

Workflows Using Artificial Intelligence

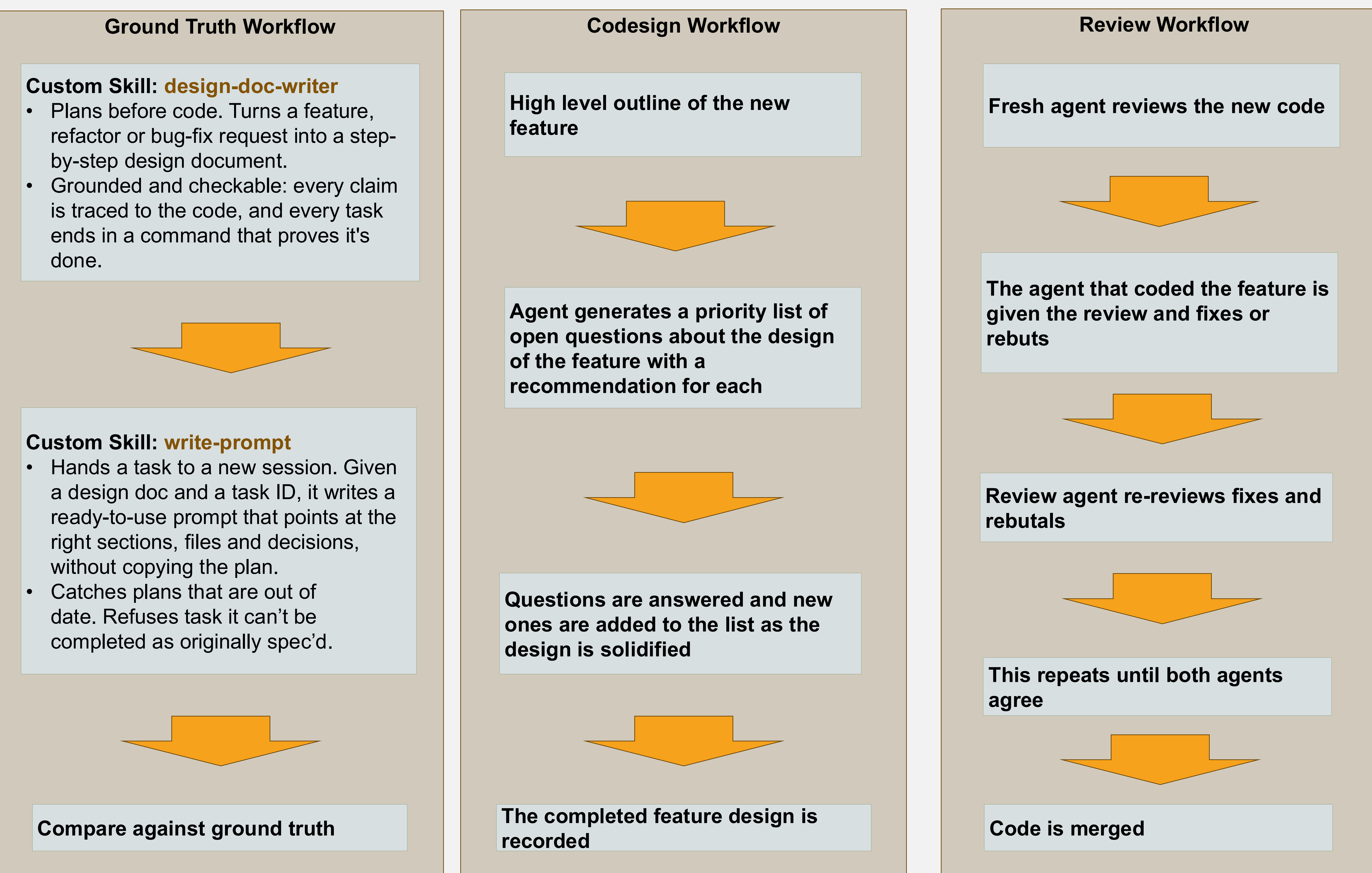
Jason Stewart and Thomas Hines

How can we harness AI to support PSAAP?

- With each new model update, AI becomes more capable.
- Our workflows are constantly in flux to adjust for the intelligence of new models.
- Optimal workflows now may be vastly different in six months

Useful Findings and Conclusions

1. Developing with a known ground truth works well
2. In the process of learning how well it writes optimal code.
3. The best use of is getting more done in the same amount of time rather than one thing done in less time. Weak scaling versus strong scaling.



Open Questions and Concerns

- How well do we need to understand the code?
 - Every single line? The general flow?
 - How do we know the tests are correct?
- AI agents produce lengthy comments
 - On one hand, this gives it needed context
 - On the other, this is annoying to human coders
- Code Bloat and Hygiene
 - Agents often take the simplest route, not the best
 - Does this cause actual performance issues?
- How quickly will our programming knowledge get rusty?
 - Do we just move up a layer?

Acknowledgements

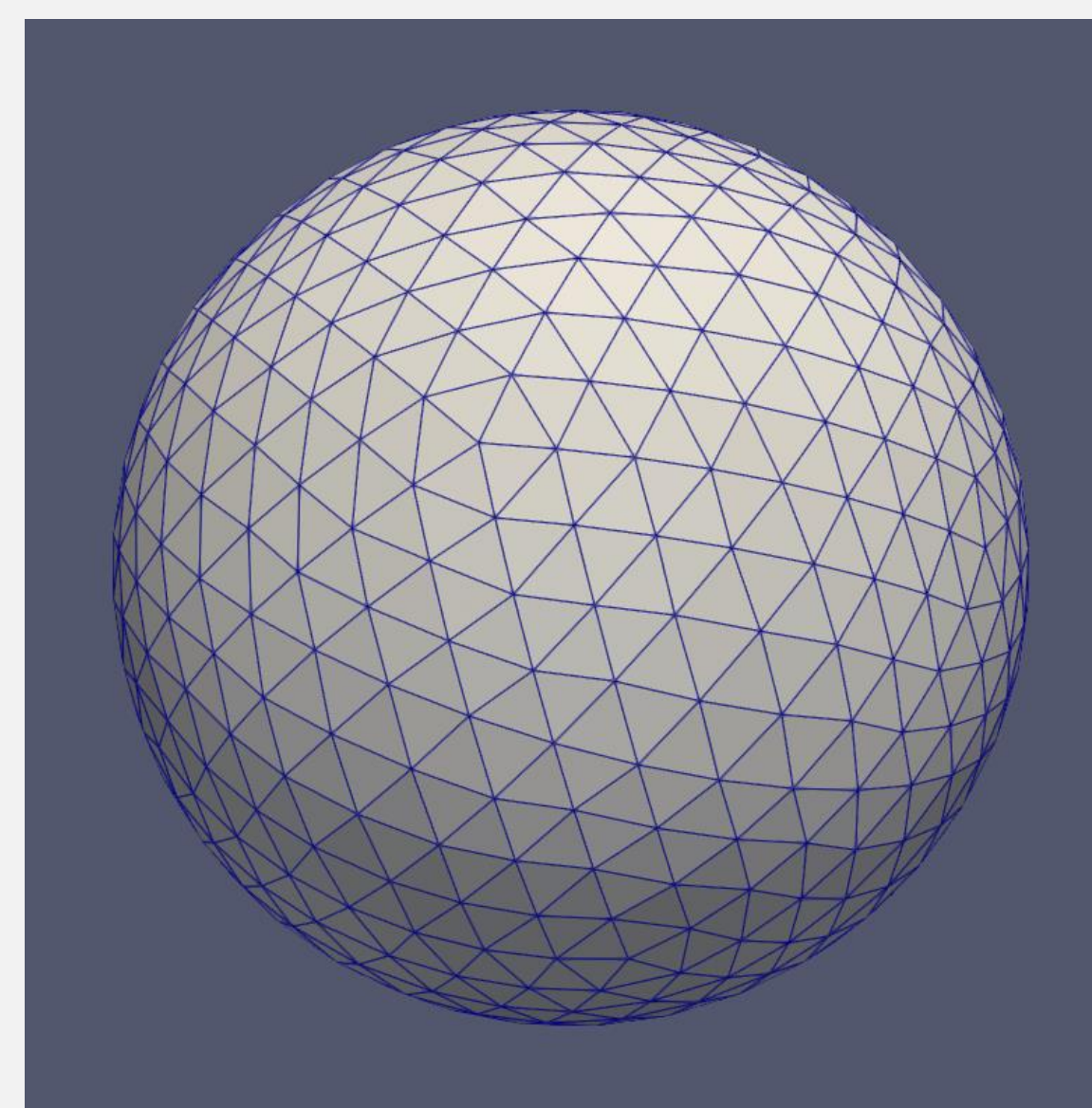
This work was supported in part by the U.S. National Science Foundation under awards 2511818, 2514054, 2450093, 2450092, 2412182, and by the U.S. Department of Energy, National Nuclear Security Administration, under the Predictive Science Academic Alliance Program, award DE-NA0004267 (PSAAP-IV). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Energy.

Engineering the Distributed Rising Bubble Problem in Beatnik

Jason Stewart and Patrick Bridges | Computer Science

The new Beatnik: closed surface; adaptive mesh refinement; more advanced mathematics

- Simulates a closed 3D surface – a rising bubble.
- Adaptive mesh refinement
- Far-field solve
- **Why the change?**
 - Adaptive mesh refinement (AMR) required the produce roll-up at large scales – too many points at simulation start requires too small timestepping

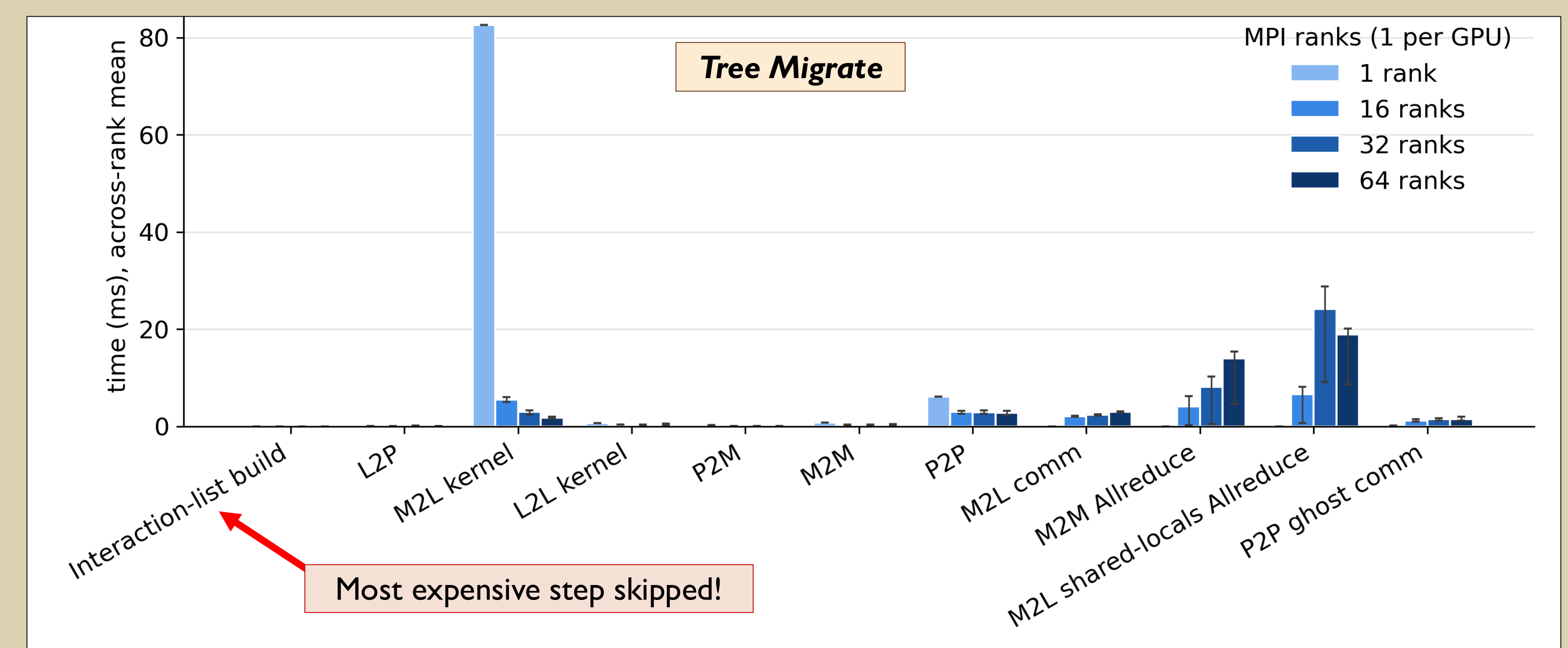
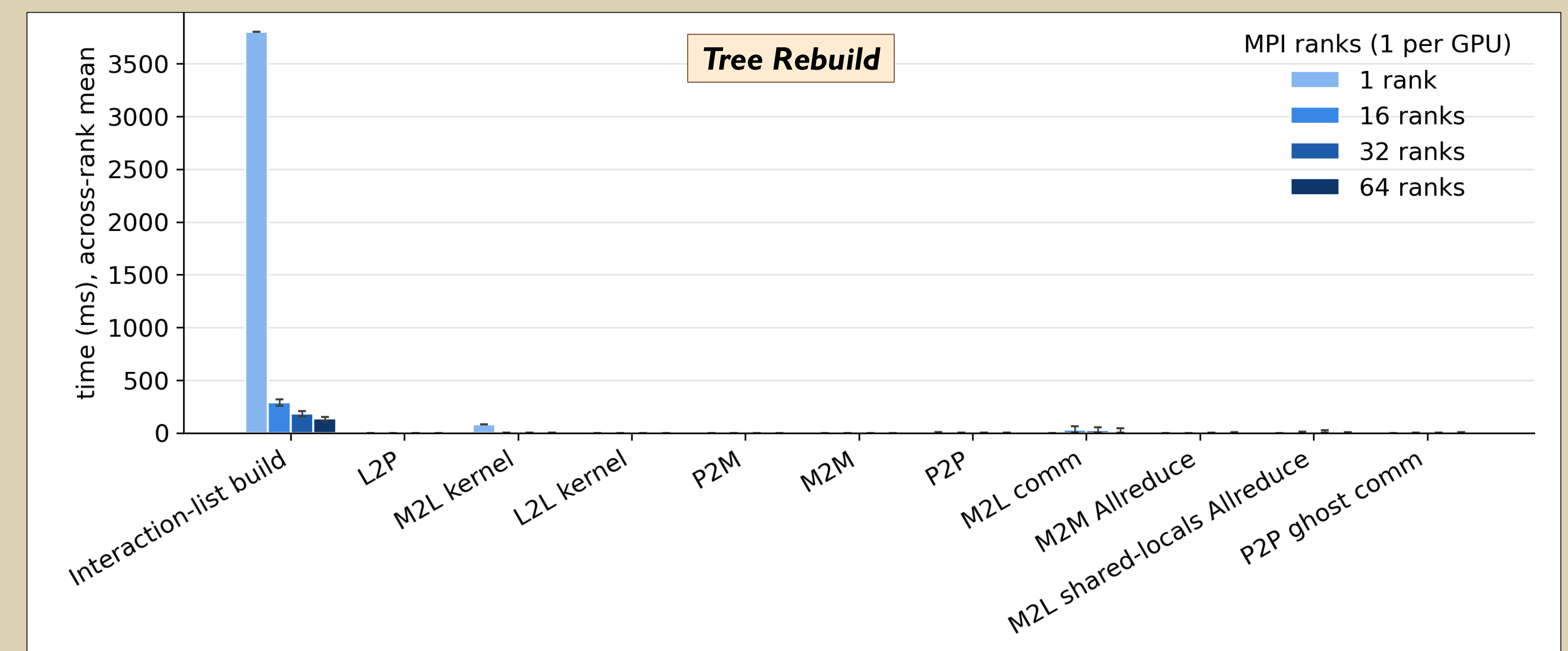


Beatnik: Initial Conditions

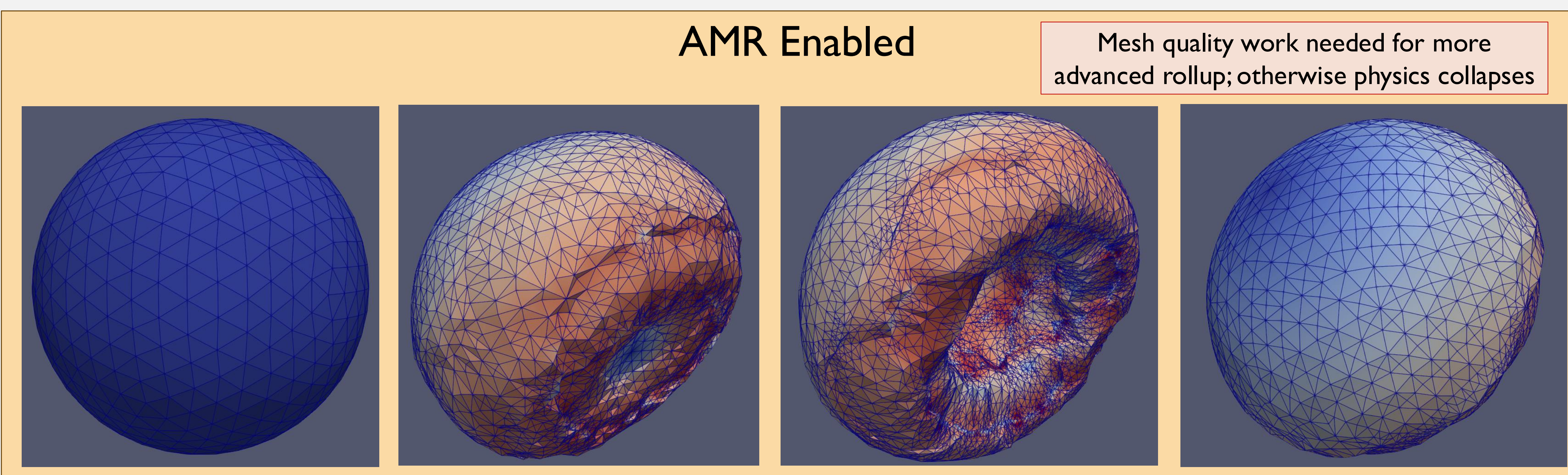
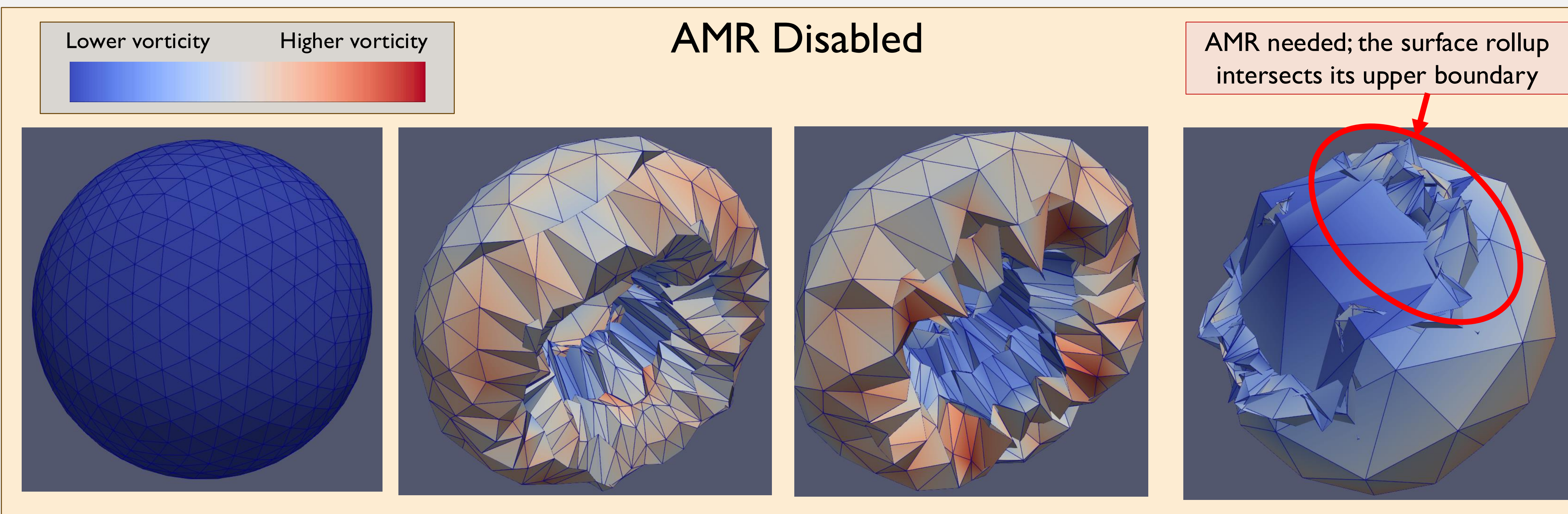
FMM optimization focuses on the cell interaction list

- Each solve particles are placed into cells; FMM math is cell-based.
- Building the cell tree is expensive.
- Tree maintenance options:
 1. **Migrate**: Move particles between cells.
 2. **Rebalance**: Coarsen/refine sections of the tree from number of particles in cells.
 3. **Rebuild**: Full tree rebuild from scratch.

FMM time breakdown: 500,000 particles, strong scaled, gravity solve (weak scaling has similar patterns)



Progression of rising bubble rollup (4 GPUs on Tuolumne)



Next steps: Debug adaptive mesh refinement (AMR); FMM profiling at large scale

- Complete Canopy profiling and optimizations at small scale (4 GPUs), focusing on algorithmic optimizations.
- Tune AMR so it can work at large scale (isotropic cleanup, min dt, remeshing constants – get enough particles to run at scale).
Cannot start with large numbers of particles or else dt is too small to get to a rollup – particles must be generated through AMR.
- Profiling and optimization of AMR.

