, requiring theoretical models to be grounded in observed runtime behavior.

***Index Terms*—Code tuning, Computational modeling, Load balancing and task assignment, Performance, Optimization**

I. Introduction

Block-structured Adaptive Mesh Refinement (AMR) codes are critical to modern scientific computing, enabling large-scale simulations in astrophysics, cosmology, climate science, and fluid dynamics [1, 2]. By dynamically refining the mesh around steep gradients, AMR delivers 1–2 orders of magnitude performance gains over uniform approaches [3]. These codes often run for weeks on hundreds of thousands of cores [2, 4, 5], making performance optimization essential. Load balancing remains a major challenge [1, 6]: while octree and space-filling curve (SFC) placements work for uniform workloads [2], they do not account for compute kernel variability. Parallel codes are limited by stragglers—ranks lagging significantly behind peers—whose delay propagates due to frequent global synchronization, and the fine-grained adaptivity of AMR exacerbates this effect. Prior analyses of proxy application traces [7] document significant MPI waiting time (>60% at 1000 ranks), while noting the limitations of standard trace data for the deeper, cross-stack diagnosis required to pinpoint root causes. The root causes, whether algorithmic or infrastructural, remain entangled. We ask: what changes when performance instrumentation has full visibility over a real AMR code?

To answer this, we sought to improve AMR placement using fine-grained runtime telemetry on a research cluster providing full-stack access. Our goal was to replace static heuristics with telemetry-driven workload models. We adopted an integrated workflow that combined telemetry collection, analysis, and intervention. However, initial experiments immediately revealed a deeper challenge: observed metrics often diverged unexpectedly from predictors such as message volume, undermining our ability to construct reliable workload models. Rather than directly enabling better placement, fine-grained telemetry first required us to diagnose and mitigate runtime artifacts to make the data useful for placement decisions (§III).

This paper describes the end-to-end process we undertook to develop telemetry-driven placement policies. We began by systematically diagnosing variability across the stack and tuning system parameters. This required extensive analysis of large volumes of telemetry, prompting a shift from standard tracing and profiling tools to a custom analytical pipeline over structured data (§IV). Tuning the system stack resulted in predictable runtime response as placement aspects such as load balance and locality were varied. Recognizing that these two properties represented competing optimization objectives, we developed CPLX (§V), a hybrid placement policy balancing these two objectives via a tunable parameter to minimize overall runtime. Because AMR codes refine frequently, placement must meet strict timing constraints ($<$ 50ms in our target codes). CPLX meets these requirements while leveraging existing octree and SFC infrastructure for compatibility with current AMR frameworks.

CPLX improves runtime by up to 21.6% over optimized baselines (§VI) through its tunable control over the load-locality tradeoff, validating our process-driven methodology. However, our experience suggests that performance optimization in large-scale systems is inherently context-dependent. While algorithmic approaches are necessary, their application must be embedded in a broader empirical process—one that interprets telemetry in the context of the runtime stack and prioritizes objectives only insofar as they measurably impact end-to-end performance. Accordingly, we focus not just on CPLX itself, but on the end-to-end process that led to it, and distill methodological lessons applicable to other large-scale tuning efforts. This paper makes three key contributions:

- An empirical case study documenting the challenges of achieving reliable, interpretable telemetry for AMR codes using full-stack access and iterative tuning.
- A novel placement policy, CPLX, that provides tunable control over the tradeoff between compute balance and communication locality (§V).
- Broader lessons on telemetry-driven performance analysis

[*]Work done while at LANL

and the empirical, context-specific nature of tuning in large-scale parallel systems.

We provide background on block-structured AMR codes and current placement approaches (§II), then outline the challenges of incorporating telemetry into placement decisions (§III). We present our methodology for obtaining reliable performance measurements (§IV), followed by the design of our placement policies (§V) and their evaluation (§VI). We then summarize the broader lessons drawn from our experiences (§VII) before concluding with related work (§VIII).

All our policies and associated artifacts are open source [8].

II. Why Telemetry-driven Placement?

Block-structured AMR simulations running on O(100K) cores face significant performance challenges due to computational variability [1, 9, 10]. 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 [11].

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 [12–15] are commonly used to mask variability, significant time is lost waiting within MPI routines despite their widespread use ($>$60% at 1,000 ranks [7]). 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 §II-A, followed by computational characteristics of AMR operations in §II-B.

A. 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 [5], Enzo-E/Cello [16], and ALPS [6]. 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 [17], SAMRAI [18]).

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 §II-B.

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++ [12], Uintah [13], HPX [14], and Legion [15]. Our work incorporating telemetry in work placement complements execution strategies masking residual imbalance.

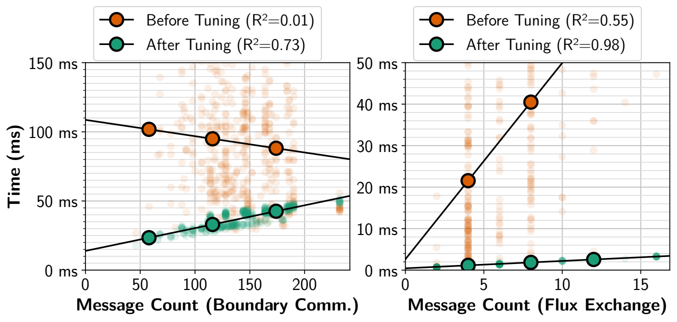
Placement Mechanisms. When a redistribution is invoked, placement mechanisms compute new block-rank assignments, and blocks are migrated accordingly. Modern block-based AMR codes rely on octree mesh structures combined with space-filling curves (SFCs) to enable efficient parallel mesh management while 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 §V where we present our placement policies.