Acknowledgements

This work was performed with support from the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP IV) Award DE-NA-0004267

Improving The Scalability of GPU-Aware Iterative Solvers Using Locality-Aware Collectives and All Available Processing Per Node

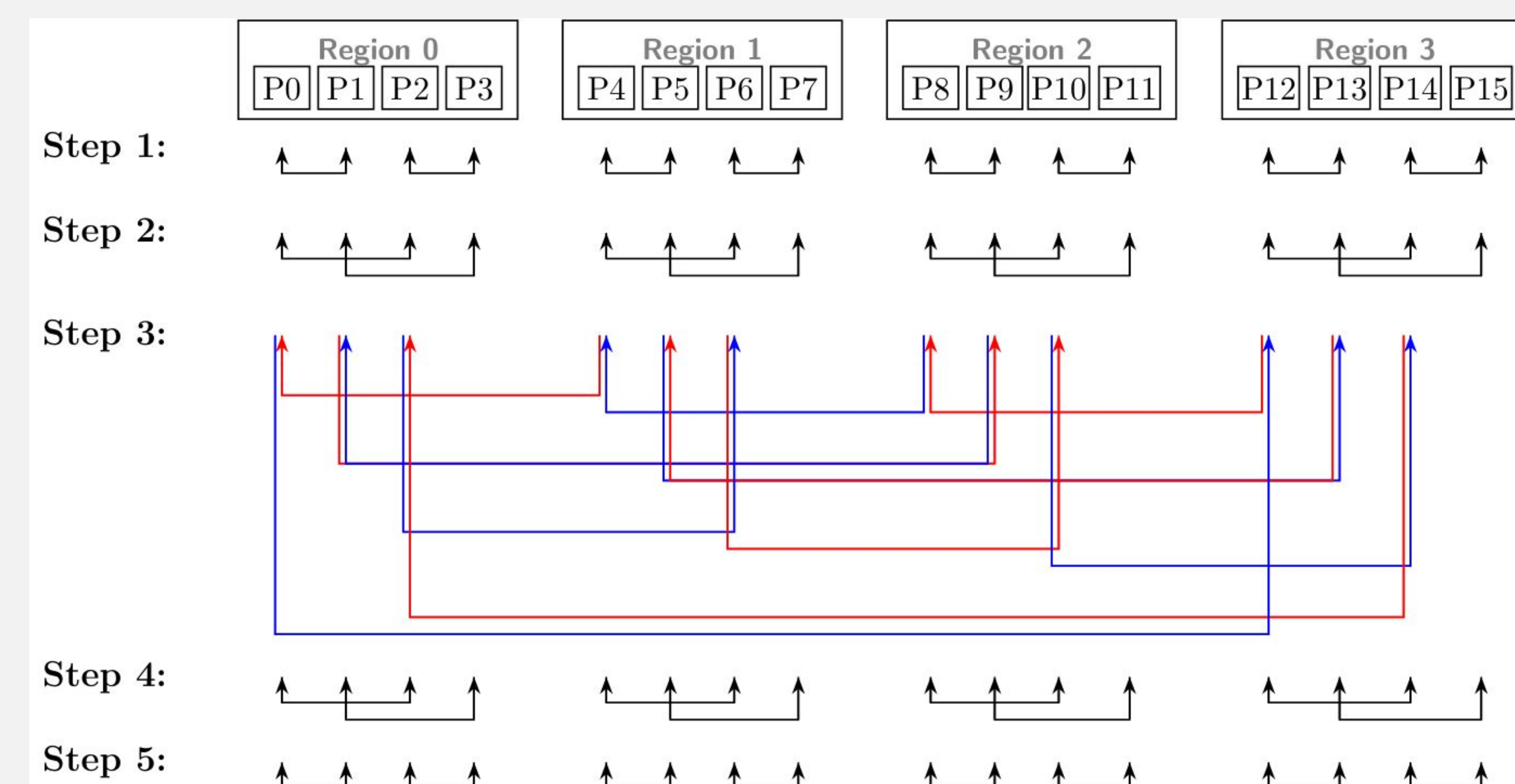
Michael Adams and Amanda Benz | Department of Computer Science

Introduction

As linear systems are strongly scaled across increasingly large process counts, message counts in all-reduce operations used in Krylov solvers, such as Conjugate Gradient (CG), grow proportionally, increasing the impact of the collective operations at scale. On modern heterogeneous systems, scaling to larger process counts reduces local, on-device computations, leading to performance drops. Only one MPI rank controls each GPU during a CG solve, so most CPU cores remain idle.

We show that CG and Algebraic Multigrid (AMG) have better scalability when all CPU cores are used instead of GPUs in the solve and setup phases within HYPRE on Tuolumne, respectively. Further, we propose an extension of the node-aware all-reduce (*Bienz, et al.) to the Non-Uniform Memory Access (NUMA) regions of Tuolumne and apply it to HYPRE's CG.

Methods



On modern, heterogeneous systems, communication latency between NUMA regions is a significant bottleneck for small collectives. We extend the node-aware all-reduce to be NUMA-aware. For n NUMA regions and ppn processes per region, there are $\log_{ppn}(n)$ inter-NUMA communication rounds where each process per NUMA region receives from a corresponding process in another NUMA region with a positive stride and then performs an on-NUMA all-reduce. While there are up to 3x the number of rounds compared to node-awareness, it shows benefits on Tuolumne.

Results

To the right, the plots show scaling studies for a HYPRE Conjugate Gradient solve of a 7 million row 5-point (top) and 27-point (bottom) Laplacian stencil on Tuolumne.

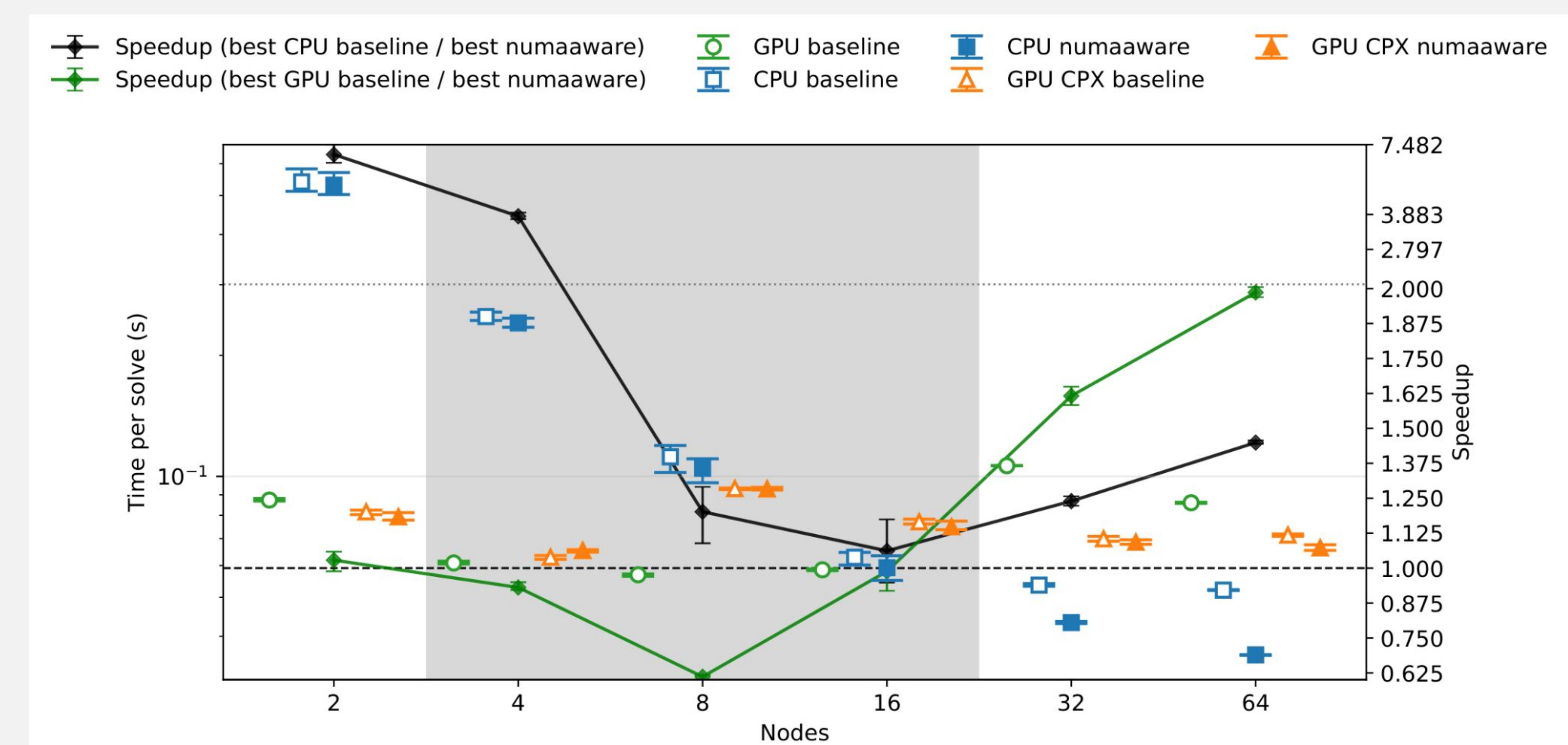
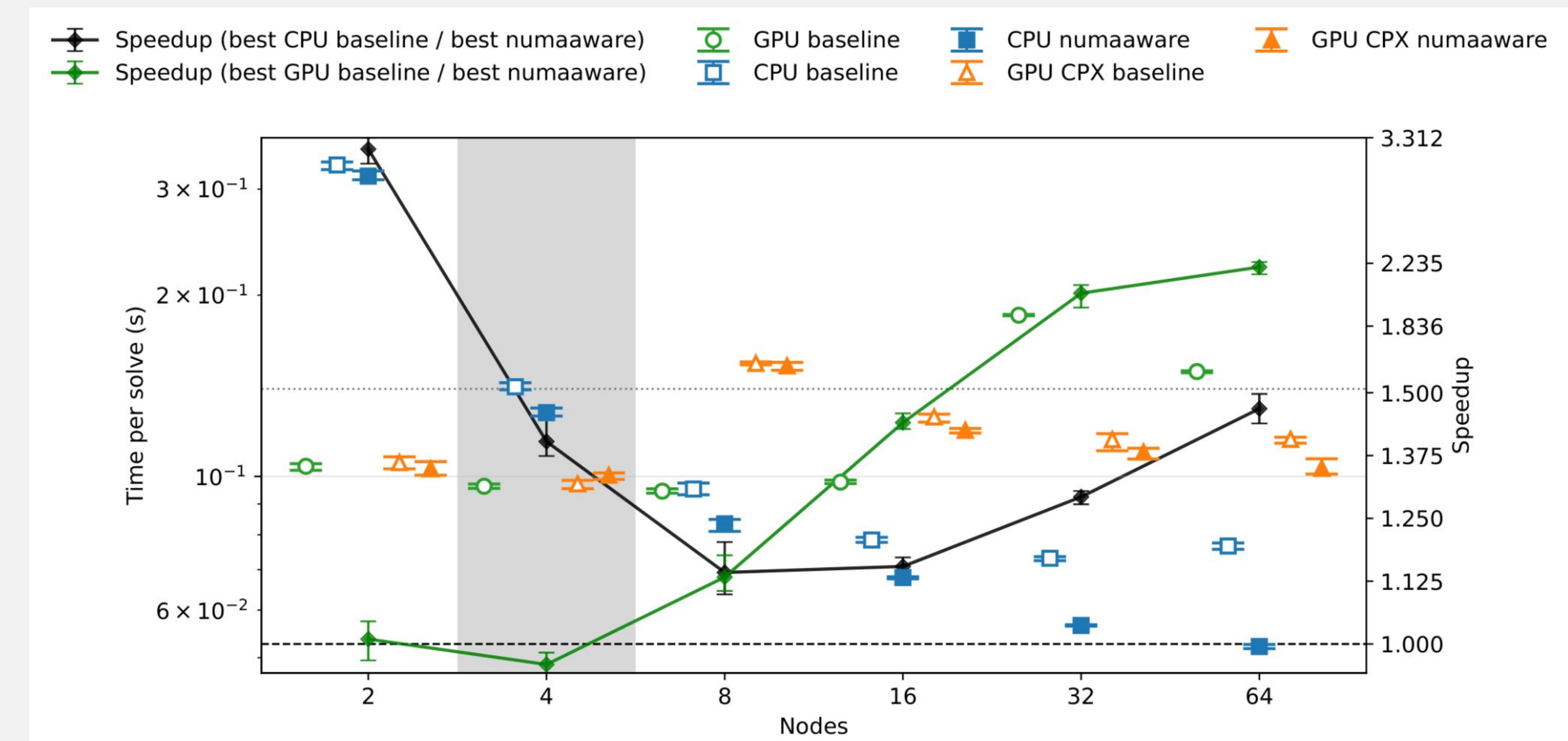
For the 5-point and 27-point stencils, GPU performance is best using a few nodes with up to **3.06x** and **6.69x** speedup over all CPUs at 2 nodes, respectively.

However, at scale, CPUs should be used for the solve. At 64 nodes, all CPUs yield up to **2.2x** and **1.98x** speedup over GPUs, respectively, with our NUMA-aware all-reduce contributing **1.46x** and **1.44x** of the speedup, respectively.

Preliminary results for AMG within HYPRE on Tuolumne yield better performance for the setup phase using all CPUs instead of GPUs past a single node, with a speedup of **2.28x** at 8 nodes.

For the solve phase, 4 GPUs per node yields the best performance with speedups of **3.27x** over all CPUs.

Using more than 4 logical GPUs per node in SPX mode by using TPX/CPX modes with 12 and 24 GPUs per node may provide benefit at scales beyond 8 nodes, shown by the decreasing gap in times as the number of nodes increases.



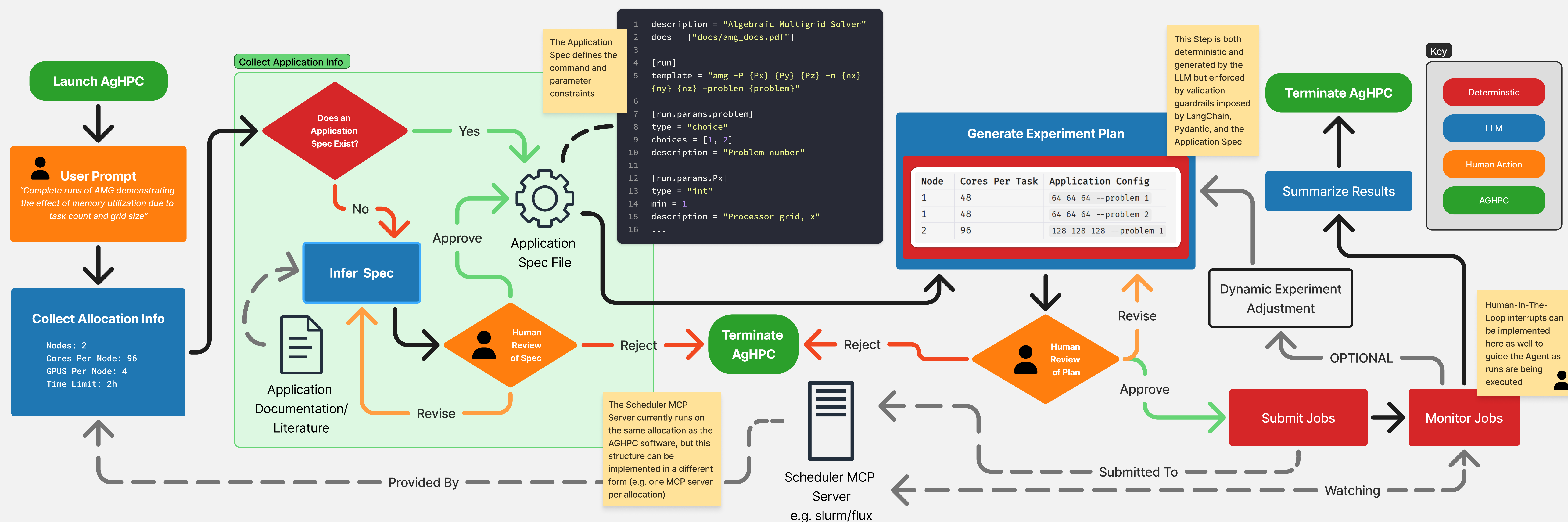
Setup	1 node	2 nodes	4 nodes	8 nodes
CPU 384x384x384	4.054044	2.025776	1.012933	0.580561
GPU 384x384x384	7.873818	4.078785	2.204089	1.327434
TPX 384x384x384	3.689112	2.095360	1.282894	0.872293
CPX 384x384x384	3.551568	2.400504	1.488354	1.209689

Solve	1 node	2 nodes	4 nodes	8 nodes
CPU 384x384x384	4.566064	2.254394	1.136380	0.581297
GPU 384x384x384	0.454276	0.299286	0.220080	0.177419
TPX 384x384x384	1.140486	0.749533	0.424559	0.207714
CPX 384x384x384	3.369955	3.077034	0.322240	0.281456

AgHPC: A Safe Agentic Framework for HPC Experiments

Nicholas Livingstone, Trilce Estrada | Computer Science Department

How can autonomous LLM agents run HPC experiments without breaking loose from their intended scope?



AgHPC workflow: from a user prompt and allocation to an approved experiment plan, submitted and watched through the Scheduler MCP server.

Introduction

Modern agentic systems now drive fast, autonomous research workflows [1], [2], with a clear use case in running HPC applications and experiments [3], [4]. However, recent incidents show the peril of unconstrained agents. Cloudflare saw severe scope drift under prompt-level boundaries [5], OpenAI and Anthropic documented agents breaking out of sandboxes and exploiting host systems [6], [7], and EchoLeak showed how untrusted text can hijack agent tool calls [8]. AgHPC addresses these failure modes with human-readable Application Specs, strictly typed MCP tools, and explicit human approval before any cluster allocation.

Design Principles

- Effective Agent Utilization**
Specs and plans are grounded in app docs, literature, and live allocation info.
- Agentic Guardrails**
Agents act only through typed MCP tools bounded by the *Application Spec*.
- Human-In-The-Loop**
Users approve, deny, or refine every agent action and can edit the spec directly.

References

- [1] J. Lála, O. O'Donoghue, A. Shtedritski, S. Cox, S. G. Rodrigues, and A. D. White, "PaperQA: Retrieval-Augmented Generative Agent for Scientific Research," no. arXiv:2312.07559. arXiv, Dec. 2023. doi: 10.48550/arXiv.2312.07559.
- [2] M. D. Skarlinski *et al.*, "Language Agents Achieve Superhuman Synthesis of Scientific Knowledge," no. arXiv:2409.13740. arXiv, Sep. 2024. doi: 10.48550/arXiv.2409.13740.
- [3] A. Nobile, A. Adelmann, and M. Sadr, "Retrieval-Augmented Generation for Scientific Code Understanding," no. arXiv:2609.12190. arXiv, Sep. 2026. doi: 10.48550/arXiv.2609.12190.
- [4] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, "DocPrompting: Generating Code by Retrieving the Docs," no. arXiv:2207.05987. arXiv, Feb. 2023. doi: 10.48550/arXiv.2207.05987.
- [5] D. Tran, J. Castro, J. Crisp, and B. Darché, "Building an Evidence-Grounded Agentic Security Operations Harness on Cloudflare," Oct. 2026.
- [6] OpenAI, "OpenAI – Hugging Face Incident," Technical Report, Aug. 2026.
- [7] Anthropic, "Investigating Unintended Model Actions in Our Evaluations and Internal Use," technical report, Oct. 2026.
- [8] P. Reddy and A. S. Gujral, "EchoLeak: The First Real-World Zero-Click Prompt Injection Exploit in a Production LLM System," no. arXiv:2509.10540. arXiv, Sep. 2025. doi: 10.48550/arXiv.2509.10540.



Do AMD and NVIDIA Count the Same Thing?

Mapping and Validating AMD MI300A and NVIDIA H100 Hardware Counters

Amr Akmal Abouelmagd¹, Michael McKinsey², David Boehme², Stephanie Brink², Olga Pearce^{2,3}

¹Tennessee Technological University | ²Lawrence Livermore National Laboratory | ³Texas A&M University

Background & Motivation

- HPC systems now mix AMD and NVIDIA GPUs. Portability layers (RAJA, Kokkos) port the code, not its performance: the same RAJAPerf [2] kernels run at different speeds (Fig. 1).
- Explaining why needs hardware counters, but the vendors' counters differ in names, granularity, and semantics, and no mapping between them exists.
- Goal:** vendor-agnostic metrics (FP64 FLOPs, memory traffic, cache behavior) that mean the same thing on both GPUs.

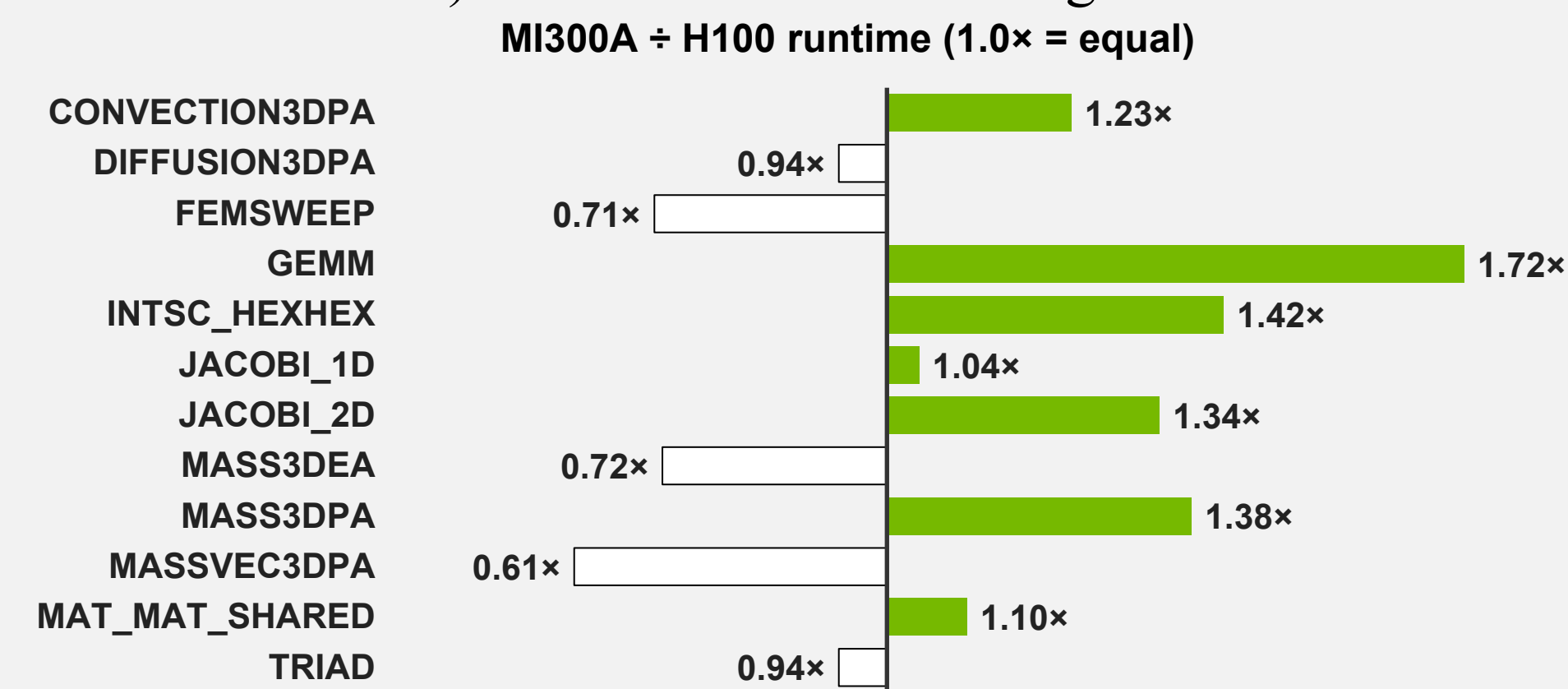


Fig. 1. MI300A ÷ H100 runtime per kernel (from [1], Fig. 1). Green: H100 faster; red: MI300A faster.

Platforms & Challenges

	AMD MI300A <i>El Capitan (SPX mode)</i>	NVIDIA H100 <i>Matrix</i>
Peak FP64 · BW	61.3 TFLOP/s · 5.3 TB/s	33.5 TFLOP/s · 3.35 TB/s
Profiler	rocprowler-sdk (ROCm 7.2.0)	Nsight Compute (ncu 26.2)
Counter names	SQ_*, TCP_*, TCC_*	smsp_*, lts_*, ...
Thread group	64-thread wavefront	32-thread warp
Cache traffic unit	128-B line request	32-B sector
L1 data cache	32 KB × 228 CUs	256 KB × 132 SMs
Collection	few counters per pass	kernel replay per group

Workload: RAJAPerf 2025.12.1, 12 kernels, 4 GB working set. Shaded rows require explicit corrections.

Methods: Workflow

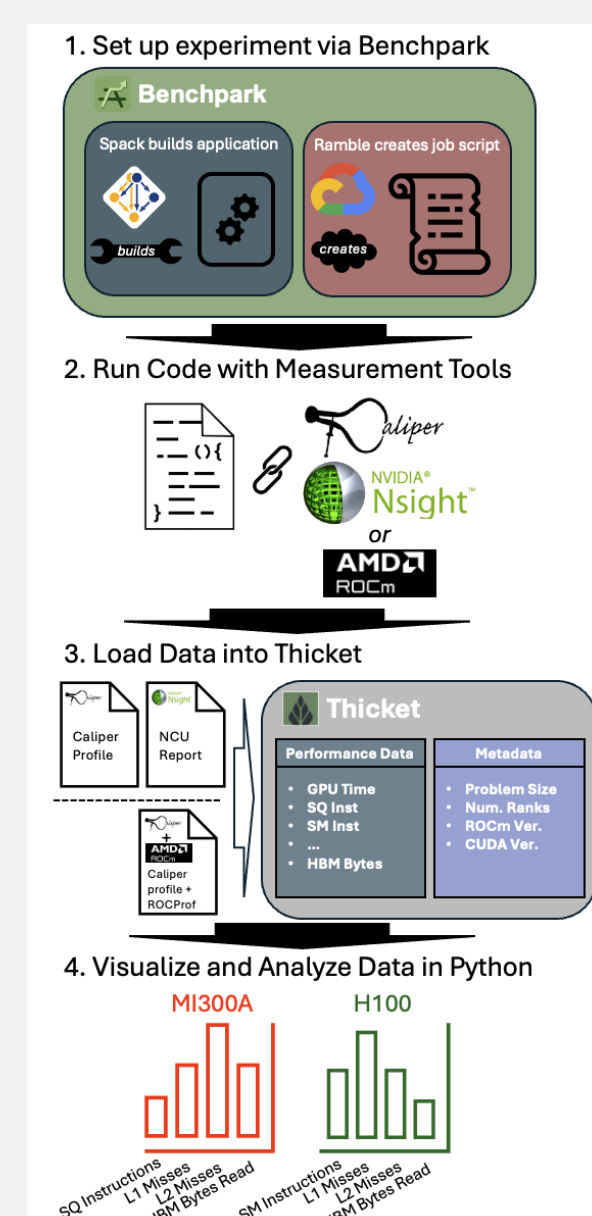


Fig. 2. Reproducible cross-platform profiling workflow [1].

Counter Mapping

Each AMD↔NVIDIA counter pair is scaled to a common unit:

Quantity	AMD counts	NVIDIA counts	NVIDIA scale
Instructions	per 64-thread wavefront	per 32-thread warp	÷ 2
Thread-level ops	per wavefront	per thread	÷ 64
FP64 FLOPs	instr. × 64 (FMA × 2)	active-thread ops	none
L1/L2 reads	128-B line requests	32-B sectors	÷ 4
L1 writes	64-B half-line requests	32-B sectors	÷ 2
HBM traffic	bytes	bytes	÷ 1

Example (FP64 MUL): SQ_INSTS_VALU_MUL_F64 ↔
smsp_sass_thread_inst_executed_op_dmuls_pred_on.sum ÷ 64

LLM-assisted counter discovery

- Claude [7] proposed candidate counters from vendor documentation; each was verified against that documentation before acceptance.
- Propose → verify → correct: a proposed 64-B L1 line (CDNA 1) was corrected to CDNA 3's 128 B [6].

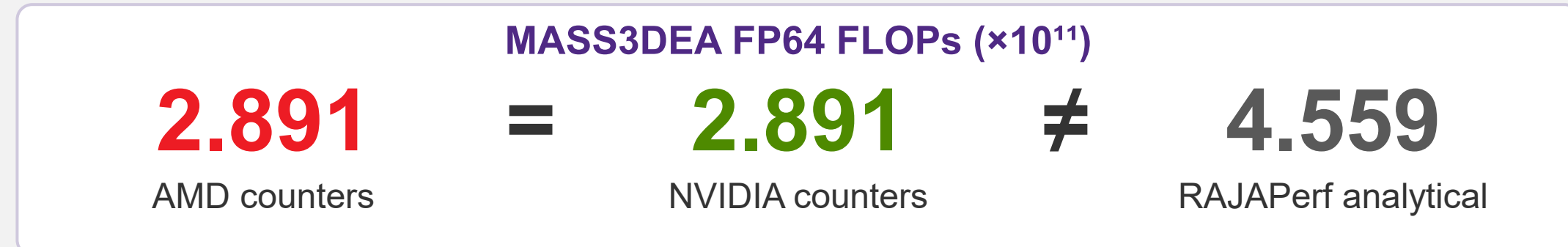
Comparability Score

$$score(a, n) = \min(a / n, n / a)$$

- a = AMD value, n = NVIDIA value; 1.0 = perfect match, 0 = only one platform reports activity.
- Symmetric, so neither vendor is ground truth; kernels where both report zero are excluded.

Validation

Simple kernels (e.g., TRIAD, JACOBI_1D) must agree, and counter-derived FLOPs are cross-checked against RAJAPerf's analytical values.



Results: Compute Counters

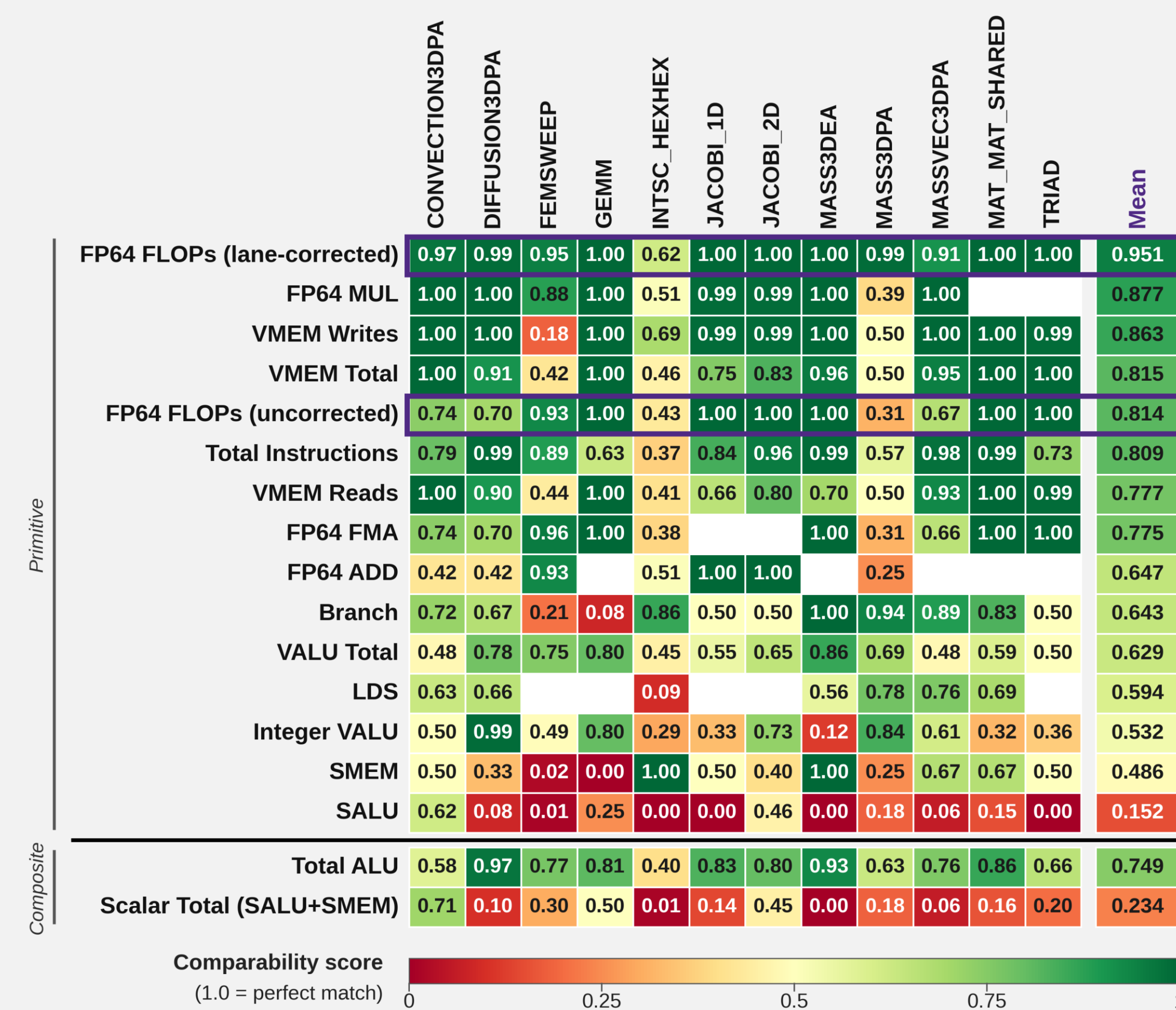


Fig. 3. Compute mappings on 12 kernels (blank = both zero; means from [1]).

- VMEM reads/writes exceed 0.9 on most kernels, so ÷2 works; MASS3DPA's 25-thread blocks fill neither unit, so ÷2 wrongly halves.
- SALU (0.152) and LDS (0.594) count different work: AMD's scalar unit runs all control flow; its LDS count includes FLAT instructions.

FP64 FLOPs: Lane Correction

- Bimodal:** 7 kernels score > 0.93, but on 5, AMD reports 1.36–3.19× more FP64 FLOPs.
- Cause:** AMD counts per wavefront, so ×64 assumes all lanes are active; NVIDIA's _pred_on counts only active threads.

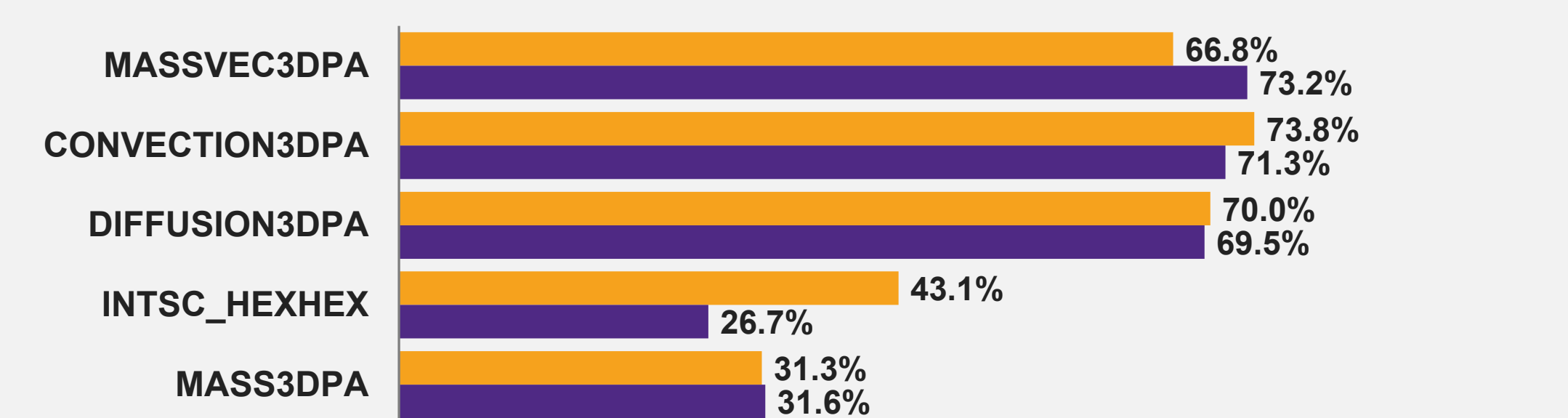


Fig. 4. NVIDIA/AMD FLOP ratio tracks AMD lane utilization [1].

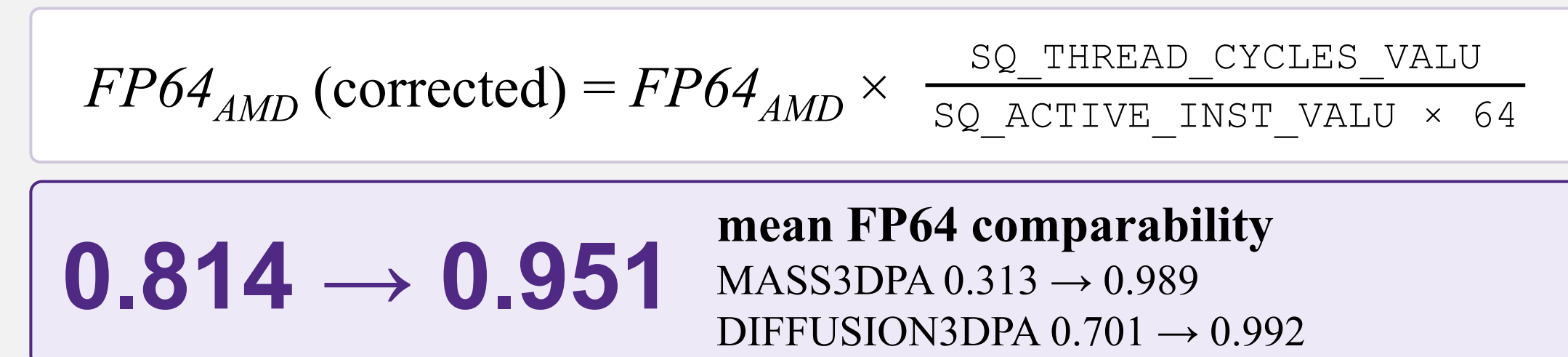


Fig. 5. Memory mappings on 12 kernels (colors as Fig. 3; means from [1]).

Results: Memory Counters

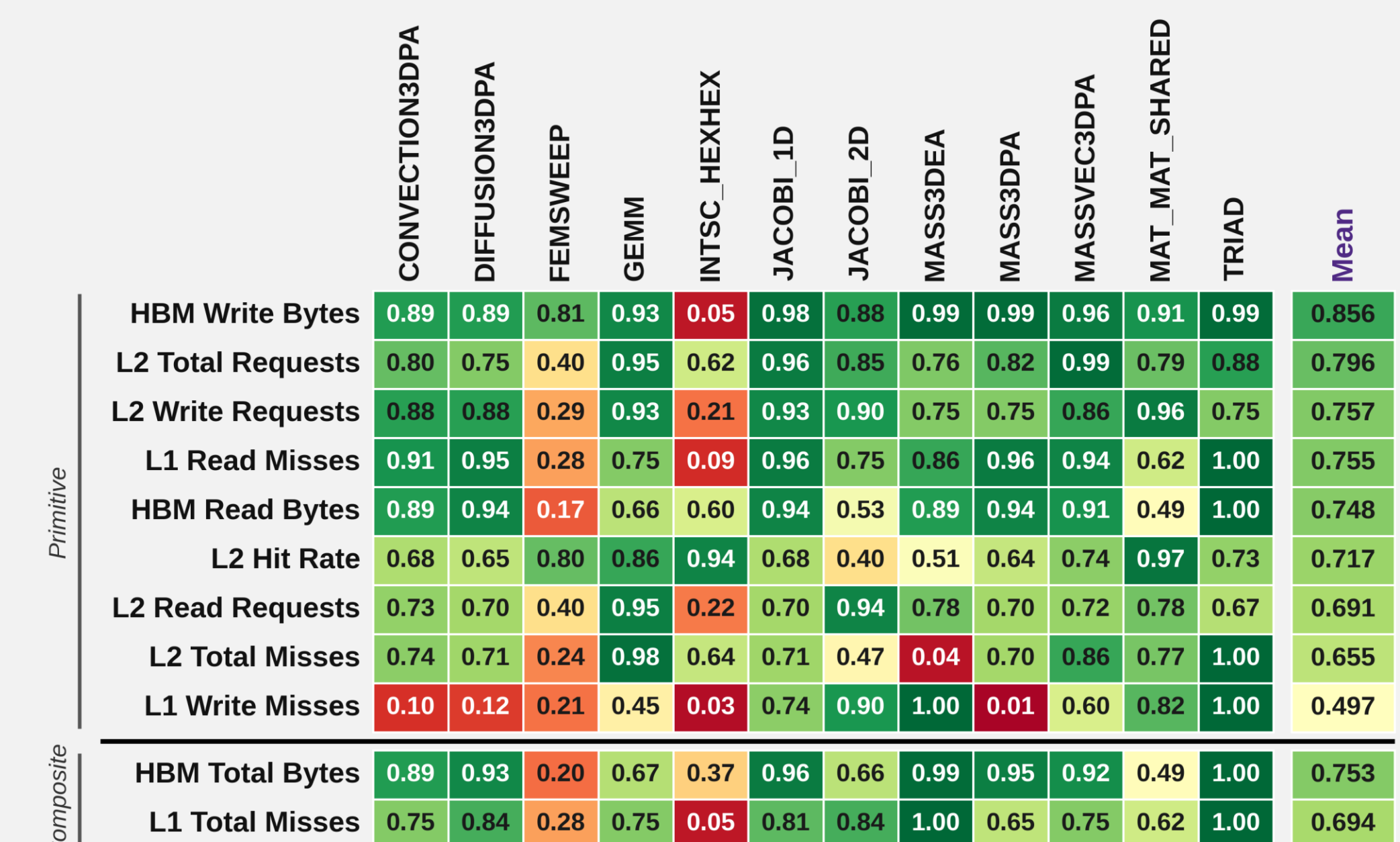


Fig. 5. Memory mappings on 12 kernels (colors as Fig. 3; means from [1]).

- HBM bytes map best** (writes 0.856): both count bytes. L2 and L1 reads follow (0.655–0.796); L1 write misses are weakest (0.497).
- L1 gaps are partly architectural:** AMD's L1 is ~5× smaller and counter scopes differ.
- INTSC_HEXHEX and FEMSWEEP** score lowest (0.348, 0.371): data-dependent control flow, indirect mesh access.

Conclusions

- Counters become comparable only after explicit thread-granularity, byte-accounting, and lane-occupancy corrections.
- Lane-utilization scaling fixes AMD's FP64 overcount (0.814 → 0.951) with two extra counters.
- Cross-vendor agreement also audits benchmarks: it exposed a RAJAPerf FLOP error.
- We report every mapping, including poor ones and gaps, so they need not be rediscovered.

References

- A. A. Abouelmagd et al., "Deriving Vendor-Agnostic GPU Metrics from AMD MI300A and NVIDIA H100 Hardware Counters," SC26 Workshops, Chicago, IL, 2026.
- O. Pearce et al., "RAJA Performance Suite: Performance Portability Analysis with Caliper and Thicket," SC24-W, 2024, pp. 1206–1218.
- D. Boehme et al., "Caliper: Performance Introspection for HPC Software Stacks," SC '16, 2016.
- S. Brink et al., "Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees," HPDC '23, 2023.
- O. Pearce et al., "Towards Collaborative Continuous Benchmarking for HPC," SC-W '23, 2023, pp. 627–635.
- AMD, "Introducing AMD CDNA 3 Architecture," white paper, 2025.
- Anthropic, "Claude" (large language model), <https://www.anthropic.com/claude>, 2024.