B. Characteristics of Key AMR Operations

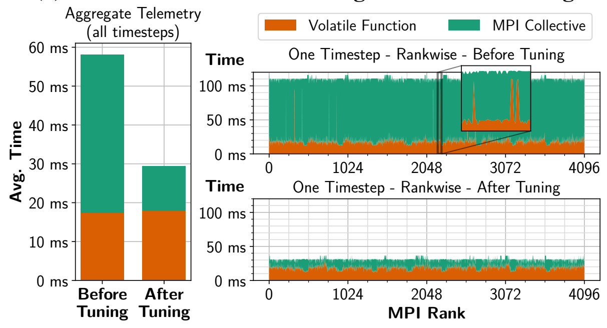
Compute Tasks. Compute tasks perform the core physical and mesh operations of the simulation. While operations like refinement tagging often have predictable costs, physics kernels are a major source of variability. For instance, regions with steep gradients may require more solver iterations, increasing 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.

Redistribution. Redistribution operations rebalance blocks across ranks as the simulation evolves, adapting to changes from mesh refinement or coarsening. They are triggered when



(a) Correlation between message counts and timing



(b) Coarse vs fine-grained telemetry from an AMR code

Fig. 1: Telemetry challenges in AMR codes. (Top) Initial telemetry shows poor correlation between work (message counts) and communication time. Tuning for system-level issues improves correlation and telemetry reliability. (Bottom) Fine-grained telemetry reveals subtle performance anomalies affecting average collective time by $3\times$.

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 [19], synchronous operations remain necessary for certain correctness guarantees, like ensuring that all ranks execute the same timestep.

III. 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. 1a), 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. 1a), our first challenge 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` [20] and eBPF [21], 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. 1b). Tracing tools [22, 23] produced unstructured, high-volume output in formats like OTF2 [23] unsuited for query-driven diagnosis. Our analytics workflow evolved organically, beginning with standard tooling such as TAU [22] 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 [24], 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: §IV describes the tuning and analytics infrastructure used to establish reliable telemetry, and §V presents placement policies enabled by this exercise.

IV. 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 (§IV-A), followed by examples from different layers of the software stack in (§IV-B). We then describe the evolution of our analysis workflow in (§IV-C), followed

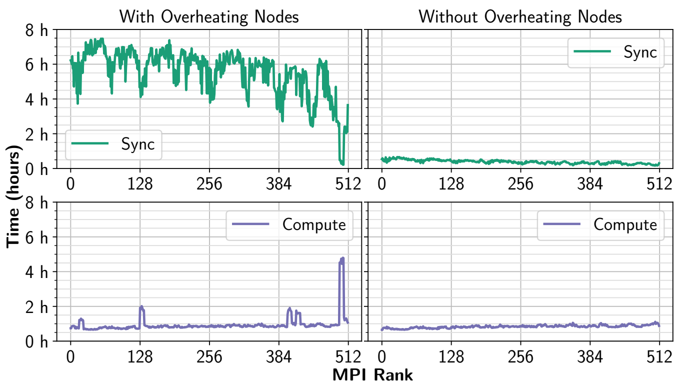


Fig. 2: Profiling data from early runs affected by CPU throttling. Compute times on impacted ranks were inflated by up to 4× and appeared in clusters of 16, leading to elevated synchronization delays across the system. Pruning affected nodes reduced overall runtime by 3× by lowering synchronization overhead

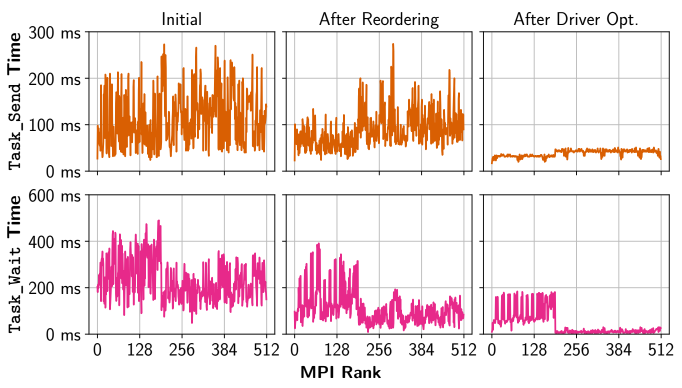


Fig. 3: Rankwise boundary communication performance before and after two optimizations. Prioritizing sends in the task schedule reduced noise, revealing faint performance trends. Subsequent queue size tuning further reduced variance, clarifying the underlying telemetry structure.

by a critical path model to formally analyze the impact of task dependencies and scheduling on runtime in §IV-D. 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 [5], Phoebus [25], and AthenaPK [5], built against MVAPICH2 [26] and the PSM fabric library [27].

A. 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. 2). 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.

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 [28, 29]. 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.

B. Tuning The Software Stack

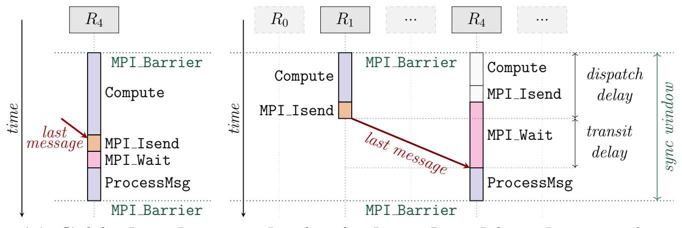
We found interactions within the software stack—application logic, MPI runtime, and network drivers—to be the primary source of performance variability impacting telemetry reliability. While tools exist to check for basic configuration errors or known MPI misuses [30], they cannot capture the dynamic, 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. 1b shows the spikes and the impact of our mitigations.

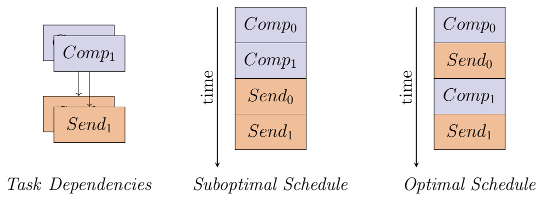
Application-Level Example: Task Reordering. In another AMR code, MPI send tasks were scheduled after compute/wait tasks, causing cascading delays on CPUs (masked on GPUs where developed) Prioritizing sends (Fig. 3, middle) unblocked dependent ranks without affecting senders, reducing wait times and jitter.