Acknowledgements

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-POST-2025517)



Partitioning the AMD MI300A: Locality Rules, Power Throttles

Amr Abouelmagd¹, David Boehme², Stephanie Brink², Jason Burmark², Michael McKinsey², Anthony Skjellum¹, Olga Pearce²

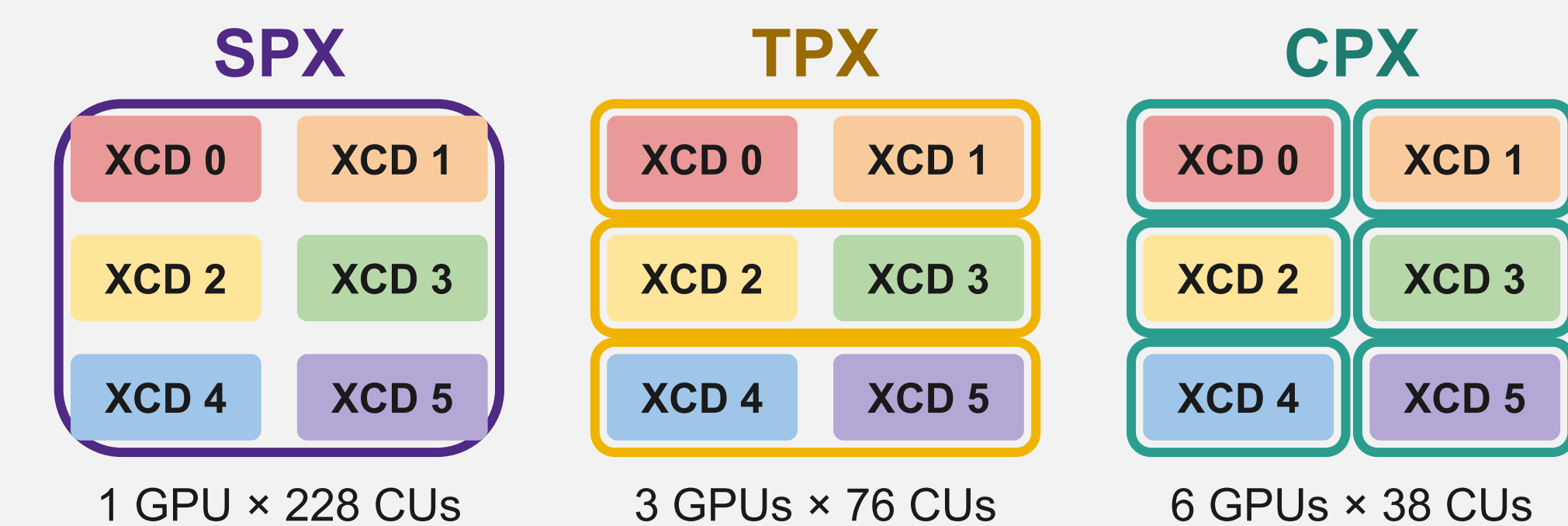
¹Tennessee Technological University, Cookeville, TN, USA

²Lawrence Livermore National Laboratory, Livermore, CA, USA

Background & Motivation

LLNL's El Capitan is built from 11,520 AMD MI300A APUs [1]. Each APU couples **6 GPU chiplets** (XCDs: 38 compute units and a private 4 MB L2 cache each) with 24 Zen 4 CPU cores and 128 GB of unified HBM [1, 3].

The GPU can run as **1, 3, or 6 logical GPUs** [2]:

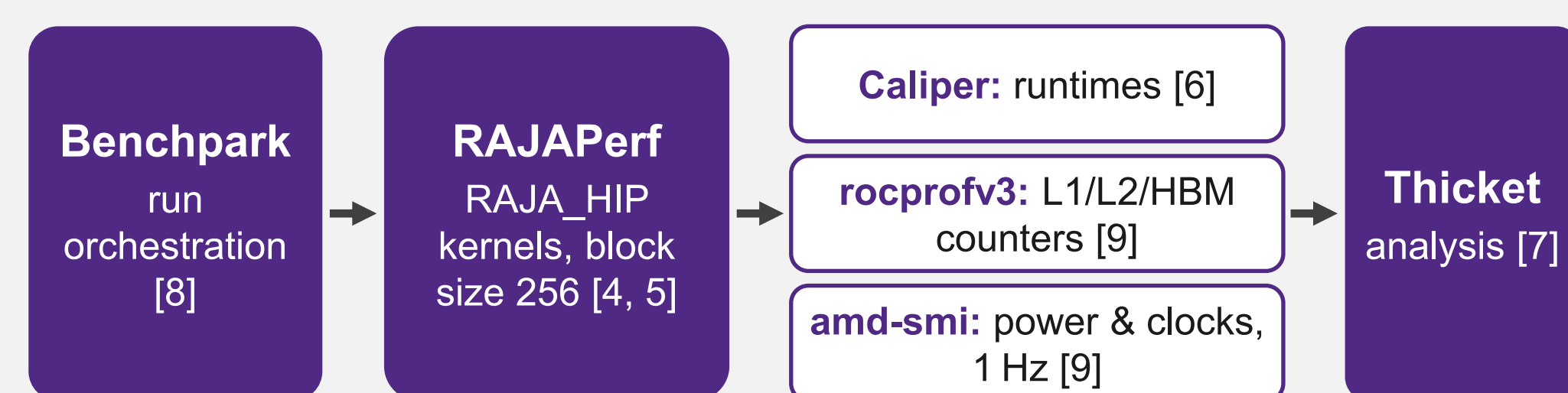


Colored outlines = logical GPUs (partitions); tiles = XCDs.

Research questions

- Does the partitioning mode change kernel runtime?
- What causes the differences?
- How does power sharing among XCDs affect runtime?

Methods



Platform: LLNL Tuolumne — 1,152 nodes × 4 MI300A APUs (512 GB HBM/node); LLVM clang 19, ROCm 6.4.2 [1].

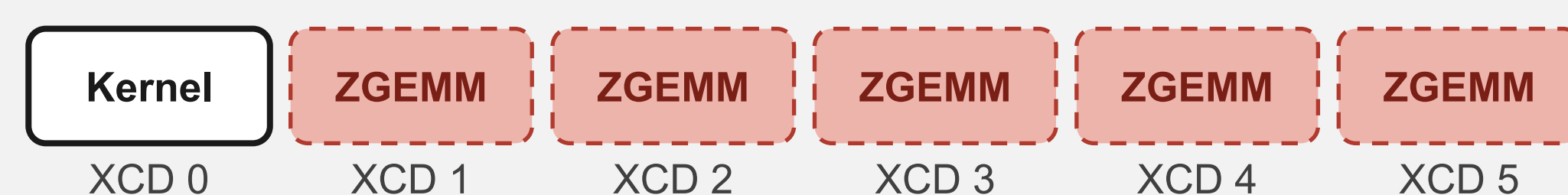
A · Partitioning modes

Same total node problem size (536,870,912); one process per logical GPU:

Mode	Processes / node	Problem size / process
SPX	4	134,217,728
TPX	12	44,739,072
CPX	24	22,369,280

B · Power sharing (CPX)

One RAJAPerf kernel on one XCD + 0–5 power-hungry ZGEMM processes on the other XCDs (ZGEMM starts 60 s earlier).

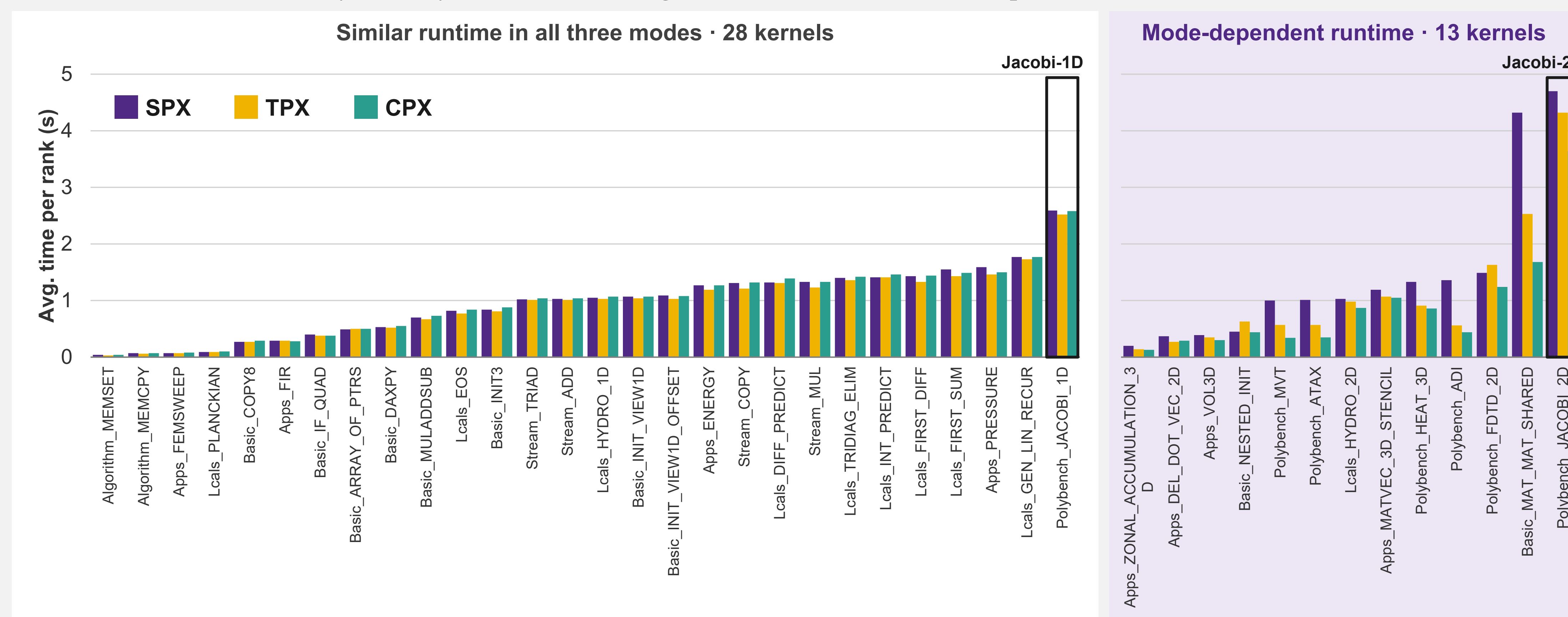


Separate runs with 0, 1, ..., 5 ZGEMM processes; amd-smi samples power, temperature & clocks at 1 Hz.

Results

1 Partitioning mode matters — for 13 of 41 kernels

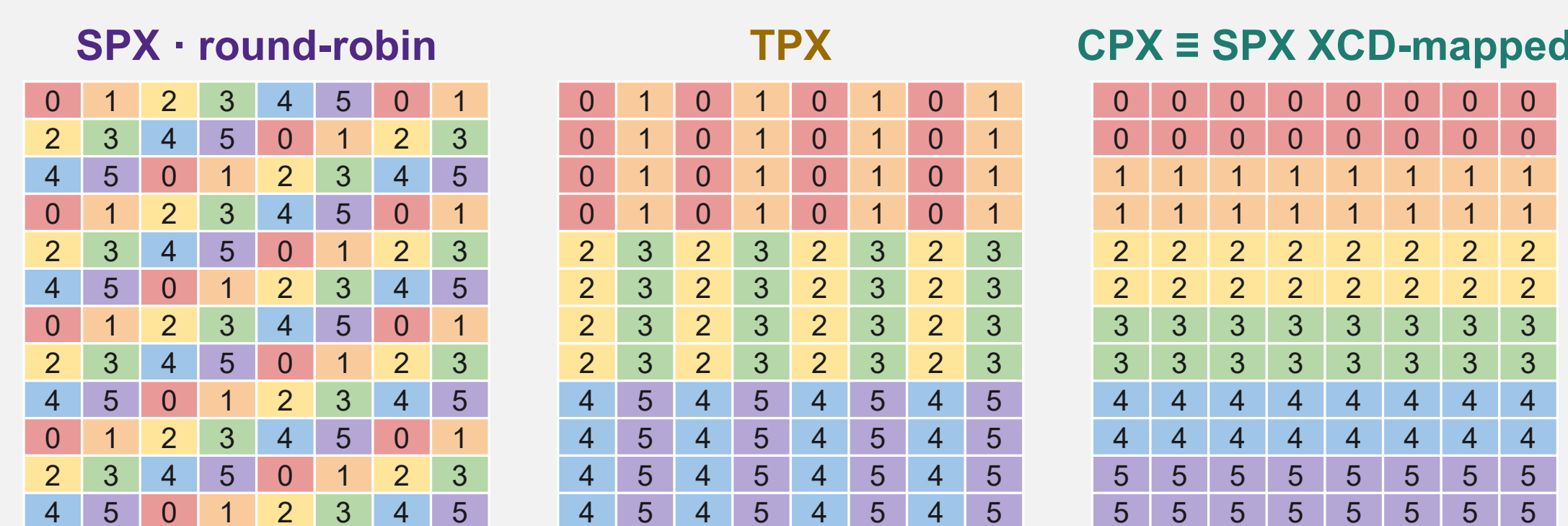
Same node workload in every mode, yet 13 kernels change runtime with the mode — up to 30% for stencils [1].



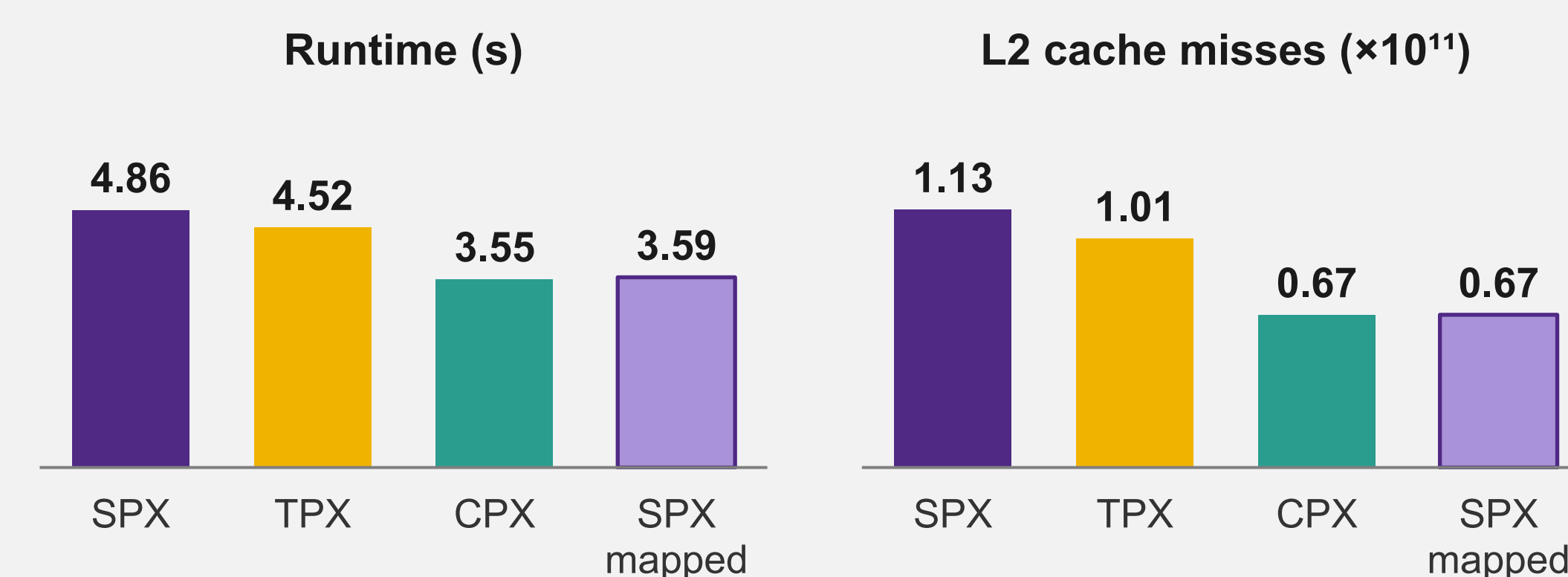
RAJA_HIP, block size 256; kernels sorted by SPX time within each group. Data from [1], Fig. 3.

2 Data locality explains the gap

- Jacobi-1D** (linear access): nearly identical counters and runtime in all modes.
- Jacobi-2D** (stencil, 3 rows per update): SPX deals thread blocks round-robin to 6 XCDs → neighbors split across private L2s.
- TPX, CPX, or **XCD-aware mapping in SPX** keep neighbors together → fewer L2 misses, faster runs.



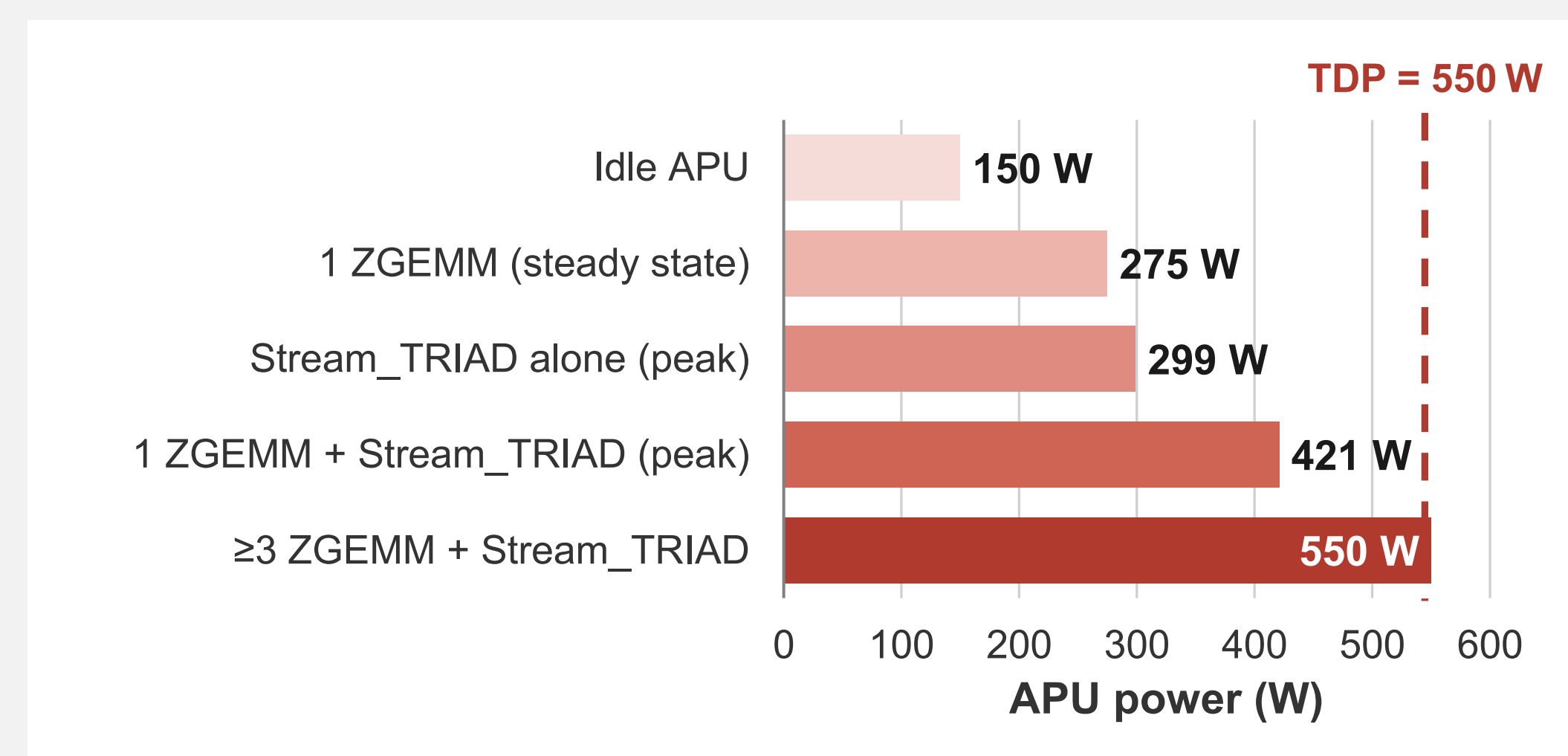
Jacobi-2D: the XCD that executes each thread block; after [1], Fig. 6.



SPX mapped = SPX with XCD-aware block mapping. Data: [1], Fig. 5.

3 Shared power budget throttles kernels

XCDs and CPU cores share one 550 W power budget (TDP). ZGEMM on ≥3 XCDs hits TDP, so GPU clocks drop [1].



APU power from amd-smi (1 Hz); values from [1], Sec. 6.

Slowdown vs. number of background ZGEMM processes [1]

- FLOP-bound** (e.g., Basic_MAT_MAT_SHARED, Apps_FIR, Apps_VOL3D): up to 5% at 1, 10% at 2, 20% at 4.
- Other kernels**: no change up to 3; ≈5% at 4, ≈20% at 5.
- Memory-bound** (e.g., Algorithm_MEMCPY, Basic_COPY8): smallest slowdowns.

Slowdowns suggest staged throttling (≈5% → 10% → 20%) [1].

Power-hungry ZGEMM can run on only **3 of 6 XCDs** at nominal clocks before the APU reaches TDP [1].

Conclusions

30% Max. runtime difference between modes (stencil kernels, same node workload)

41% Fewer L2 misses for Jacobi-2D with XCD-aware block mapping in SPX

20% Max. slowdown from power throttling; FLOP-bound kernels are hit first

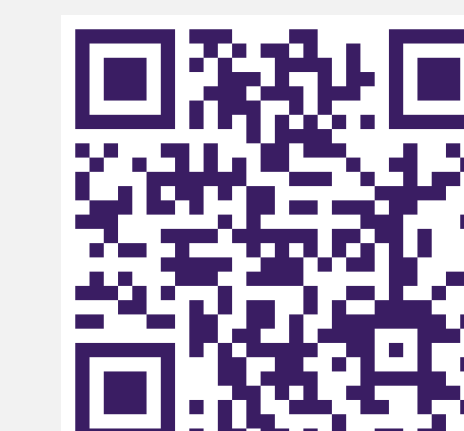
- The mode matters for some kernels; the gap comes from **data locality across XCD L2 caches**, not the mode itself.
- Keeping neighboring thread blocks on one XCD lets **SPX match CPX**.
- XCDs share one 550 W budget: co-located power-hungry work **throttles FLOP-bound kernels** first.
- Next:** workload-aware placement and scheduling on APUs.

References

- A. Abouelmagd, D. Boehme, S. Brink, J. Burmark, M. McKinsey, A. Skjellum, O. Pearce. GPU Partitioning, Power, and Performance of the AMD MI300A. *SCA/HPCAsia 2026*, ACM, pp. 215–227. doi:10.1145/3773656.3773680
- AMD. AMD Instinct MI300A APU: GPU Partitioning Overview. *instinct.docs.amd.com*
- A. Smith et al. Realizing the AMD Exascale Heterogeneous Processor Vision: Industry Product. *ISCA 2024*.
- O. Pearce et al. RAJA Performance Suite: Performance Portability Analysis with Caliper and Thicket. *SC24-W (P3HPC)*, 2024.
- D. A. Beckingsale et al. RAJA: Portable Performance for Large-Scale Scientific Applications. *P3HPC 2019*.
- D. Boehme et al. Caliper: Performance Introspection for HPC Software Stacks. *SC'16*.
- S. Brink et al. Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees. *HPDC 2023*.
- O. Pearce et al. HPC Benchmarking: Repeat, Replicate, Reproduce. *ACM REP 2025*.
- AMD. ROCprofiler v3 and AMD SMI documentation. *rocm.docs.amd.com*

Acknowledgements

Prepared by LLNL under Contract DE-AC52-07NA27344 (LLNL-POST-2025518)



Read the full paper [1]

SCA/HPCAsia 2026 (ACM)
doi.org/10.1145/3773656.3773680

Contact: Amr Akmal Abouelmagd
aabouelma42@ntech.edu

Streamlining Cabana + MPI Using CPU-free Communication

Evan Drake Suggs, Derek Schafer, and Anthony Skjellum | Tennessee Technological University

Introduction

- GPU technology is often bottlenecked by CPU-GPU communication.
- MPI Advance stream-triggering library avoids this by queueing GPU operations.**
- By integrating it with the Cabana particle simulation library and our CabanaGhost benchmark, we make a sample case for how MPI Advance's stream-triggering API can be both performant and flexible in integration with existing programming libraries.
- This work shows that that MPI Advance stream-triggering with can work well with many libraries and applications that use MPI and performance portable frameworks.

Results

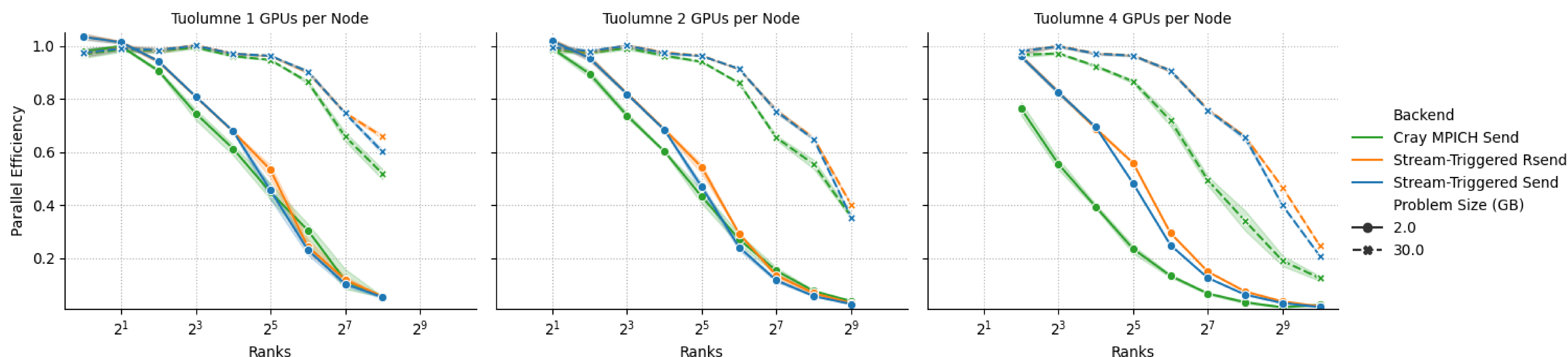
- Once completed, this backend was integrated into our CabanaGhost benchmarking tool, which features a halo exchange strong scaling test and ran on LLNL's Tuolumne and ORNL's Frontier system.
- Runs shown use the MPI Advance stream-triggering Send (code at right) and RSend, along with a Cray MPICH implementation for both 2 and 30 GB problem size.
- While most of runs performed as well or better than Cray MPICH, larger rank sizes on ≥ 4 nodes had worse performance. Variance of MPICH results is also likely due to Cray MPICH optimizations.

```
template <class... ArrayTypes>
void enqueueGather(
    const ArrayTypes&... arrays)
{
    // Enqueue starting the receives
    MPI_Enqueue_startall( _my_queue,
        num_n, &_requests[0] );
    // Enqueue packing the data to send
    enqueuePackBuffers( _owned_buffers,
        _owned_steering, arrays.view()... );

    // Enqueue sends and waiting for
    // communication to complete
    MPI_Enqueue_startall( _my_queue,
        num_n, &_requests[num_n] );
    MPI_Enqueue_waitall( _my_queue );

    // Enqueue unpacking receive buffers
    enqueueUnpackBuffers( _ghost_buffers,
        _ghost_steering, arrays.view()... );
}
```

Cabana Benchmarks



Design

- For benchmarking, our CabanaGhost benchmarking tool was integrated with a version of Cabana's Grid Halo [1] with MPI Advance stream-triggering as a backend.
- Stream-triggered versions of the collectives Scatter and Gather (enqueueScatter and enqueueGather) were created (with options for both Send and RSend), as well as a default implementation using Cray MPICH.
- These stream-triggered functions first start the receives, pack the send buffers, and after receives are completed, unpack their buffers. This avoids fences from the default Cabana implementation.

Conclusions

- This work showcases the capabilities and successful application of the MPI Advance Stream-triggering library.
- We are currently working to improve performance, and fix remaining issues, such as a limited number of resource handles.
- For more details, see QR code for our ArXiv version of our paper published at Cluster Computing 2026.

Future plans

- Publishing a paper on Kokkos Comm stream-triggering integration
- Creation of more collectives, such as AllReduce
- New applications including Trilinos integration

QR Code



QR code for full paper on arXiv

Acknowledgements

This work was performed with partial support from NSF Grants 2514054, 2450093, 2405142 and 2412182 and from the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP III & IV), Awards DE-NA0003966 and DENA0004267.

We'd like to thank Trey White (ORNL), Jack Lange (ORNL), Whit Schonbein (Sandia), Matthew Donsanji (Sandia), and Naveen Ravi (HPE) for all their insights on our interface and debugging the CXI libfabric library.

References

- [1] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandi'e, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, and S. Mniszewski, "Cabana: A Performance Portable Library for Particle-Based Simulations," Journal of Open Source Software, vol. 7, no. 72, p. 4115, Apr. 2022. [Online]. Available: <https://joss.theoj.org/papers/10.21105/joss.04115>

Optimizing MPI Parameters under Uncertainty

Xavier Barr, Professor Patrick Bridges | Department of Computer Science

Modeling Performance Variation for Sparse Solvers

aCG displays high run-to-run variation on Tuolumne, even under consistent conditions. Figure 1 shows performance variation from three different autotuning sessions of aCG. Run-to-run spread reached as high as 29.7% for the default configuration, and 39.6% for a study on intra-node communication parameters for GPUs.

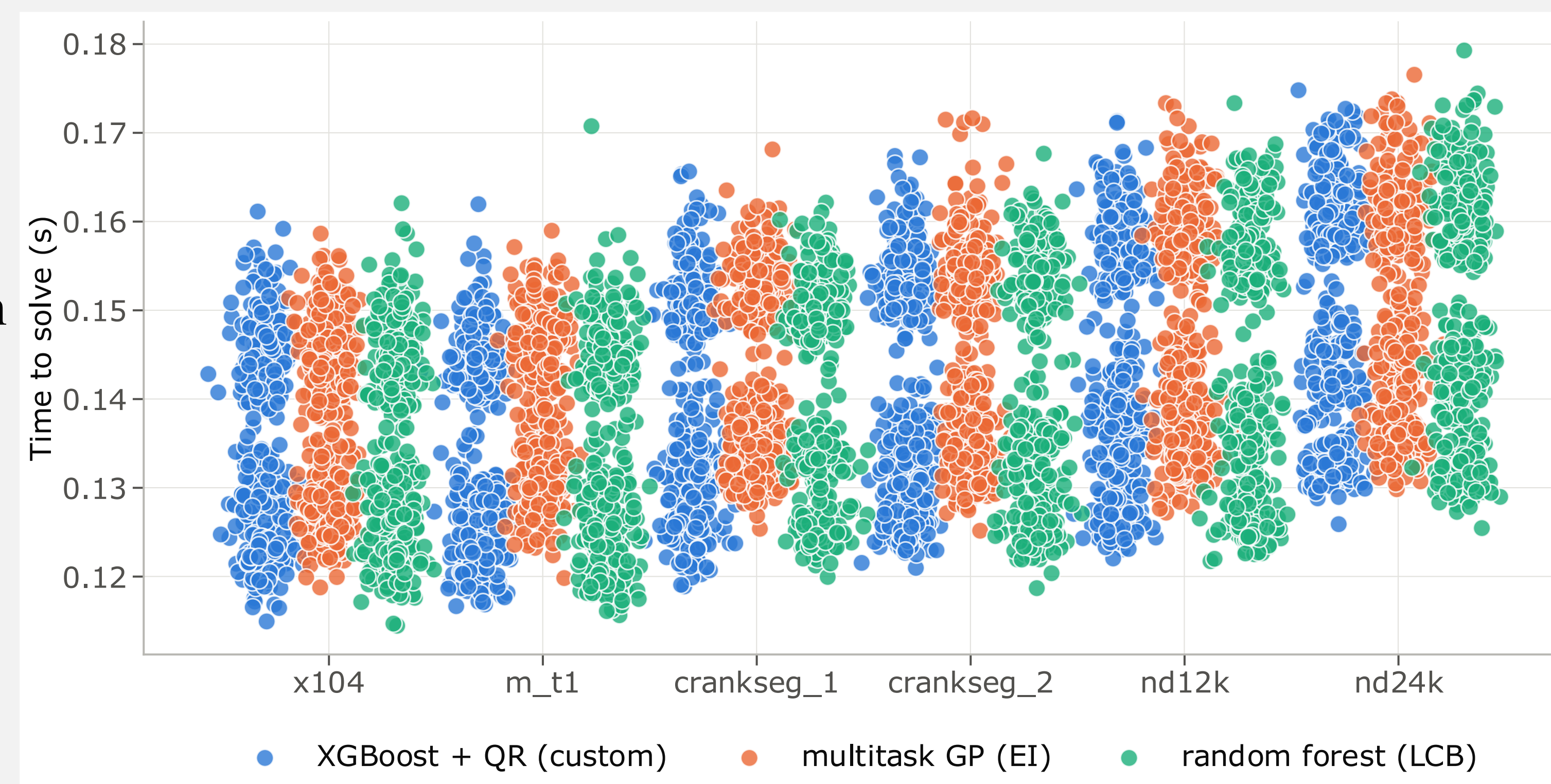


Figure 1: Performance variation of aCG on Tuolumne with different MPI configurations.

Autotuning Cray MPICH Parameters for aCG on Tuolumne

Bayesian optimization finds near-optimal configurations with significantly less data than an exhaustive search [1, 2, 4]. There is little work investigating how different uncertainty quantification techniques affect Bayesian optimization when applied to sparse solvers. We hypothesize that quantile regression can improve parameter autotuning. We use quantile regression with XGBoost and a custom acquisition function. We compare it against a Gaussian process (GP) and a random forest (RF) for tuning MPI parameters of aCG running on Tuolumne. These models are based on those used in [1, 2].

Our approach finds better configurations than the GP, likely because the GP assumes time to solve is normally distributed. The RF has similar performance to ours, but lacks clear exploration and exploitation phases. This may be caused by the LCB acquisition function or the normality assumption in the predicted median.

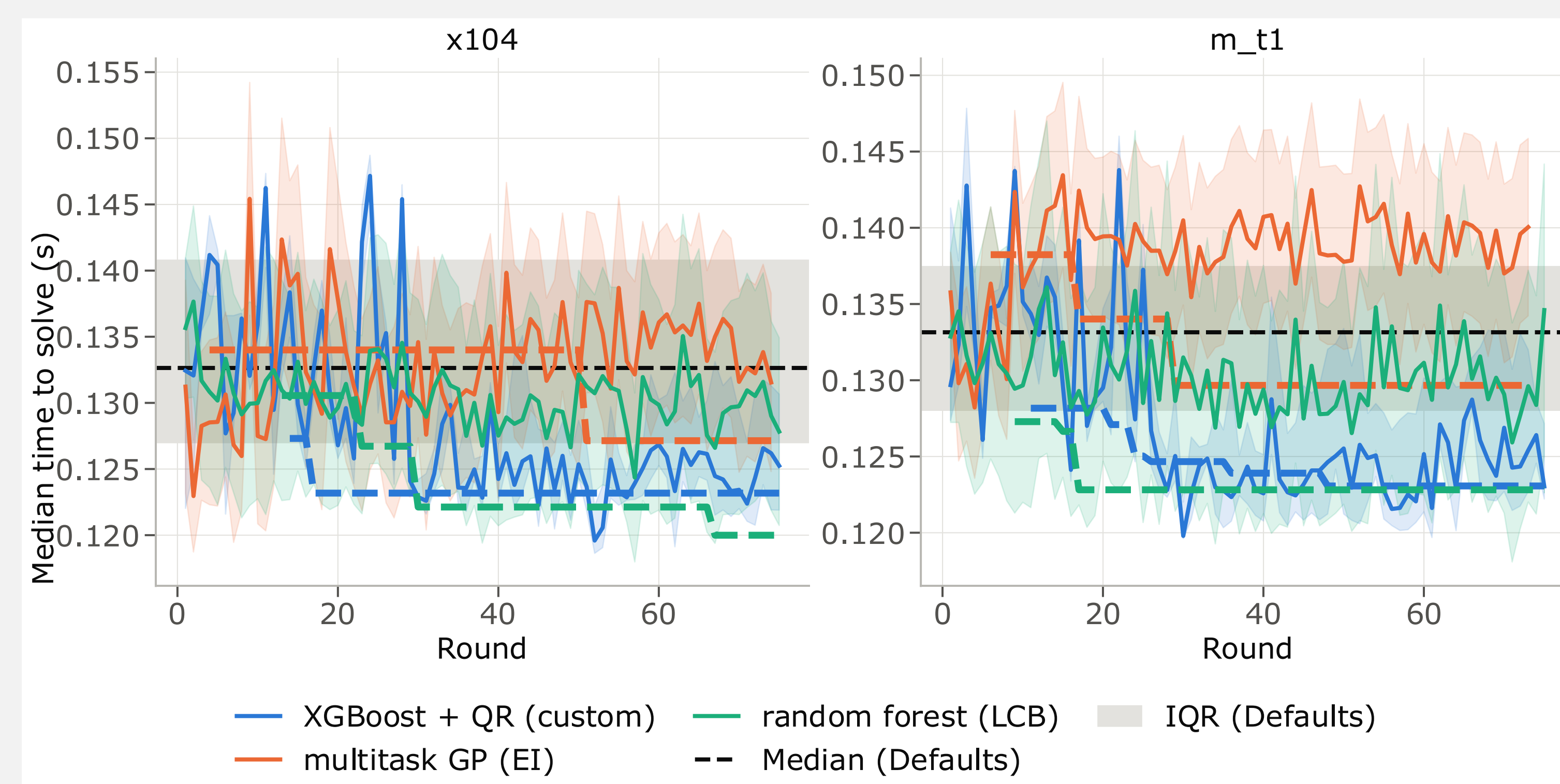


Figure 2: Three models tuning intra-node communication parameters for aCG on Tuolumne. Solid lines: predicted median of the model. Dashed lines: median time to solve of best configuration encountered if run at least 5 times. Shaded regions: models' IQR. The number of processes and nodes is fixed at 16 and 4, respectively.

Quantifying MPI Parameter Impact

We use the approach proposed in [4] to quantify each parameter's impact. This method uses fANOVA and a random forest trained on data collected from Bayesian optimization. It then computes the effect of each parameter and their interactions.

Figure 3 shows the effect of each MPI parameter. The uncertainty in each effect is high and indicates noise or parameter interactions dominate time to solve. Vendors and system administrators can use this method to identify which MPI parameters are worth tuning or exposing to users.

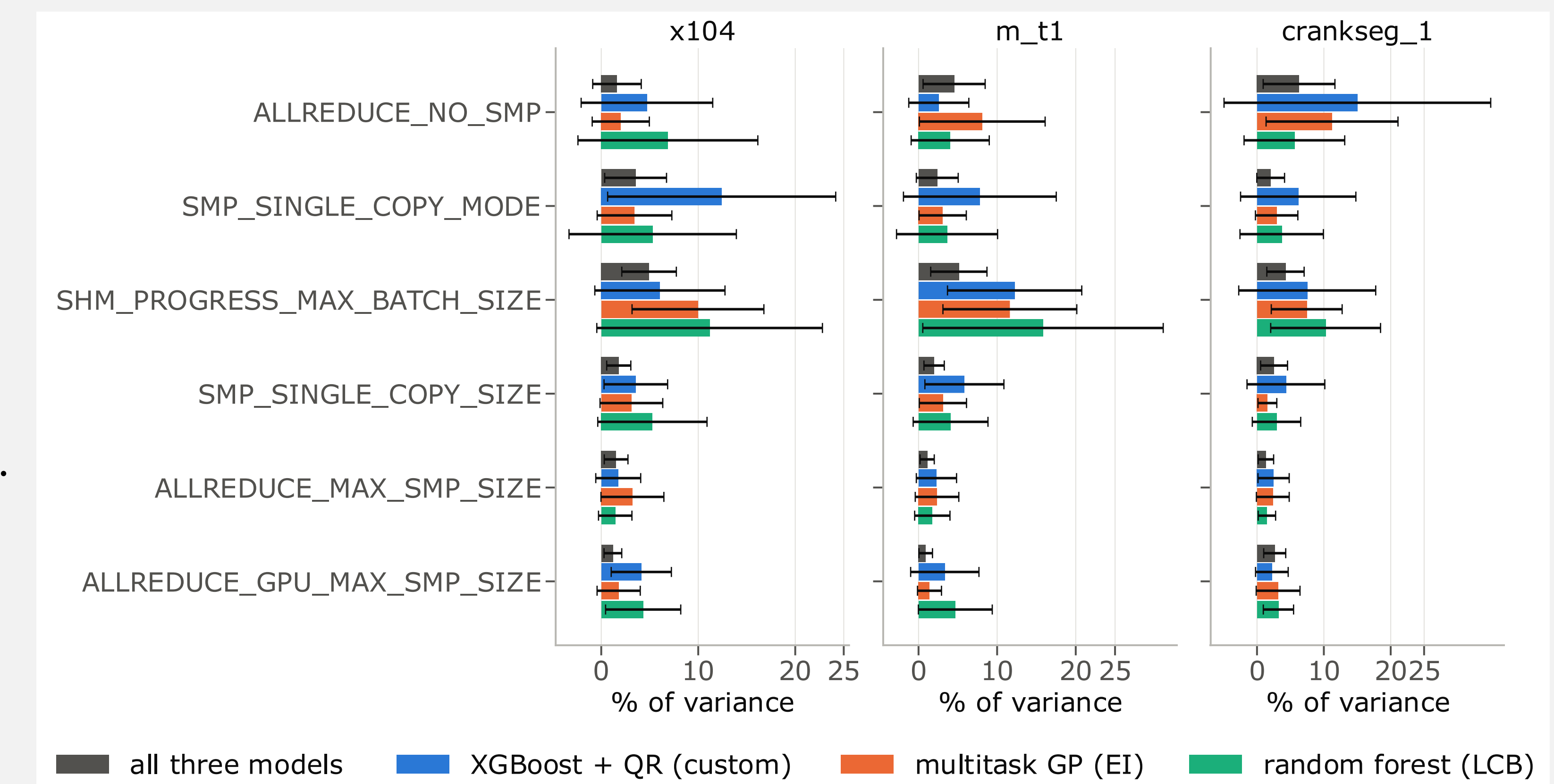


Figure 3: Estimate of each parameter's share of variation in time to solve. Variance is calculated using fANOVA on a random forest trained on the data collected while tuning. The error bars are computed as the disagreement between the trees.

General Purpose Models

Many factors outside MPI and application parameters introduce variation in sparse solvers, such as matrix structure, load imbalance, and network contention. More advanced deep learning methods could help capture and explain this high variation and predict performance for entirely different MPI applications. Recent work using AST representations to optimize OpenMP codes [3] inspires this goal. The following table lists the strategies we plan to explore.