Network-Level Example: Queue Size Tuning. Persistent variance, contributing to poor correlation between communication time and work (Fig. 1a), was traced using `perf` [20] 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. 3, 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 in-



(a) Critical path examples in single and multi-rank scenarios



(b) Impact of send ordering on task scheduling

Fig. 4: Critical paths within a synchronization window. (Top) Single- vs two-rank critical paths, demonstrating how delay chains can be purely local or involve exactly two ranks with one P2P communication round. (Bottom) 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.

feasible. Systematically identifying root causes—correlating events across ranks, time, application phases, and system layers—necessitated programmable low-level interfaces like eBPF [21], 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.

C. Evolution of our Analysis Workflow

Addressing the analytical limitations described above required evolving our workflow substantially beyond standard tools: TAU [22] 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 [31] and Kokkos [32] profiling interfaces. Eventually, we outgrew pandas' analytical bandwidth and found our telemetry queries naturally mapped to SQL over data ingested into ClickHouse [33], 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. We explore the broader implications of this model in §VII.

D. 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 §IV-B, but also helps formally identify other avenues for optimization. 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* [34] 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. 4a. This model reveals two strategies to shorten such paths:

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 §II-A 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.

V. Designing a Placement Policy

With the reliable and interpretable telemetry foundation established in §IV, 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 [24], necessitating efficient heuristics.

Communication Locality. Baseline SFC-based placements (§V-A) 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 §V-A, followed by load-centric and locality-preserving policies in §V-B and §V-C. We conclude with CPLX (§V-D), a hybrid policy that enables tunable control over the tradeoff between locality and load balance. All policies were implemented in the Parthenon AMR framework [5] and are directly available in Parthenon-based codes such as Phoebus [25] and AthenaPK [5].

A. Existing Placement Infrastructure

Modern block-based AMR codes use octrees and space-filling curves (SFCs) as foundational data structures for mesh management. This section describes these building blocks and the baseline placement policy, as understanding their strengths and limitations is crucial for developing better policies.

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. 5. 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.

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

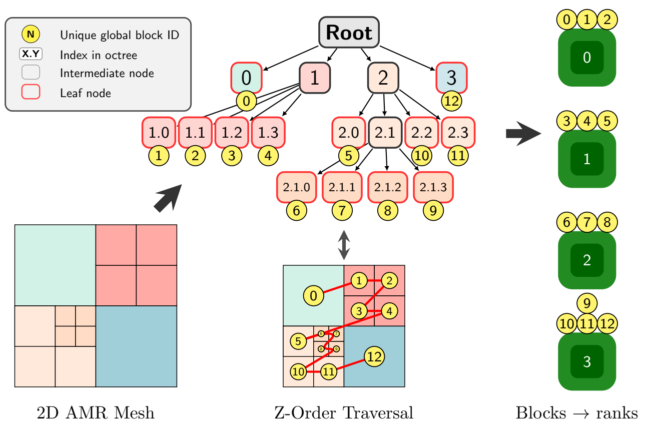


Fig. 5: Example of an adaptively refined mesh, octrees, and Z-order Space-Filling Curves (SFCs) in two dimensions. Mesh blocks correspond to octree nodes with refinement creating child nodes. Sequential block IDs are assigned to leaf blocks using a depth-first traversal, equivalent to a Z-Order SFC. Contiguous block ID ranges are then assigned to simulation ranks to balance block counts while preserving locality.

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.

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.

B. 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 [35]. In practice, LPT performs remarkably well; we could not obtain better solutions from a commercial ILP solver [36] 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.

C. 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 DP$[i][k]$ for partitioning i blocks among k ranks follows (DP$[0][0] = 0$):

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

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.

D. 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.

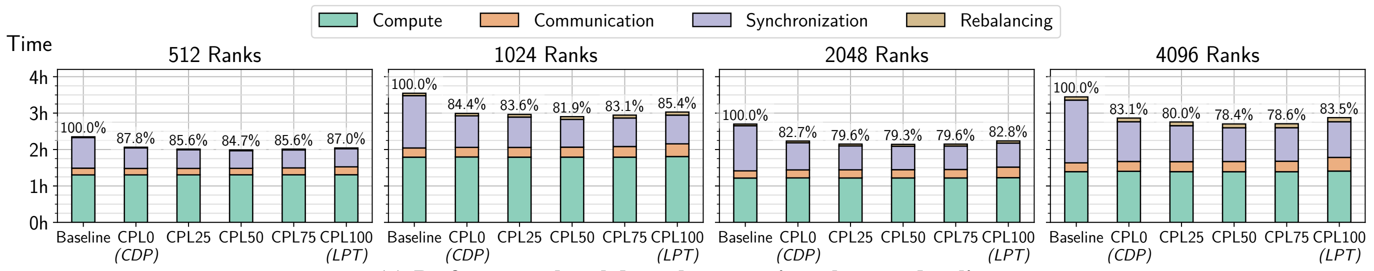
Developing such a hybrid proved challenging. Our initial attempts to blend the policies produced unpredictable results—small sacrifices in load balance did not translate to proportional gains in locality, and vice versa. We eventually realized that it was easier to selectively break locality in a contiguous 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.

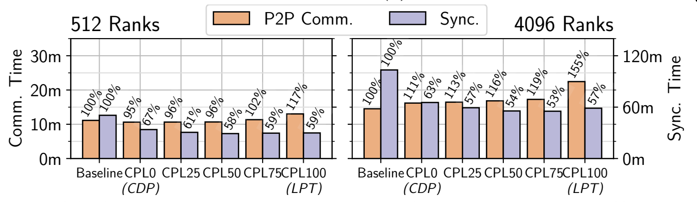
The parameter X, ranging from 0 to 100%, provides precise control over this locality disruption. At X=0 (`CPL0`), no ranks participate in rebalancing, preserving the locality characteristics of CDP. At X=100 (`CPL100`), all ranks undergo LPT rebalancing, effectively becoming pure LPT. Intermediate values 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.

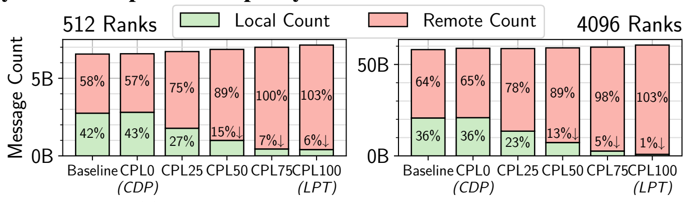
VI. Evaluation of Placement Policies

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.



(a) Performance breakdown by execution phase and policy





(b) Communication and synchronization time tradeoffs **(c) Locality analysis by message volume**

Fig. 6: Runtime statistics comparing different load-balancing policies across multiple scales (512–4096 ranks). (Top) Total runtime, classified into phases. CPL50 performs best, with up to 21.6% runtime reduction over baseline. (Bottom, left) 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. (Bottom, right) Message patterns show that increasing X reduces local in favor of remote communication, providing evidence for the locality-performance tradeoff.

RANKS	MESH SIZE	t_{total}	t_{lb}	$n_{initial}$	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 I: 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.

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 [25], a hydrodynamics code built on top of the Parthenon [5] framework. While we also studied placement in other codes, such as a galaxy cooling setup in AthenaPK [5], results were directionally similar: codes with high compute variability benefit more from better placement, and vice-versa. We describe our hardware setup in §VI-A, present Sedov results in §VI-B, and use microbenchmarks to evaluate CPLX's load balance and locality properties under different synthetic regimes in §VI-C.

A. Experimental Setup

We evaluate four configurations of the Sedov Blast Wave 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 §IV, with 16 ranks per node, at scales from 512–4096 ranks. Telemetry was collected using the custom profiling stack described in §III. 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 §IV. 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 [26], compiled against a tuned version of PSM [27], the userspace library for our QLogic fabric.

B. 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. 6a), the synchronization—communication tradeoff (Fig. 6b), and P2P communication patterns (Fig. 6c).

Finding 1: *Baseline: synchronization reaches 50% of total runtime at scale.*

Figure 6a 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 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.*

Figure 6a 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.*

Figure 6b 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.

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 (§V-B), 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. 6c 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 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 §VI-C using synthetic microbenchmarks.

A subtler effect visible in Fig. 6c 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. 6b.

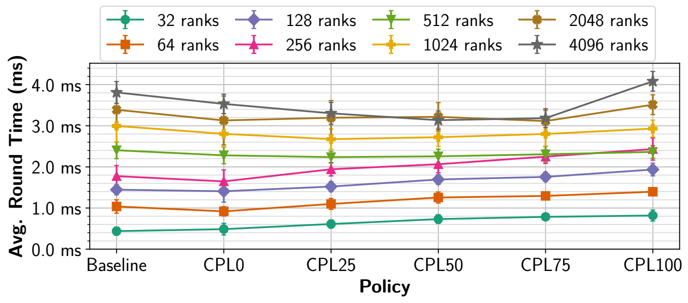
Finally, even the baseline placement routes a majority of messages across nodes: at 4096 ranks, **64%** of messages are already remote. This is an intrinsic property of space-filling curves—dimensionality reduction to 1-D preserves only partial spatial locality. CPLX does not introduce a new communication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

C. Microbenchmarks

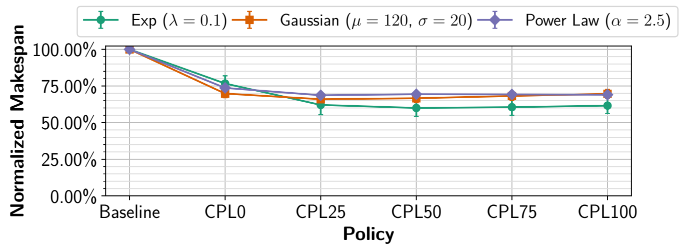
To complement the application-level Sedov results (§VI-B) and enable controlled evaluation across a broader range of conditions, we developed two microbenchmarks. `commbench` isolates P2P communication under varying placements, while `scalebench` evaluates the effectiveness and computational cost of 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 [37], `commbench` simulates a full AMR placement pipeline and accepts custom placement policies as drop-in modules.

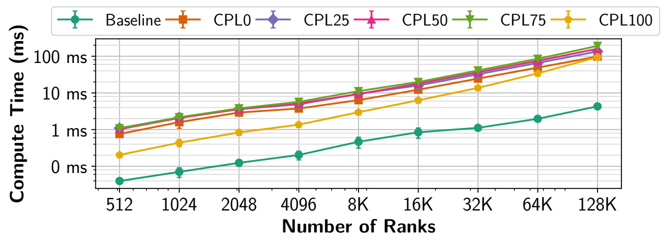
Each communication round emulates a boundary exchange, with blocks exchanging messages based on neighbor type. Message sizes are realistic: face-neighbor exchanges are proportionally larger than edge or vertex ones. Meshes are refined to yield 1–2 blocks per rank, and each round includes all neighbor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



(a) Boundary communication latency vs. policies (`commbench`)



(b) Load balance quality vs. polices (`scalebench`)



(c) Policy compute cost vs. simulated rank count (`scalebench`)

Fig. 7: (Top) Boundary communication round latency vs. locality at different scales, measured using `commbench`. 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. (Middle) Average makespans from CPLX placements with three synthetic distributions. While LPT performs best across all distributions, the bulk of the benefits are realized with $X = 25$. (Bottom) 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.

and those with latency above 10 ms, which we found to reflect fabric-level recovery behavior unrelated to placement.

Figure 7a 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 values of X 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.

Figure 7b 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. Figure 7c 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 [38].

VII. Lessons From Our Placement Exercise

We distill our takeaways from this exercise into the following five lessons.

Lesson 1: *AMR Performance Is Straggler Mitigation.*

High synchronization costs in AMR codes are often attributed to load imbalance. Our experience suggests that this attribution may be premature: stragglers can also arise from tuning artifacts, execution strategy interactions, or hardware-level variability. Placement strategies cannot compensate for unstable system behavior—such instability must be identified and resolved independently before placement can be meaningfully evaluated.