Strategies	Purpose
GNNs on communication graphs	Account for load imbalance in sparse solvers
GNNs on MPI program representations	Single model for different applications
Combining analytical models with machine learning	Reduce data collection and improve extrapolation

References

- Y. Liu et al., "GPTune: multitask learning for autotuning exascale applications," in *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Virtual Event Republic of Korea: ACM, Feb. 2021, pp. 234–246. doi: [10.1145/3437801.3441621](https://doi.org/10.1145/3437801.3441621).
- X. Wu et al., "ytot: Autotuning Scientific Applications for Energy Efficiency at Large Scales," Mar. 28, 2023, arXiv: arXiv:2303.16245. doi: [10.48550/arXiv.2303.16245](https://doi.org/10.48550/arXiv.2303.16245).
- A. Tehranijamsaz, A. Mishra, A. Dutta, A. M. Malik, B. Chapman, and A. Jannesari, "ParaGraph: Weighted Graph Representation for Performance Optimization of HPC Kernels," in *2024 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, San Francisco, CA, USA: IEEE, May 2024, pp. 293–300. doi: [10.1109/IPDPSW63119.2024.00070](https://doi.org/10.1109/IPDPSW63119.2024.00070).
- F. Hutter, H. Hoos, and K. Leyton-Brown, "An efficient approach for assessing hyperparameter importance," in *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, in ICML'14. JMLR.org, 2014, pp. 1-754–762.

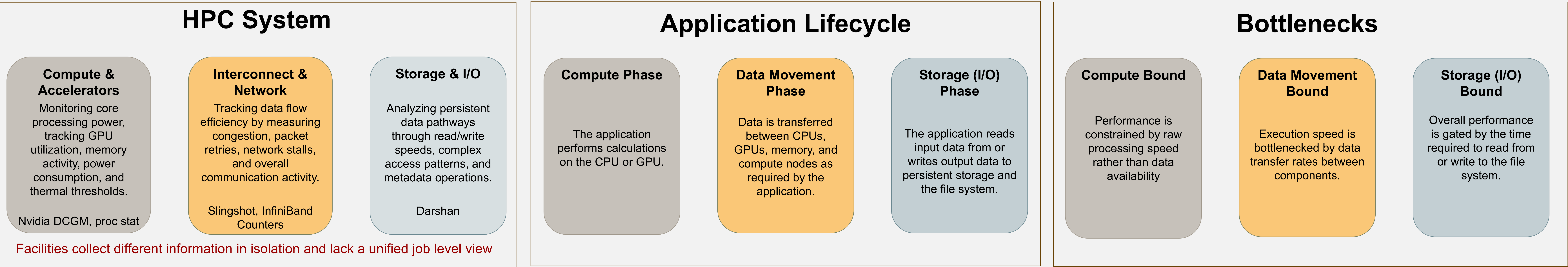
Acknowledgements

This research was supported in part by an award from the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program award DE-NA0004267.

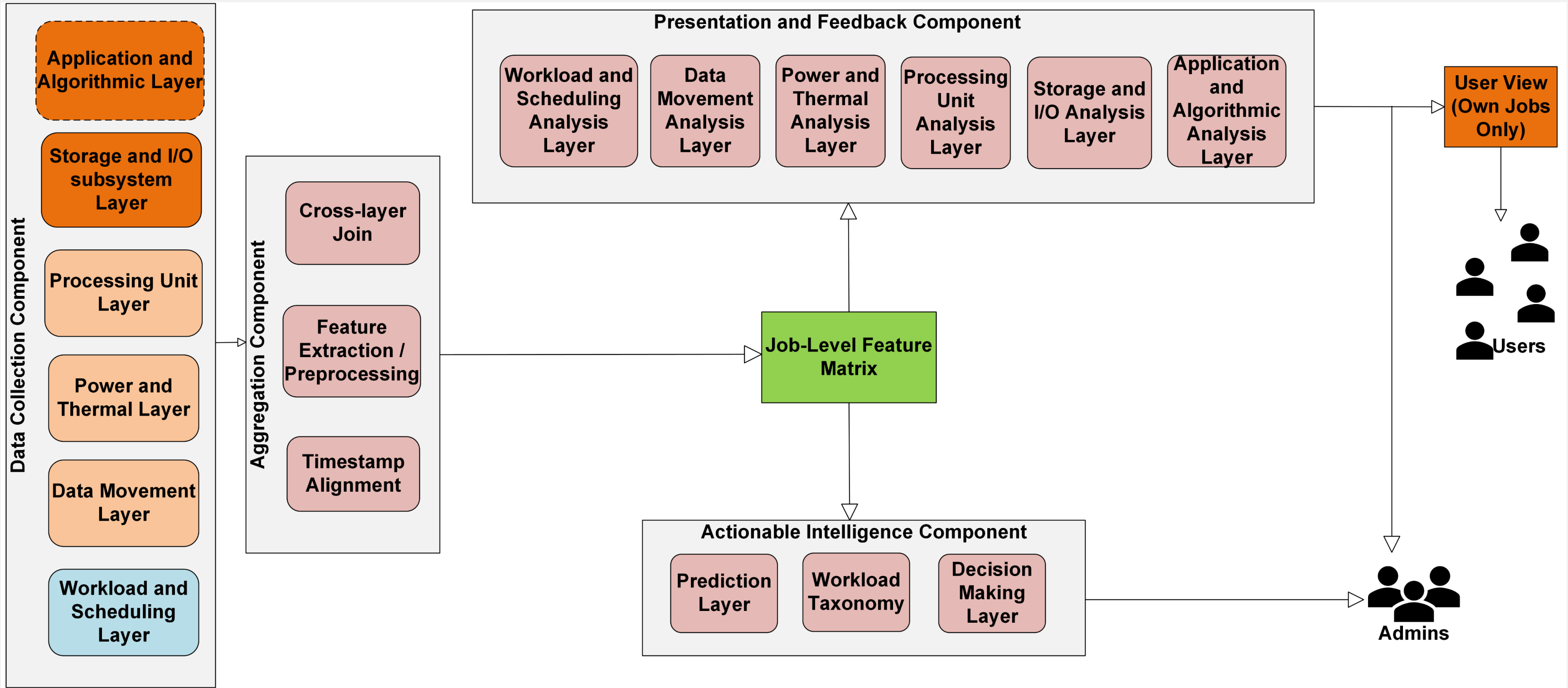


Cross-Layer Characterization Framework for HPC Applications

Hari Teja Jajula, Purushotham V. Bangalore, Department of Computer Science



Cross-Layer Characterization Framework

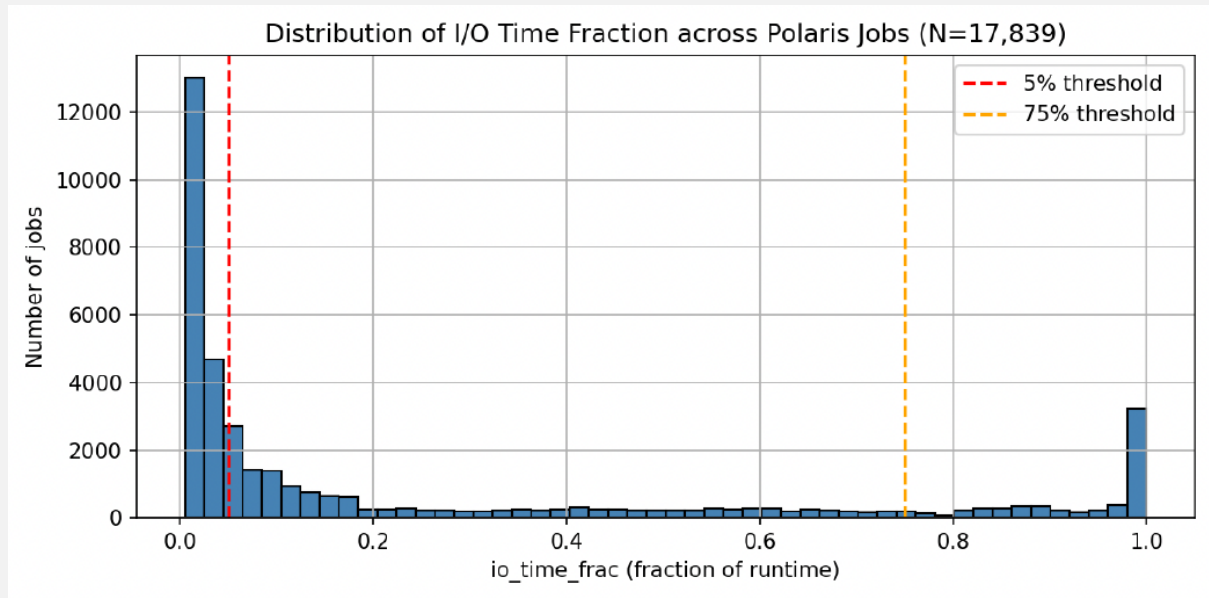


ANL Polaris Implementation

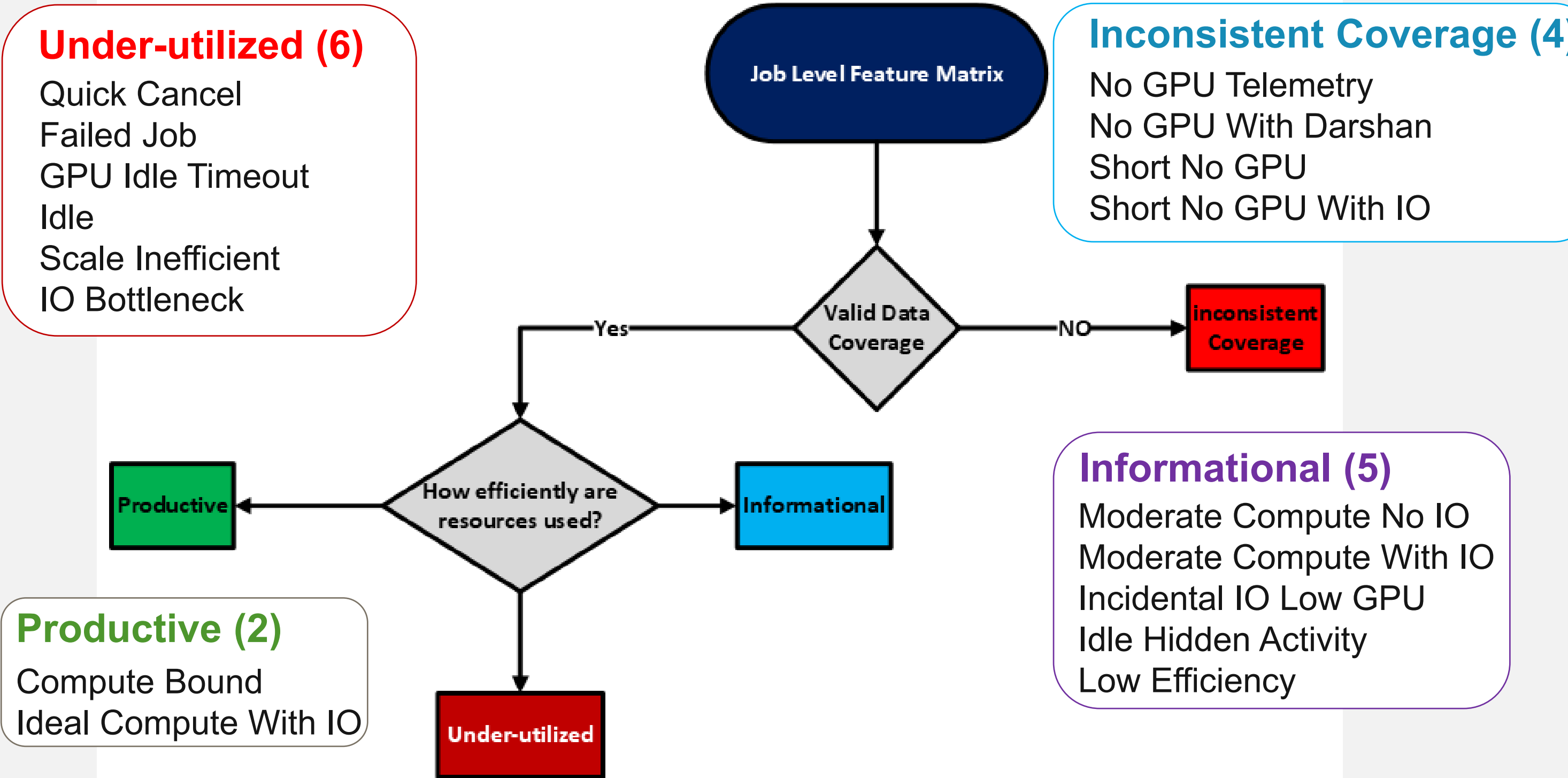
262,634 jobs across 13 months, totaling 16.53M allocated GPU-hours

Source	Job-level information	Coverage
Scheduler	Allocation, runtime, exit status	All jobs
DCGM	GPU utilization, power, thermal state	78.1%
Darshan	I/O timing, bytes, metadata, imbalance	25.2%

Joint DCGM + Darshan coverage: 18.4%. Missing telemetry is dismissed.

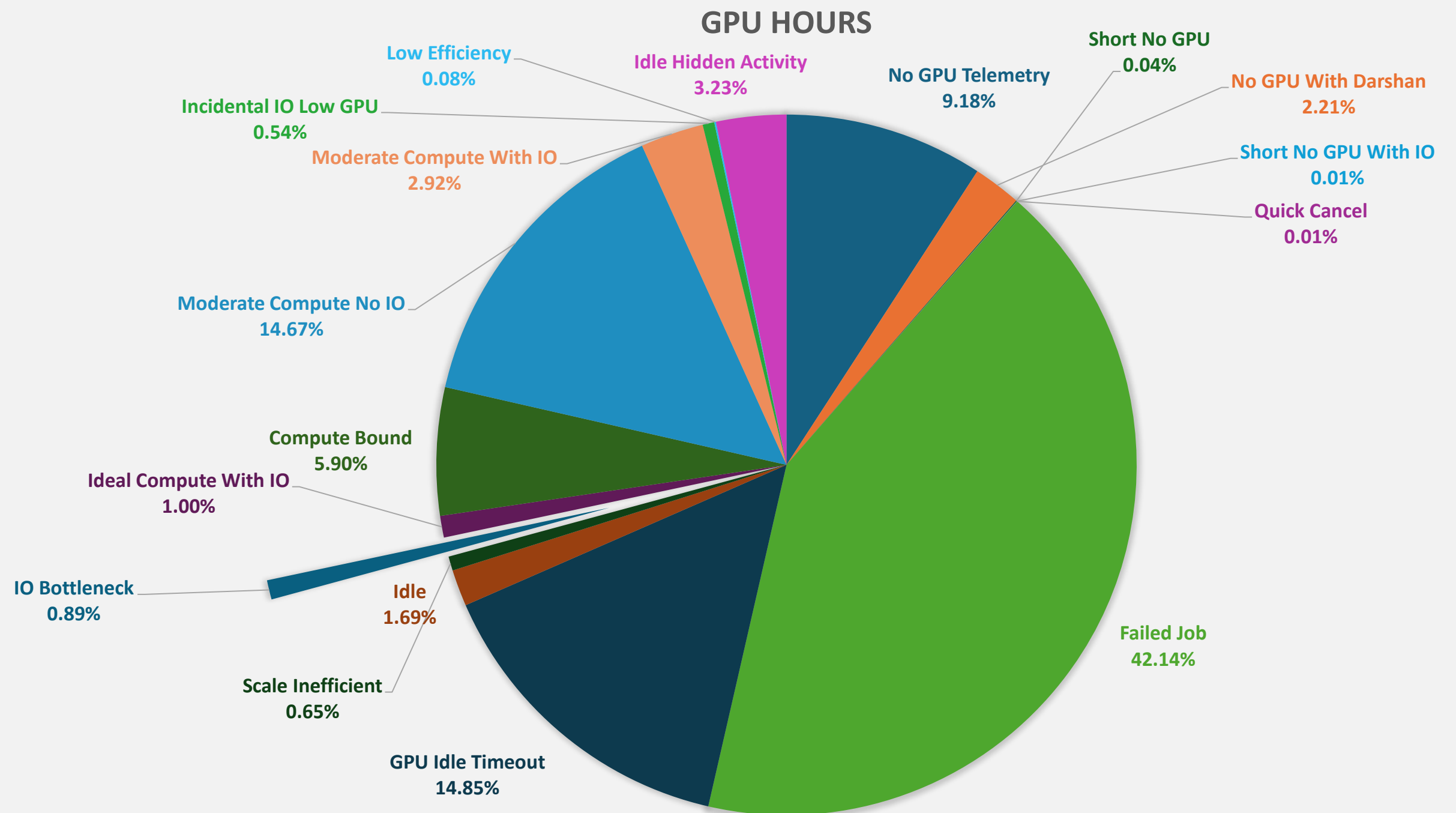


Workload Taxonomy



Cross-Layer Workload Taxonomy on Polaris

Tier	Jobs	Alloc. GPU-hrs	% GPU-hrs
Excluded from predictive modeling			
No_GPU_Telemetry	11,334	1,516,516	9.2
No_GPU_With_Darshan	850	365,707	2.2
Short_No_GPU	4,644	6,628	0.0
Short_No_GPU_With_IO	686	1,620	0.0
Subtotal	17,514	1,890,471	11.4
Under-utilized			
Quick_Cancel	94,439	7,185	0.0
Failed_Job	53,392	6,963,090	42.1
GPU_Idle_Timeout	25,630	2,454,515	14.8
Idle	9,836	278,728	1.7
Scale_Inefficient	995	108,179	0.7
IO_Bottleneck	5,776	147,222	0.9
Subtotal	190,068	9,958,919	60.25
Productive			
Ideal_Compute_With_IO	282	164,992	1.0
Compute_Bound	5,971	974,265	5.9
Subtotal	6,253	1,139,257	6.9
Informational			
Moderate_Compute_No_IO	20,665	2,423,423	14.7
Moderate_Compute_With_IO	2,871	482,304	2.9
Incidental_IO_Low_GPU	1,512	89,160	0.5
Low_Efficiency	1,640	12,740	0.1
Idle_Hidden_Activity	22,111	534,164	3.2
Subtotal	48,799	3,541,791	21.4
Total	262,634	16,530,438	100.0



Decomposition of I/O Bottleneck Class

- 72.9% of I/O-bottleneck jobs exhibit multiple diagnostic signatures.
- The decomposition helps identify what to investigate next: data-transfer rates, metadata operations, or uneven I/O across ranks.
- All 12,605 MPI-IO jobs with a recovered `cb_nodes` value recorded `cb_nodes = 4`. This identifies a configuration worth investigating for larger allocations.

Mode	Jobs	%
Bottleneck signature		
Bandwidth-only (effective rate < 1000 MB/s)	1,186	20.5
Metadata-only (> 1000 ops/GB)	83	1.4
Rank-imbalance-only (> 2x slowest/fastest)	68	1.2
Mixed	4,208	72.9
Read/write temporal dominance		
Read-dominated (read time > 1.5x write time)	857	14.8
Write-dominated (write time > 1.5x read time)	2,219	38.4
Balanced or interleaved	2,700	46.7
Burstiness		
Sustained (≤ 2 bursts)	2,433	42.1
Moderate (3–4 bursts)	1,230	21.3
Bursty (≥ 5 bursts)	2,113	36.6
Concurrent read/write (> 10% overlapping bins)	4,001	69.3

Key Takeaways

- 50.7% of under-utilized GPU-hours come from the top 10 of 1,008 users
- 83.7% of under-utilized GPU hours come from the top 51 users
- 86.6% of substantive under-utilized jobs occur in consecutive user bursts
- Recurring workload histories create an opportunity for targeted support
- Make users aware of their utilization and suggest potential optimization strategies
- Use better allocation strategies such as creating different job partitions
- Having historical utilization context of the user helps to design targeted interventions
- Publicly available Polaris dataset was used for this evaluation, but any facility can collect and process the information and understand system utilization using this framework

Optimizing Persistent Collectives Using RMA and Earlybird Communication

Jackson J. S. Wesley¹, Matthew G. F. Dosanjh², Whit Schonbein², Amanda Bienz¹

¹Department of Computer Science, University of New Mexico ²Center for Computing Research, Sandia National Laboratories

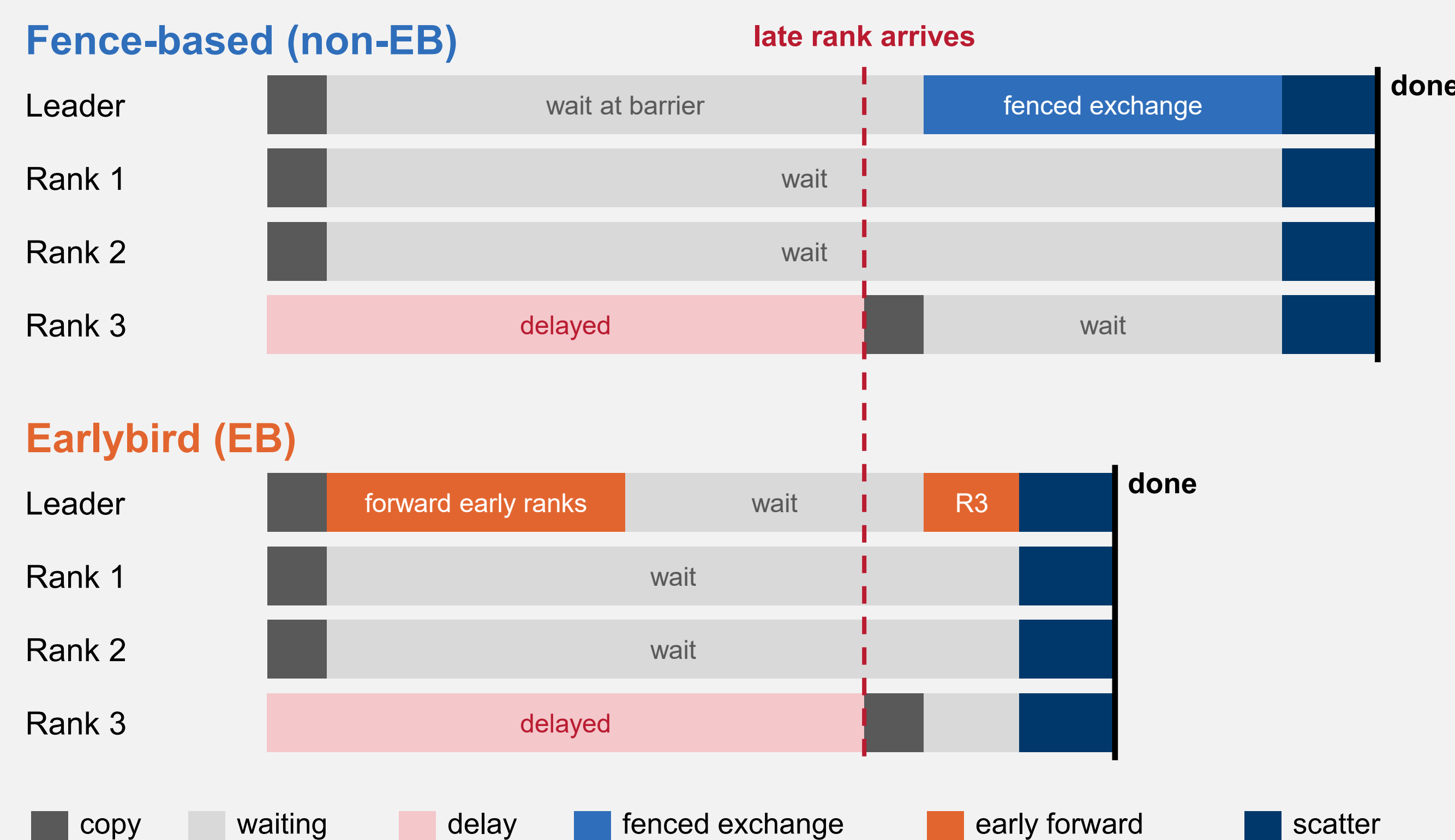
Background

Collectives need every rank. When one rank arrives late (OS noise, load imbalance), every other rank waits.