Lesson 2: *Effective Tuning is Empirical and Idiosyncratic.*

Performance tuning is not about correctness, but about aligning system behavior with the workload. Default configurations reflect common-case assumptions that may vary across codes. Tuning, therefore, is both empirical and workload-specific [39–41]. It improves performance directly, while also making application behavior more predictable under placement changes. In our experience, progressive tuning clarified the telemetry structure, reduced confounding noise, and exposed deeper residual issues that would otherwise be misattributed.

Lesson 3: *P2P Communication Is A Key Variability Source.*

Our tuning efforts primarily focused on point-to-point (P2P) communication patterns used in ghost cell and halo exchanges. Unlike collectives, which are structured and easy to benchmark in isolation, P2P exchanges are fine-grained, latency-sensitive, and highly variable. They involve dynamic neighbor counts, varying message sizes, and inconsistent communication paths—each of which can activate different code paths in the network stack. We found these interactions to be the dominant source of variability, and they required targeted attention during tuning.

Lesson 4: *Diagnosis Needs Structured, Queryable Telemetry.*

Understanding why performance deviated required analytics over fine-grained telemetry to surface anomalous low-level behavior. While tracing tools capture such detail, they rely on unstructured formats like JSON [42] or OTF2 [43], which are poorly suited to multi-dimensional analysis across rank, time, and task. We converged on SQL-based analytics over telemetry grouped by timestep and sorted by rank, enabling views of application behavior aligned with synchronization intervals.

While tools like Caliper [44] do support structured telemetry, they—like our ad hoc pipeline—rely on parsing and postprocessing of plaintext data, which we found to slow down our iterative tuning workflows. Binary columnar formats like Arrow [45] and Parquet [46], when paired with in-situ collection, offer a promising foundation for low-latency BSP telemetry by enabling low-overhead parsing and efficient querying via embedded statistics over partitioned data.

Lesson 5: *Placement Optimization is Systems Engineering.*

AMR placement policy design is a systems engineering problem, guided primarily by empirical constraints rather than solely by theoretical optimality. Tradeoffs between compute balance, communication locality, and placement overhead must be evaluated based on observed performance impact, not static preferences. Placement strategies need to be empirically calibrated to each specific deployment context. Their computation must also respect strict time budgets—in our case, under 50 ms per redistribution—which constrains the complexity of the algorithms and necessitates focus on dominant effects.

VIII. Related Work

Variability and Runtime Analysis. Performance variability in HPC environments is well documented [9, 11, 47–49], with root causes ranging from tuning to contention. Separately, fail-slow hardware has been reported as a source of performance degradation [28, 29]. Similar variability challenges have been identified in the context of collective performance in hyper-scale ML clusters [50]. Varbench [51] is a framework to characterize computational variability using a suite of compute kernels. Klenk et al. [7] 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 [22] and Score-P [23] collect profiles and traces (e.g., OTF2), but lack the queryability required for scalable analytics, a limitation previously noted by Klenk et al. [7]. Caliper [44] supports structured telemetry and programmable aggregation, but relies on plaintext data formats. Parquet [46] provides a compact, binary columnar format with embedded statistics for efficient analytics, but is not widely adopted in performance tools. Hatchet [52] builds a structured in-memory representation from external telemetry sources but assumes prior parsing. Metric frameworks like LDMS [53] offer low-overhead system-level metrics, but often lack application context needed for bottleneck attribution. Yang et al. [54] describe a general critical path analysis framework, while our framework in §IV-D is tailored for AMR codes.

AMR and Communication Models. Modern AMR codes either use MPI or asynchronous runtimes like Charm++ [12] and Legion [15]. Neighborhood collectives [47, 55, 56] 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 [57] and Zoltan [58] handle complex meshes but incur high overhead. Sasidharan et al. [59] 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. Meta-Balancer [60] focuses on rebalancing triggers, while Zheng et al. [38] propose hierarchical schemes to scale load balancing. Solver-based approaches are used to optimize placement in cloud datacenters [61], but are computationally expensive (multi-hour) and impractical for AMR.

IX. Conclusions

Two decades ago, Petrini et al. [11] traced *missing performance* in flagship supercomputers to noise and variability across the system stack. Our analysis of contemporary AMR codes shows this diagnostic challenge remains deeply relevant. Modern system complexity, driven by adaptive behaviors tuned for common cases, means performance is highly variable and context-dependent. As a result, simple attributions to load imbalance are often incomplete, obscuring deeper sources of variability. Systematically identifying and mitigating these issues across hardware and software layers was crucial for effective optimization. Only after establishing reliable, interpretable telemetry could our CPLX placement policy demonstrate tunable control over the load-locality tradeoff, achieving up to 21.6% runtime improvement over tuned baselines. Our experience suggests that low-latency analytics built on relational telemetry models may offer a viable foundation for resolving the persistent opacity of performance root causes.

Acknowledgments

We thank the anonymous reviewers of our paper for their valuable comments on improving the manuscript. We are grateful to Jonah Miller, Philipp Grete, Phil Carns, and Rob Ross for their support and guidance throughout this work. George Amvrosiadis holds concurrent appointments at Carnegie Mellon University (CMU) and Amazon Web Services—this paper describes work performed at CMU and is not associated with Amazon. This manuscript has been approved for unlimited release and has been assigned LA-UR-25-27334. We also thank the member companies of the PDL Consortium (Amazon, Google, Honda, IBM Research, Intel, Jane Street, Meta, Microsoft Research, Oracle, Pure Storage, Salesforce, Samsung Semiconductor, Two Sigma, and Western Digital) for their interest, insights, feedback, and support.

References

[1] A. Dubey et al., “A survey of high level frameworks in block-structured adaptive mesh refinement packages,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3217–3227, Dec. 2014, ISSN: 07437315. DOI: 10.1016/j.jpdc.2014.07.001. Accessed: May 15, 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0743731514001178.

[2] C. Burstedde, L. C. Wilcox, and O. Ghattas, “P4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees,” *SIAM Journal on Scientific Computing*, vol. 33, no. 3, pp. 1103–1133, Jan. 2011, ISSN: 1064-8275. DOI: 10.1137/100791634. Accessed: May 15, 2024. [Online]. Available: https://epubs.siam.org/doi/10.1137/100791634.

[3] M. P. Vittitow, “A Decade of Adaptive Mesh Refinement in CTH,”

[4] M. T. P. Liska et al., “H-AMR: A New GPU-accelerated GRMHD Code for Exascale Computing with 3D Adaptive Mesh Refinement and Local Adaptive Time Stepping,” *The Astrophysical Journal Supplement Series*, vol. 263, no. 2, p. 26, Nov. 2022, ISSN: 0067-0049. DOI: 10.3847/1538-4365/ac9966. Accessed: Oct. 25, 2024. [Online]. Available: https://dx.doi.org/10.3847/1538-4365/ac9966.

[5] P. Grete et al., “Parthenon—a performance portable block-structured adaptive mesh refinement framework,” *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, pp. 465–486, Sep. 1, 2023, ISSN: 1094-3420. DOI: 10.1177/10943420221143775. Accessed: Aug. 21, 2024. [Online]. Available: https://doi.org/10.1177/10943420221143775.

[6] C. Burstedde et al., “Scalable adaptive mantle convection simulation on petascale supercomputers,” in *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin, TX, USA: IEEE, Nov. 2008, pp. 1–15, ISBN: 978-1-4244-2834-2. DOI: 10.1109/SC.2008.5214248. Accessed: Jul. 28, 2023. [Online]. Available: http://ieeexplore.ieee.org/document/5214248/.

[7] B. Klenk and H. 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*, Berlin, Heidelberg: Springer-Verlag, Jun. 18, 2017, pp. 217–236, ISBN: 978-3-319-58666-3. DOI: 10.1007/978-3-319-58667-0_12. Accessed: Aug. 1, 2025. [Online]. Available: https://doi.org/10.1007/978-3-319-58667-0_12.

[8] Ankush Jain, *AMR Tools*, Aug. 8, 2025. [Online]. Available: https://github.com/pdlfs/amr-tools.

[9] B. Van Straalen, J. Shalf, T. Ligocki, N. Keen, and Woo-Sun Yang, “Scalability challenges for massively parallel AMR applications,” in *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, Italy: IEEE, May 2009, pp. 1–12, ISBN: 978-1-4244-3751-1. DOI: 10.1109/IPDPS.2009.5161014. Accessed: Oct. 25, 2024. [Online]. Available: http://ieeexplore.ieee.org/document/5161014/.

[10] F. Schornbaum and U. Rüde, “Extreme-Scale Block-Structured Adaptive Mesh Refinement,” *SIAM Journal on Scientific Computing*, vol. 40, no. 3, pp. C358–C387, Jan. 2018, ISSN: 1064-8275, 1095-7197. DOI: 10.1137/17M1128411. arXiv: 1704.06829 [cs]. Accessed: Jun. 7, 2024. [Online]. Available: http://arxiv.org/abs/1704.06829.

[11] F. Petrini, D. J. Kerbyson, and S. 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*, Phoenix AZ USA: ACM, Nov. 15, 2003, p. 55. DOI: 10.1145/1048935.1050204. Accessed: May 1, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/1048935.1050204.

[12] B. 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*, Nov. 2014, pp. 647–658. DOI: 10.1109/SC.2014.58. Accessed: Sep. 30, 2024. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7013040.

[13] Q. Meng, J. Luitjens, and M. Berzins, “Dynamic task scheduling for the Uintah framework,” in *2010 3rd Workshop on Many-Task Computing on Grids and Supercomputers*, Nov. 2010, pp. 1–10. DOI: 10.1109/MTAGS.2010.5699431. Accessed: Sep. 26, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/5699431/?arnumber=5699431.

[14] G. Daiß et al. “Asynchronous-Many-Task Systems: Challenges and Opportunities – Scaling an AMR Astrophysics Code on Exascale machines using Kokkos and HPX.” arXiv: 2412.15518 [cs], Accessed: Jan. 31, 2025. [Online]. Available: http://arxiv.org/abs/2412.15518, pre-published.

[15] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *SC ’12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, Nov. 2012, pp. 1–11. DOI: 10.1109/SC.2012.71. Accessed: Jan. 31, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/6468504.

[16] A. Emerick, “Enzo-E/Cello Project: Enabling Exa-Scale Astrophysics,” 2019.

[17] W. Zhang, A. Myers, K. Gott, A. Almgren, and J. Bell, “AMReX: Block-structured adaptive mesh refinement for multiphysics applications,” *The International Journal of High Performance Computing Applications*, vol. 35, no. 6, pp. 508–526, Nov. 1, 2021, ISSN: 1094-3420. DOI: 10.1177/10943420211022811. Accessed: Jun. 19, 2024. [Online]. Available: https://doi.org/10.1177/10943420211022811.

[18] B. T. Gunney and R. W. Anderson, “Advances in patch-based adaptive mesh refinement scalability,” *Journal of Parallel and Distributed Computing*, vol. 89, pp. 65–84, Mar. 2016, ISSN: 07437315. DOI: 10.1016/j.jpdc.2015.11.005. Accessed: Jun. 7, 2024. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0743731515002129.

[19] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine, “A Case for Standard Non-blocking Collective Operations,” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, F. Cappello, T. Herault, and J. Dongarra, Eds., vol. 4757, Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 125–134, ISBN: 978-3-540-75415-2 978-3-540-75416-9. DOI: 10.1007/978-3-540-75416-9_22. Accessed: May 2, 2025. [Online]. Available: http://link.springer.com/10.1007/978-3-540-75416-9_22.

[20] Linux Kernel Developers, *Linux perf: Performance analysis tools for linux*, Accessed: 2025-02-05, 2025. [Online]. Available: https://perfwiki.github.io/main/.