Hierarchical collectives copy data on-node, exchange it between node leaders, then scatter it. Fence-based versions synchronize between every phase.

Earlybird (EB): moves data as soon as it is ready. Early ranks' data is already off-node when the late rank arrives, so less work remains.

Thus, a persistent EB Allgather and Alltoall on MPI one-sided (RMA).



Illustrative timeline, not measured data. Non-EB cannot start the inter-node exchange until every rank reaches the barrier; EB forwards early ranks' data while the late rank is delayed.

Results: Earlybird vs. Non-Earlybird

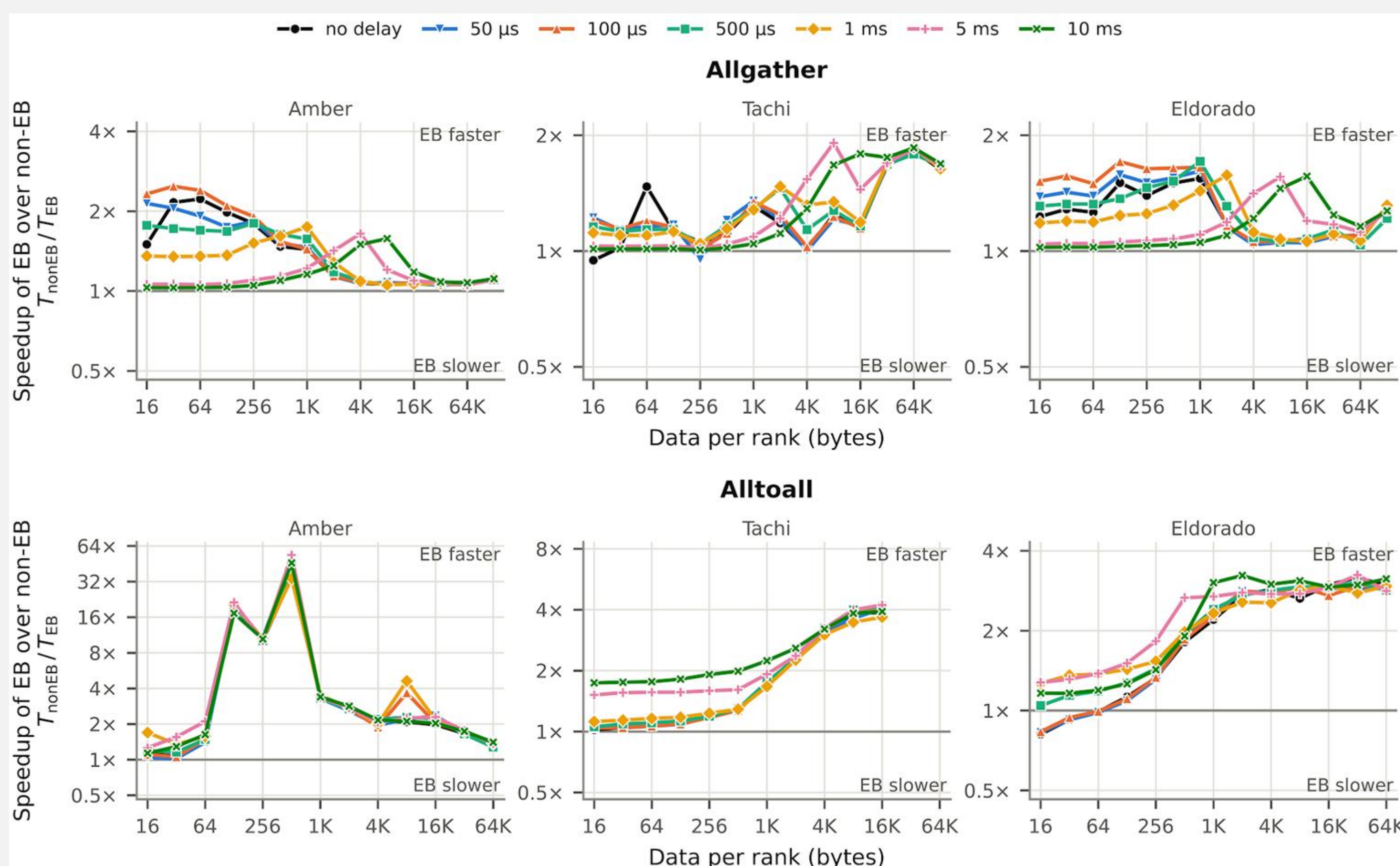


Fig. 4: Speedup of EB over non-EB (T_{nonEB} / T_{EB}), 32 nodes $\times$ 64 ranks/node, one line per injected delay.

	vs. non-EB	vs. system MPI
Allgather	1.2–2.6 $\times$ faster below 1 KiB	Faster at most sizes (6–20 $\times$ at 1 KiB); slower at ≤ 64 B on Cray systems
Alltoall	1.7–4 $\times$ faster from 1 KiB	Faster at small to medium sizes only (Eldorado 128 B–1 KiB)
P3DFFT	2.5–57.1% faster	Faster when per-destination messages are small

P3DFFT: P3DFFT++ forward + backward transform, 64 ranks/node, no injected delay. System MPI is included for context as vendor MPIs use optimizations on the algorithms chosen, so we compare a similar collective with different leader exchange mechanisms.

Methods

- Persistent API:** init builds windows, communicators, and flags once; Start/Wait per call.
- Three hierarchical phases:** on-node copy, leader exchange, on-node scatter.
- Non-EB:** barrier and fence between phases.
- EB:** each rank sets a generation flag; the leader does MPI_Put of that rank's data as soon as the flag appears.

System	CPU	Network	MPI
Amber	Xeon 8480+, 2 $\times$ 56 cores	Omni-Path, 100 Gb/s	Open MPI 4.1
Tachi	Xeon Max 9480, 2 $\times$ 56 cores	Slingshot 11, 200 Gb/s	Cray MPICH 8.1
Eldorado	EPYC 24-core, 4 sockets (MI300A)	Slingshot 11, 200 Gb/s	Cray MPICH 9.0

Benchmark: Scaling test was performed up to 64 nodes. For further evaluation, 32 nodes $\times$ 64 ranks/node is the tested configuration, with 16 B to 64 KiB per rank.

Delay injection: each iteration, one random non-leader sleeps before entering the collective.

Absorbing Delay

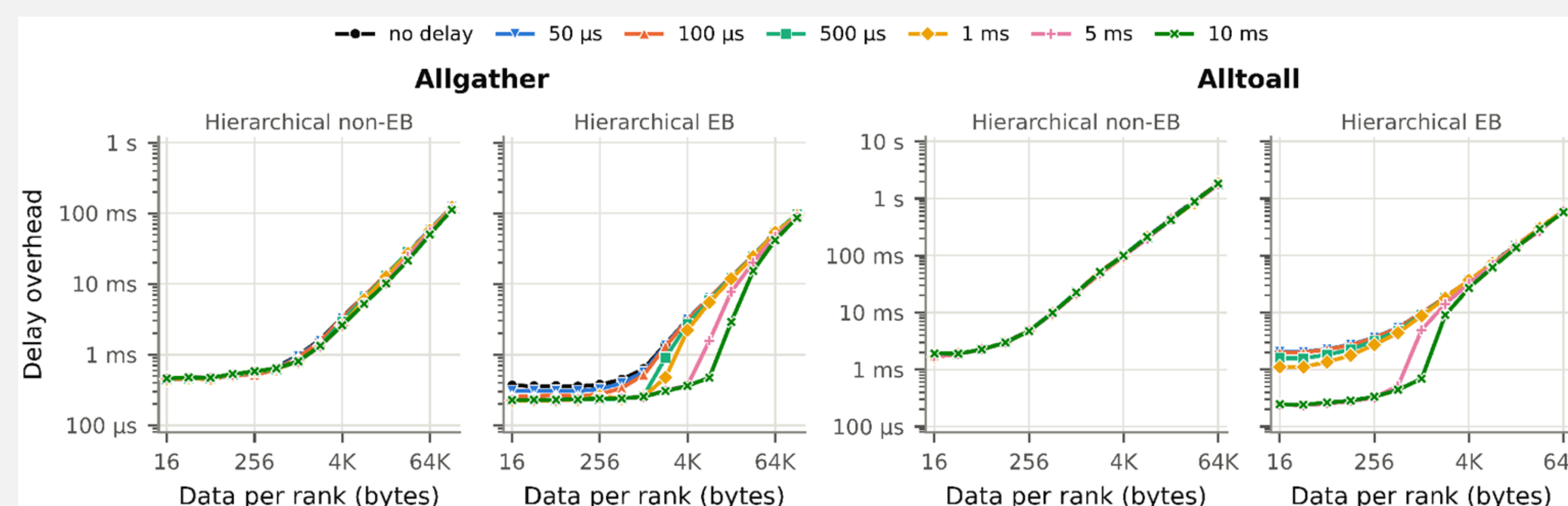


Fig. 3: Delay overhead, Eldorado, 32 nodes $\times$ 64 ranks/node.

- Non-EB** pays the delay in full: every delay curve lies on the undelayed one.
- EB** overhead falls as delay grows. At 10 ms, Alltoall (≤ 64 B) overhead is 0.25 ms vs. 2.0–2.3 ms undelayed.
- Under benchmark run (128 B) for Alltoall, Hierarchical EB completes in 2.06 ms undelayed, and 2.10 ms under a 1 ms delay. Non-EB completes this in 1.90 ms undelayed, and 2.90 ms under 1 ms delay.

Conclusions

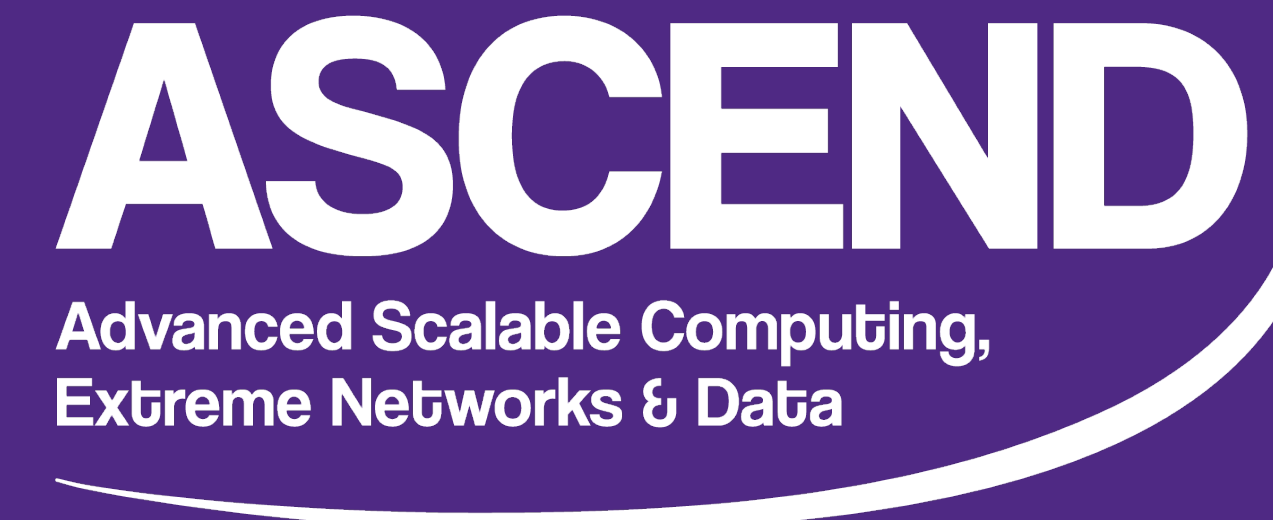
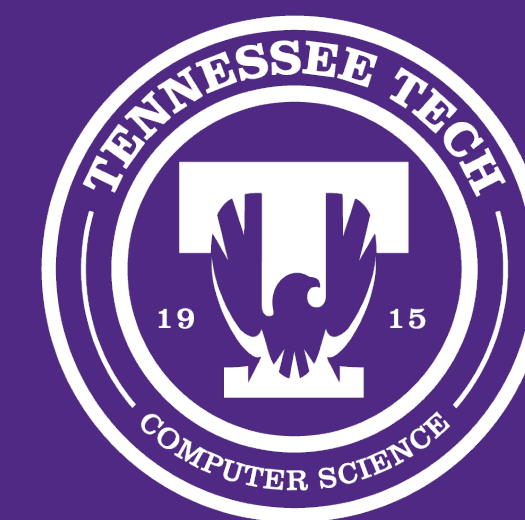
- EB beats fence-based non-EB at almost every size, with or without delay.
- Against system MPI: EB Allgather is 6–20 $\times$ faster at 1 KiB; EB Alltoall is competitive at small and medium sizes only.
- Setup is a one-time cost, recovered within a few calls in most cases (vs MPI, this is in the regimes where benefit is seen).
- Next:** Triggered operations, other earlybird methods, optimize with leader and data-pooling.

References

- [1] Parsons & Pai, ICS 2015.
- [2] Qian & Afsahi, IJPP 2011.
- [3] Holmes et al., ExaMPI 2021.
- [4] Pekurovsky, SIAM J. Sci. Comput. 2012.
- [5] Kinkead, Wesley et al., SC '25 Workshops.

Acknowledgements

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.



Tennessee
TECH

OpenRAPIDS Cascade: Kernel-Initiated Data Transfers on Slingshot-11

Thomas Hines, Anthony Skjellum, and Riley Shipley | ASCEND and Dept. of Computer Science, Tennessee Technological University

OpenRAPIDS

Lower middleware (*CCLs, MPI implementations) needs a **few data-transfer operations, done fast**. OpenRAPIDS (Reduced API Data-transfer Specifications) is four independent primitives on one shared runtime, **split by what triggers a transfer**.

	Plenum	Cascade	Mobius	Herald
trigger	host	kernel	host	host
on-node	shared mem	GPU copy	shared mem	NIC/MPI
off-node	mpi, cxi	cxi	mpi, cxi, verbs	mpi, cxi
API	experimental	stable	stable	stable

Why Cascade

- ▶ On Slingshot-11 with AMD GPUs, existing paths have the **host build the network work in advance** [1, 2].
- ▶ When a GPU computes and its host communicates, **exchanges cross between GPU and host**.
- ▶ Cascade removes the crossing: **the kernel that produced the data sends it**.

How Cascade works

- ▶ A running kernel **writes its own NIC commands** into a Cassini command queue through `libcxi`, with **no host on the transfer path**.
- ▶ Commands are prebuilt once; a post patches address, length, and slot, then issues **one 128-byte store**.
- ▶ In **flow-controlled slots**, delivery frees the source and a credit frees the slot.
- ▶ Slots **held across kernels** let communication overlap computation.
- ▶ On one node, the same kernel code **copies GPU to GPU**.

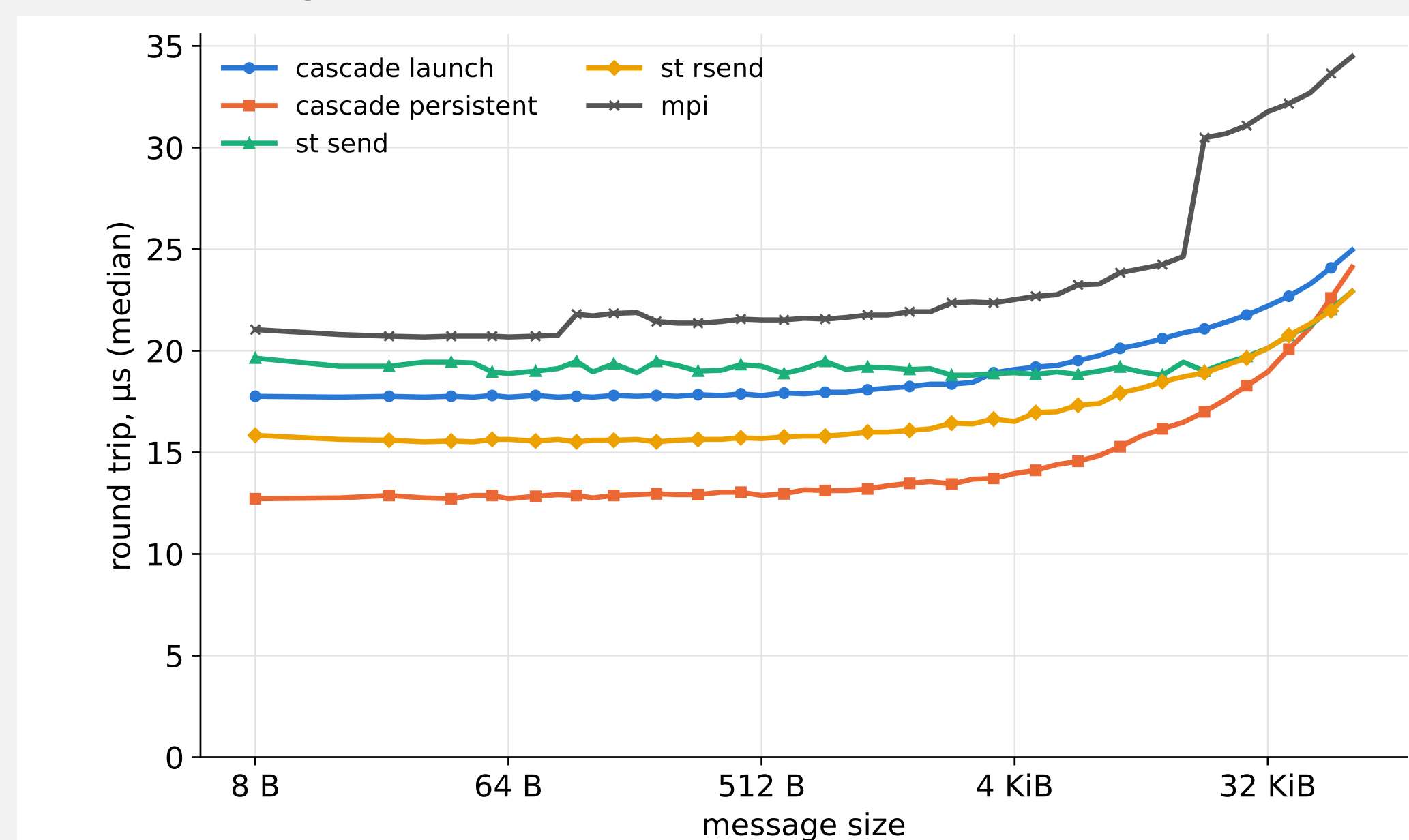
Open question: the interface

- ▶ **Kernel-initiated only** (today): fastest; needs a kernel.
- ▶ **Two levels**: these verbs plus a stream-ordered call.
- ▶ **Stream-triggered only**, accepting the overhead: a kernel per step takes 8 B from 12.7 to 17.8 μ s.