[21] Linux Kernel Developers, *Ebpf: Extended berkeley packet filter*, Accessed: 2024-02-05, 2024. [Online]. Available: https://ebpf.io.

[22] S. S. Shende and A. D. Malony, “The Tau Parallel Performance System,” *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, May 1, 2006, ISSN: 1094-3420. DOI: 10.1177/1094342006064482. Accessed: Feb. 3, 2025. [Online]. Available: https://doi.org/10.1177/1094342006064482.

[23] D. an Mey et al., “Score-P: A Unified Performance Measurement System for Petascale Applications,” in *Competence in High Performance Computing 2010*, C. Bischof, H.-G. Hegering, W. E. Nagel, and G. Wittum, Eds., Berlin, Heidelberg: Springer, 2012, pp. 85–97, ISBN: 978-3-642-24025-6. DOI: 10.1007/978-3-642-24025-6_8.

[24] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. USA: W. H. Freeman & Co., 1979, ISBN: 0-7167-1044-7.

[25] B. Barker et al. “Phoebus: Performance Portable GRRMHD for Relativistic Astrophysics.” version 2. arXiv: 2410.09146 [astro-ph], Accessed: Feb. 7, 2025. [Online]. Available: http://arxiv.org/abs/2410.09146, pre-published.

[26] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, “The MVAPICH project: Transforming research into high-performance MPI library for HPC community,” *Journal of Computational Science*, Case Studies in Translational Computer Science, vol. 52, p. 101 208, May 1, 2021, ISSN: 1877-7503. DOI: 10.1016/j.jocs.2020.101208. Accessed: Feb. 5, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750320305093.

[27] *Intel/psm*, Intel® Corporation, Jan. 27, 2023. Accessed: Feb. 5, 2025. [Online]. Available: https://github.com/intel/psm.

[28] H. S. Gunawi et al., “Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems,” *ACM Transactions on Storage*, vol. 14, no. 3, pp. 1–26, Aug. 31, 2018, ISSN: 1553-3077, 1553-3093. DOI: 10.1145/3242086. Accessed: Dec. 4, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3242086.

[29] R. Lu et al., “PERSEUS: A Fail-Slow Detection Framework for Cloud Storage Systems,” 2023.

[30] T. Hilbrich, M. Schulz, B. R. De Supinski, and M. S. Müller, “MUST: A Scalable Approach to Runtime Error Detection in MPI Programs,” in *Tools for High Performance Computing 2009*, M. S. Müller, M. M. Resch, A. Schulz, and W. E. Nagel, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 53–66, ISBN: 978-3-642-11260-7 978-3-642-11261-4. DOI: 10.1007/978-3-642-11261-4_5. Accessed: Apr. 29,

2025. [Online]. Available: https://link.springer.com/10.1007/978-3-642-11261-4_5.

[31] MPI Forum, *Mpi: A message-passing interface standard, version 3.0*, https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, Chapter 14.2: Profiling Interface, 2012.

[32] S. D. Hammond, C. R. Trott, D. Ibanez, and D. Sunderland, "Profiling and Debugging Support for the Kokkos Programming Model," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds., vol. 11203, Cham: Springer International Publishing, 2018, pp. 743–754, ISBN: 978-3-030-02464-2 978-3-030-02465-9. DOI: 10.1007/978-3-030-02465-9_53. Accessed: May 2, 2025. [Online]. Available: http://link.springer.com/10.1007/978-3-030-02465-9_53.

[33] R. Schulze, T. Schreiber, I. Yatsishin, R. Dahimene, and A. Milovidov, "ClickHouse - Lightning Fast Analytics for Everyone," *Proceedings of the VLDB Endowment*, vol. 17, no. 12, pp. 3731–3744, Aug. 2024, ISSN: 2150-8097. DOI: 10.14778/3685800.3685802. Accessed: Apr. 29, 2025. [Online]. Available: https://dl.acm.org/doi/10.14778/3685800.3685802.

[34] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," vol. 21, no. 7, 1978.

[35] R. L. Graham, "Bounds on Multiprocessing Timing Anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, no. 2, pp. 416–429, Mar. 1969, ISSN: 0036-1399. DOI: 10.1137/0117039. Accessed: Feb. 3, 2025. [Online]. Available: https://epubs.siam.org/doi/10.1137/0117039.

[36] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2024. [Online]. Available: https://www.gurobi.com.

[37] M. P. Vittitow, "Structural Simulation Toolkit (SST)," 2015.

[38] G. Zheng, A. Bhatelé, E. Meneses, and L. V. Kalé, "Periodic hierarchical load balancing for large supercomputers," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 371–385, Nov. 2011, ISSN: 1094-3420, 1741-2846. DOI: 10.1177/1094342010394383. Accessed: Feb. 7, 2025. [Online]. Available: https://journals.sagepub.com/doi/10.1177/1094342010394383.

[39] D. A. Kruse, W. Schonbein, and M. G. F. Dosanjh, "A Statistical Analysis of HPC Network Tuning," in *Proceedings of the SC '23 Workshops of the International Conference on High Performance Computing, Network, Storage, and Analysis*, Denver CO USA: ACM, Nov. 12, 2023, pp. 458–465. DOI: 10.1145/3624062.3624114. Accessed: May 1, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3624062.3624114.

[40] R. Mijaković, M. Firbach, and M. Gerndt, "An architecture for flexible auto-tuning: The Periscope Tuning Framework 2.0," in *2016 2nd International Conference on Green High Performance Computing (ICGHPC)*, Feb. 2016, pp. 1–9. DOI: 10.1109/ICGHPC.2016.7508066. Accessed: May 1, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7508066/.

[41] S. Pellegrini, R. Prodan, and T. Fahringer, "Tuning MPI Runtime Parameter Setting for High Performance Computing," in *2012 IEEE International Conference on Cluster Computing Workshops*, Sep. 2012, pp. 213–221. DOI: 10.1109/ClusterW.2012.15. Accessed: May 1, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/6355867/.

[42] Google. "Catapult - The trace-event file format," Accessed: Aug. 8, 2025. [Online]. Available: https://chromium.googlesource.com/catapult/+/HEAD/docs/trace-event-format.md.

[43] Eschweiler Dominic, Wagner Michael, Geimer Markus, Kn&uuml;pfer Andreas, Nagel Wolfgang E., and Wolf Felix, "Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries," in *Advances in Parallel Computing*, IOS Press, 2012. DOI: 10.3233/978-1-61499-041-3-481. Accessed: Jul. 15, 2025. [Online]. Available: https://www.medra.org/servlet/aliasResolver?alias=iospressISSNISBN&issn=0927-5452&volume=22&spage=481.

[44] D. 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*, Nov. 2016, pp. 550–560. DOI: 10.1109/SC.2016.46. Accessed: Aug. 6, 2025. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7877125.

[45] "Apache Arrow," Apache Arrow, Accessed: May 26, 2025. [Online]. Available: https://arrow.apache.org/.

[46] "Apache Parquet," Apache Parquet, Accessed: May 26, 2025. [Online]. Available: https://parquet.apache.org/docs/overview/.

[47] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT: IEEE, Nov. 2012, pp. 1–10, ISBN: 978-1-4673-0805-2 978-1-4673-0806-9. DOI: 10.1109/SC.2012.86. Accessed: Jun. 11, 2024. [Online]. Available: http://ieeexplore.ieee.org/document/6468467/.

[48] O. Yildiz, M. Dorier, S. Ibrahim, R. Ross, and G. Antoniu, "On the Root Causes of Cross-Application I/O Interference in HPC Storage Systems," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2016, pp. 750–759. DOI: 10.1109/IPDPS.2016.50. Accessed: Jul. 4, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/7516071.

[49] B. Acun, P. Miller, and L. V. Kale, "Variation Among Processors Under Turbo Boost in HPC Systems," in *Proceedings of the 2016 International Conference on Supercomputing*, Istanbul Turkey: ACM, Jun. 2016, pp. 1–12, ISBN: 978-1-4503-4361-9. DOI: 10.1145/2925426.2926289. Accessed: Feb. 5, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/2925426.2926289.

[50] A. Gangidi et al., "RDMA over Ethernet for Distributed Training at Meta Scale," in *Proceedings of the ACM SIGCOMM 2024 Conference*, Sydney NSW Australia: ACM, Aug. 4, 2024, pp. 57–70, ISBN: 9798400706141. DOI: 10.1145/3651890.3672233. Accessed: Jan. 28, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3651890.3672233.

[51] B. Kocoloski and J. Lange, "Varbench: An Experimental Framework to Measure and Characterize Performance Variability," in *Proceedings of the 47th International Conference on Parallel Processing*, Eugene OR USA: ACM, Aug. 13, 2018, pp. 1–10, ISBN: 978-1-4503-6510-9. DOI: 10.1145/3225058.3225125. Accessed: Oct. 17, 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3225058.3225125.

[52] A. Bhatele, S. Brink, and T. Gamblin, "Hatchet: Pruning the overgrowth in parallel profiles," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver Colorado: ACM, Nov. 17, 2019, pp. 1–21. DOI: 10.1145/3295500.3356219. Accessed: Aug. 6, 2025. [Online]. Available: https://dl.acm.org/doi/10.1145/3295500.3356219.

[53] B. Schwaller, "Integrated System and Application Continuous Performance Monitoring and Analysis Capability (Final)," SAND2021-11954, 1822583, 700207, Sep. 1, 2021, SAND2021–11 954, 1822583, 700207. Accessed: May 2, 2025. [Online]. Available: https://www.osti.gov/servlets/purl/1822583/.

[54] C.-Q. Yang and B. Miller, "Critical path analysis for the execution of parallel and distributed programs," in *[1988] Proceedings. The 8th International Conference on Distributed*, San Jose, CA, USA: IEEE Comput. Soc. Press, 1988, pp. 366–373, ISBN: 978-0-8186-0865-0. DOI: 10.1109/DCS.1988.12538. Accessed: Feb. 2, 2025. [Online]. Available: http://ieeexplore.ieee.org/document/12538/.

[55] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff, "The scalable process topology interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, 2011, ISSN: 1532-0634. DOI: 10.1002/cpe.1643. Accessed: Jun. 11, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1643.

[56] S. M. Ghazimirsaeed, "Efficient Designs for Distributed MPI Neighborhood Collectives,"

[57] D. Lasalle and G. Karypis, "Multi-threaded Graph Partitioning," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, May 2013, pp. 225–236. DOI: 10.1109/IPDPS.2013.50. Accessed: Jun. 11, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/6569814.

[58] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and Lee Ann Riesen, "Hypergraph-based Dynamic Load Balancing for Adaptive Scientific Computations," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, USA: IEEE, 2007, pp. 1–11, ISBN: 978-1-4244-0909-9. DOI: 10.1109/IPDPS.2007.370258. Accessed: Jun. 11, 2024. [Online]. Available: http://ieeexplore.ieee.org/document/4227986/.

[59] A. Sasidharan, J. M. Dennis, and M. 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*, New York, NY: IEEE, Aug. 2015, pp. 875–879, ISBN: 978-1-4799-8937-9. DOI: 10.1109/HPCC-CSS-ICESS.2015.192. Accessed: Nov. 6, 2023. [Online]. Available: https://ieeexplore.ieee.org/document/7336274/.

[60] H. Menon, N. Jain, G. Zheng, and L. Kalé, “Automated Load Balancing Invocation Based on Application Characteristics,” in *2012 IEEE International Conference on Cluster Computing*, Sep. 2012, pp. 373–381. DOI: 10.1109/CLUSTER.2012.61. Accessed: Oct. 25, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/6337800/?arnumber=6337800.

[61] A. Newell et al., “RAS: Continuously Optimized Region-Wide Datacenter Resource Allocation,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP ’21, New York, NY, USA: Association for Computing Machinery, Oct. 26, 2021, pp. 505–520, ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483578. Accessed: Jul. 27, 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3477132.3483578.