What we measured

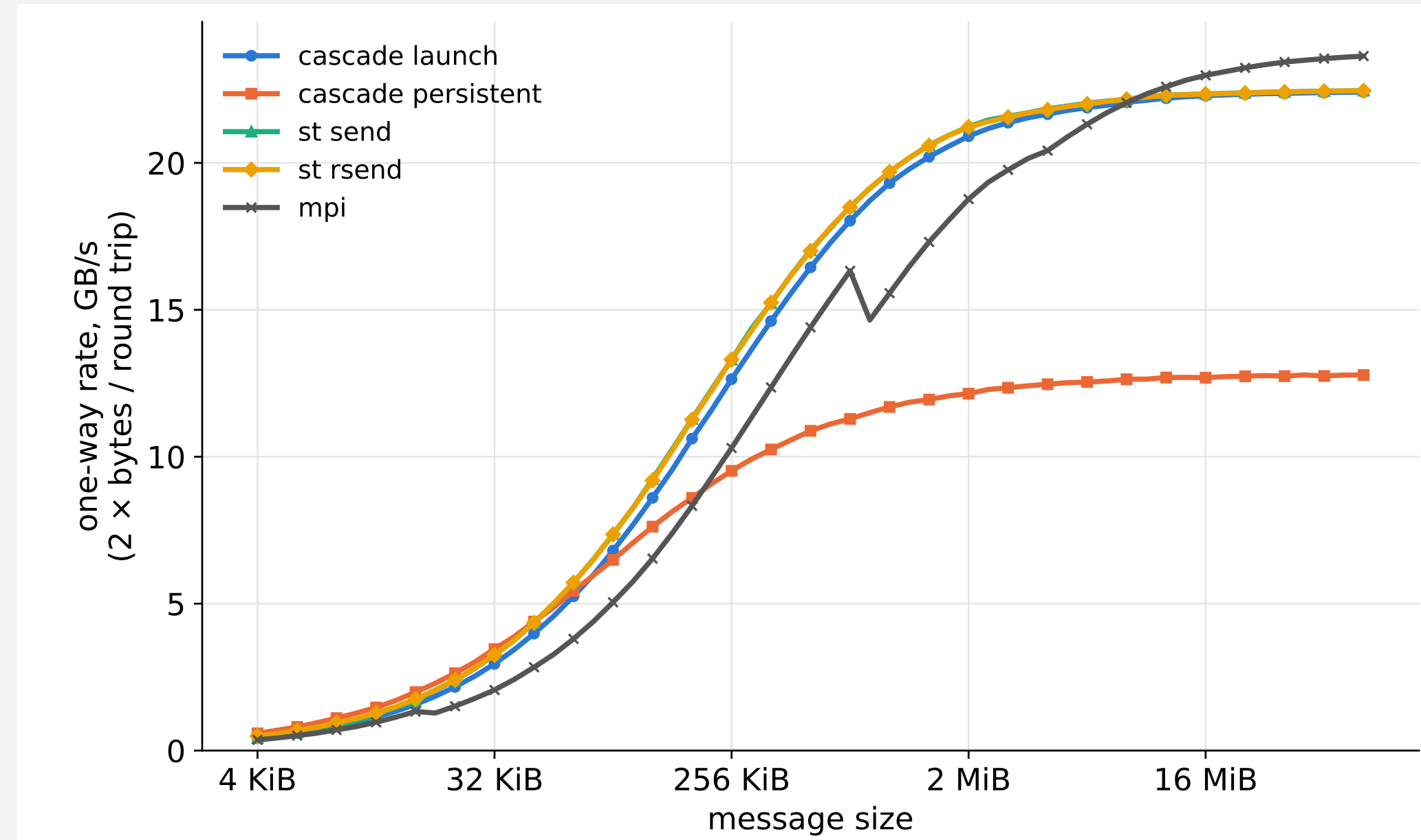
Rally: a GPU ping-pong; each side adds to the vector before returning it. **cascade launch**: a kernel per turn; **persistent**: one resident kernel. **st**: MPI Advance stream-triggering [3], MPI started from the GPU stream. **MPI**: Cray MPICH 9.1.0; the host waits each turn.

Latency between two nodes



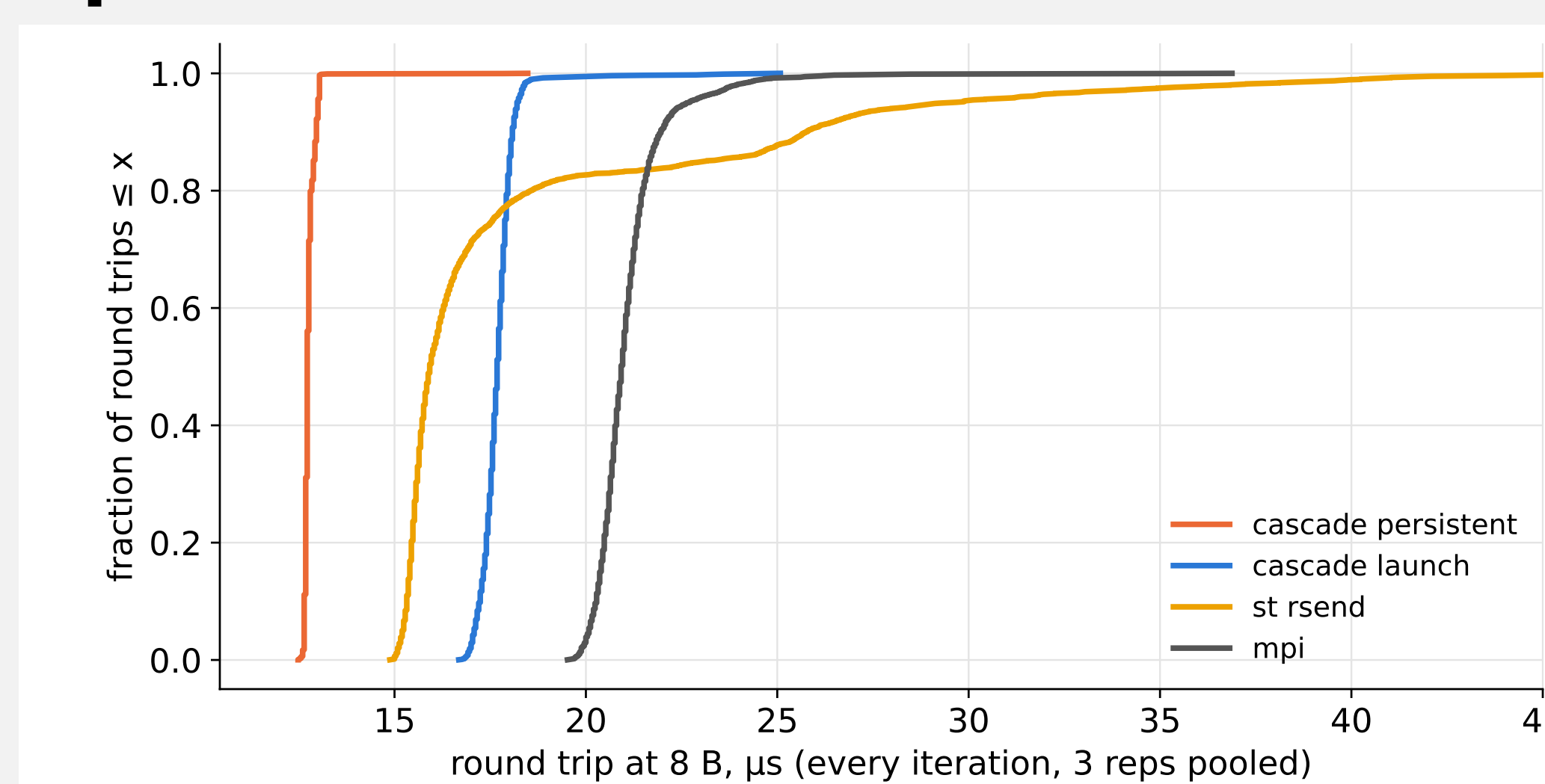
Rally between two nodes of LLNL Tuolumne (AMD MI300A, Slingshot-11): median round trip of 3 runs.

Bandwidth between two nodes



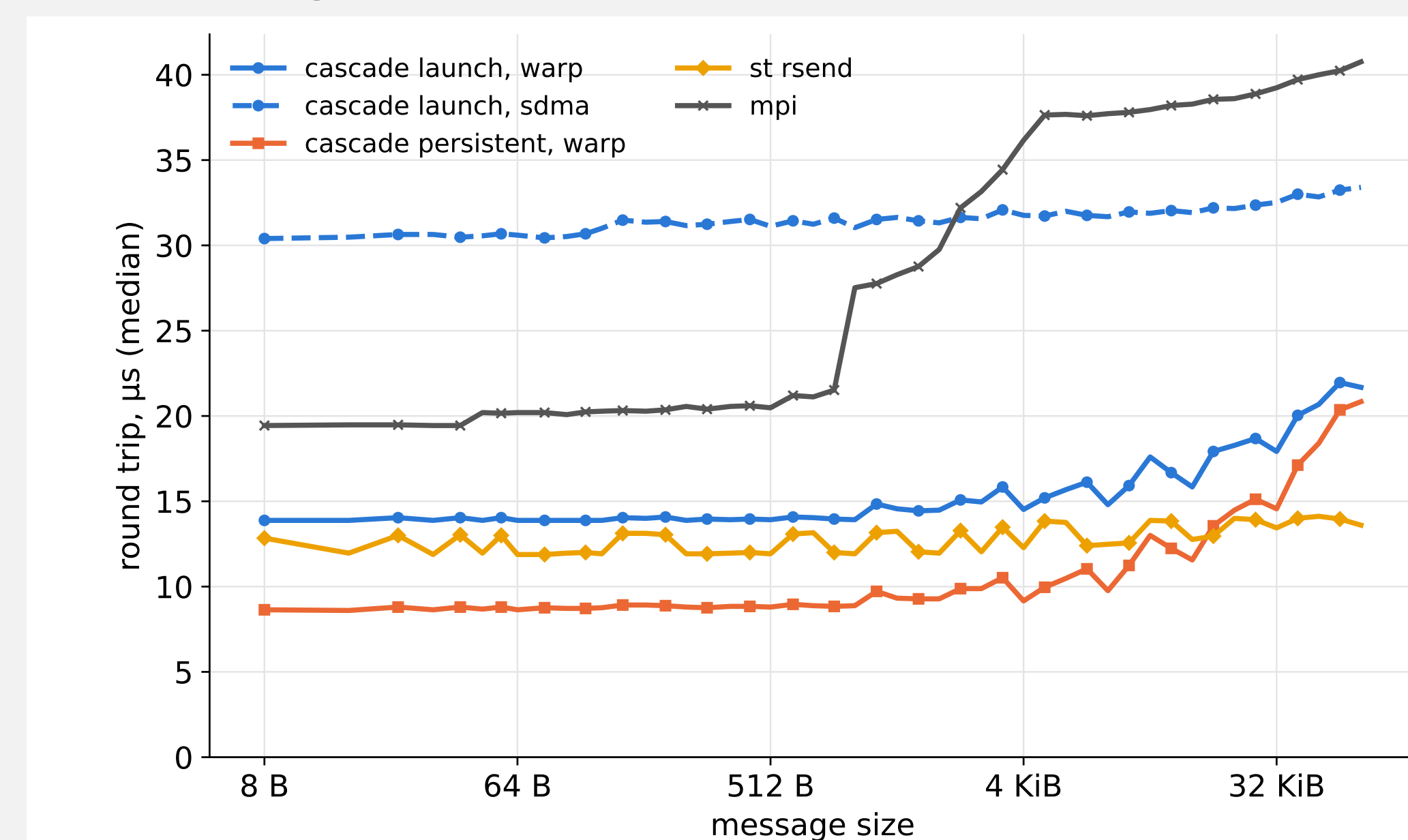
Persistent levels off near 12.8 GB/s: its single block also does the add.

Spread at 8 B between nodes



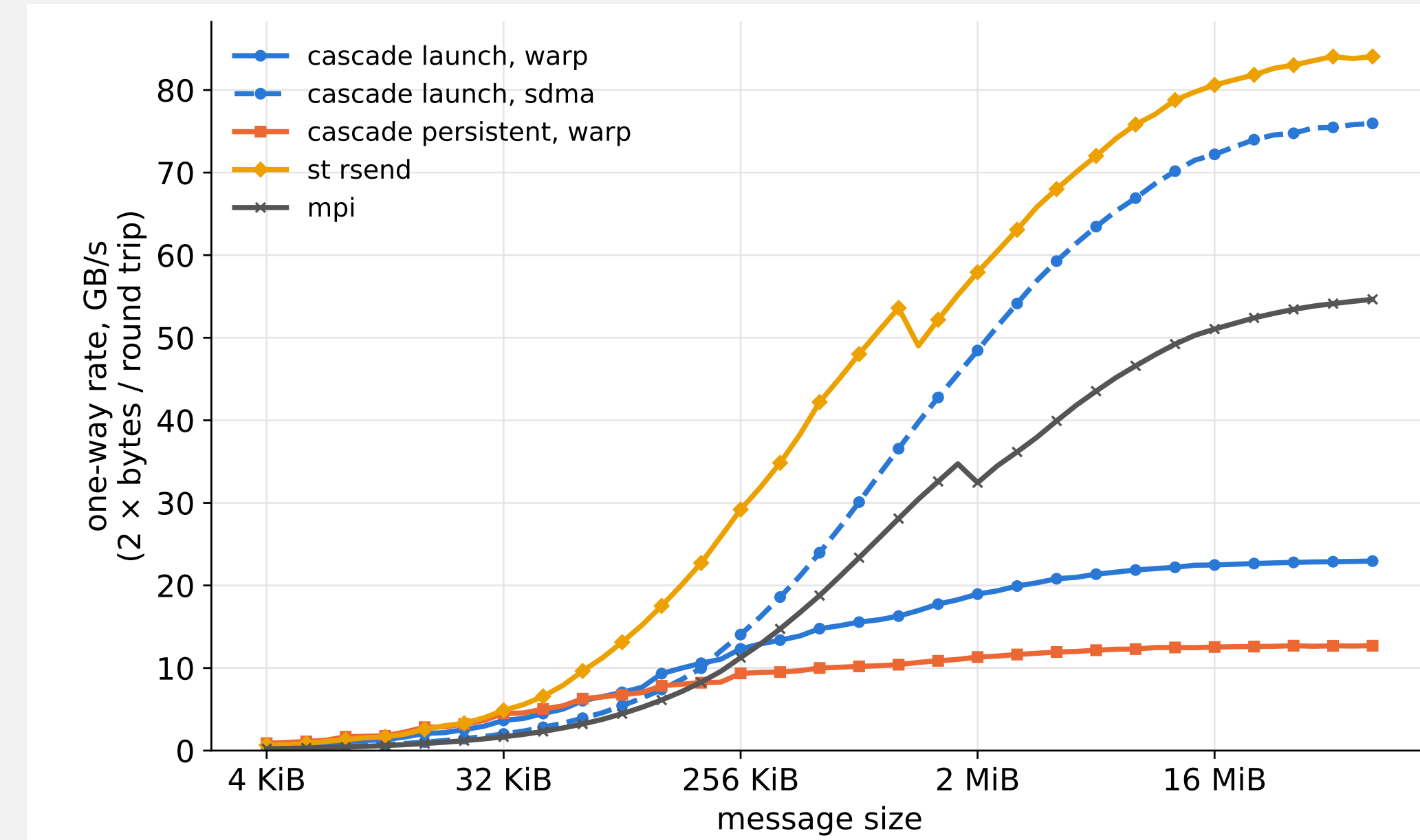
3,000 round trips per line, 3 runs pooled. 98% of persistent round trips fall within 12.6–13.0 μ s, Cray MPICH's within 19.8–24.6. Stream-triggered MPI (ready send): 15.9 μ s median, but 17% of round trips exceed 20 μ s.

Latency on one node



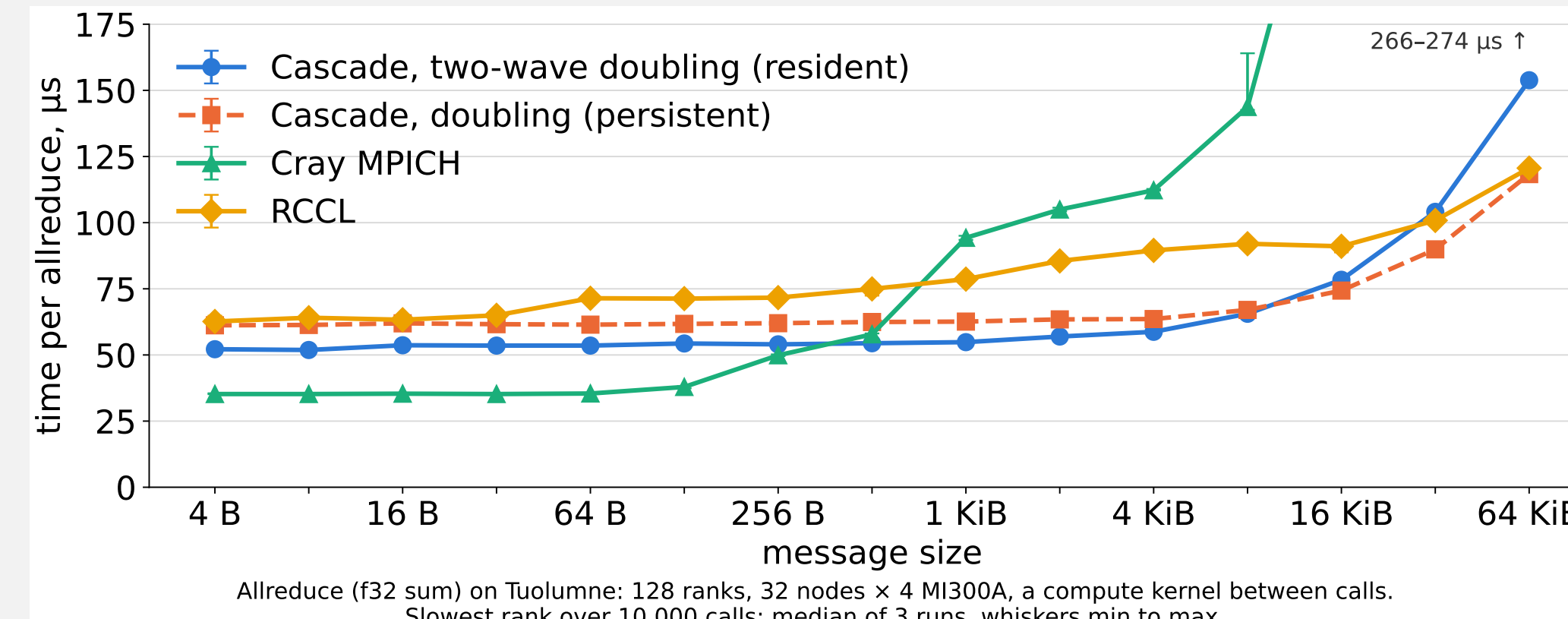
Rally on one node: median round trip of 3 runs. warp: one wave copies; sdma: two copy engines.

Bandwidth on one node



The copy engines carry Cascade to 76 GB/s at 64 MiB.

Allreduce at 128 GPUs



f32 sum on 128 GPUs (32 nodes $\times$ 4), a compute kernel between calls; slowest rank over 10,000 calls, median of 3 runs.

- ▶ **Cascade beats RCCL and Cray MPICH above 256 B through 32 KiB**, each in its best configuration.
- ▶ Fastest of all four from 512 B through 8 KiB; Cray MPICH leads at 256 B and below.

CXI to be Deprecated by HPE—Bad for DOE/NNSA

- ▶ Libfabric is too much and too little: not portable between providers, hides special features of specific networks, creates new resource limitations, and was not designed with GPUs in mind.
- ▶ It is “mid-level middleware”: an abstraction barrier.
- ▶ Outperforming RCCL on Tuolumne with Cascade on CXI shows that libfabric alone is suboptimal **even for AI applications**.
- ▶ Hand-coded or AI-assisted, we want to exploit the smartest features at the lowest level and drive them directly through kernel initiation.
- ▶ DOE/NNSA must retain access to lowest-level primitives and should press every network provider for them. NVIDIA has low-level APIs; all should. **CXI is a star example**.
- ▶ Interfaces may change; we want HPE to conserve **semantics and resources** (mechanisms, counters, registers), not precise interfaces. With generative AI, we can adapt quickly.

Current work

- ▶ **OpenCCL on Slingshot-11**: its collectives call Cascade’s kernel-side verbs directly.
- ▶ **Beyond CXI**: similar work with Cornelis; analogous work with BXI and Bull.
- ▶ **A stream-ordered interface** for callers without a kernel (the open question at left).

Selected references

- [1] N. Namashivayam, K. Kandalla, T. White, N. Radcliffe, L. Kaplan, and M. Pagel, “Exploring GPU stream-aware message passing using triggered operations,” <https://arxiv.org/abs/2208.04817>, 2022.
- [2] B. Shan, M. Araya-Polo, and B. Chapman, “GICC: A high-performance runtime for GPU-initiated communication and coordination in modern HPC systems,” in *Proceedings of the 35th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '26)*, 2026, pp. 45–57, arXiv:2604.22126. [Online]. Available: <https://doi.org/10.1145/3806645.3807576>
- [3] P. G. Bridges, D. Schafer, J. Lange, J. B. White III, A. Skjellum, E. Suggs, T. Hines, P. Bangalore, M. G. F. Dosanjh, and W. Schonbein, “Co-design and evaluation of a CPU-free MPI GPU communication abstraction and implementation,” in *Proc. IEEE International Conference on Cluster Computing (CLUSTER 2026)*, 2026, also arXiv:2602.15356.

Acknowledgments

This work was supported in part by the U.S. National Science Foundation under awards 2511818, 2514054, 2450093, 2450092, 2412182, and by the U.S. Department of Energy, National Nuclear Security Administration, under the Predictive Science Academic Alliance Program, award DE-NA0004267 (PSAAP-IV). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Energy.

AI disclosure (ACM guidelines): OpenRAPIDS and Cascade are built with Claude (Anthropic; Opus 4.8, Opus 5, and Opus 5.5) through test-driven development, with human experts in and on the loop; Thomas Hines is the human-in-the-loop lead designer. Claude and the authors wrote this poster’s text together, and Claude prototyped its diagrams; the charts are the authors’ measured data. The authors verified, and own, every claim and result.

Implementing Software Strong Progress RMA in ExaMPI

Javontae Martin, Kayla Humphrey, Riley Shipley, Anthony Skjellum | ASCEND Center, Tennessee Technological University

What is ExaMPI?

- ExaMPI is a C++-based MPI subset implementation that is the result of a collaboration between universities in both current and prior PSAAP centers.
- ExaMPI is currently being used as a testbed for development and testing of experimental MPI features
- The progress engine is based on Dimitrov's strong progress model [1], which describes a process for handling communication requests that enables overlap. The full progress engine design is shown in Figure 1.
- The main benefit of this model is that it allows communication to progress in the background, so the user thread does not have to spend time explicitly progressing communication.

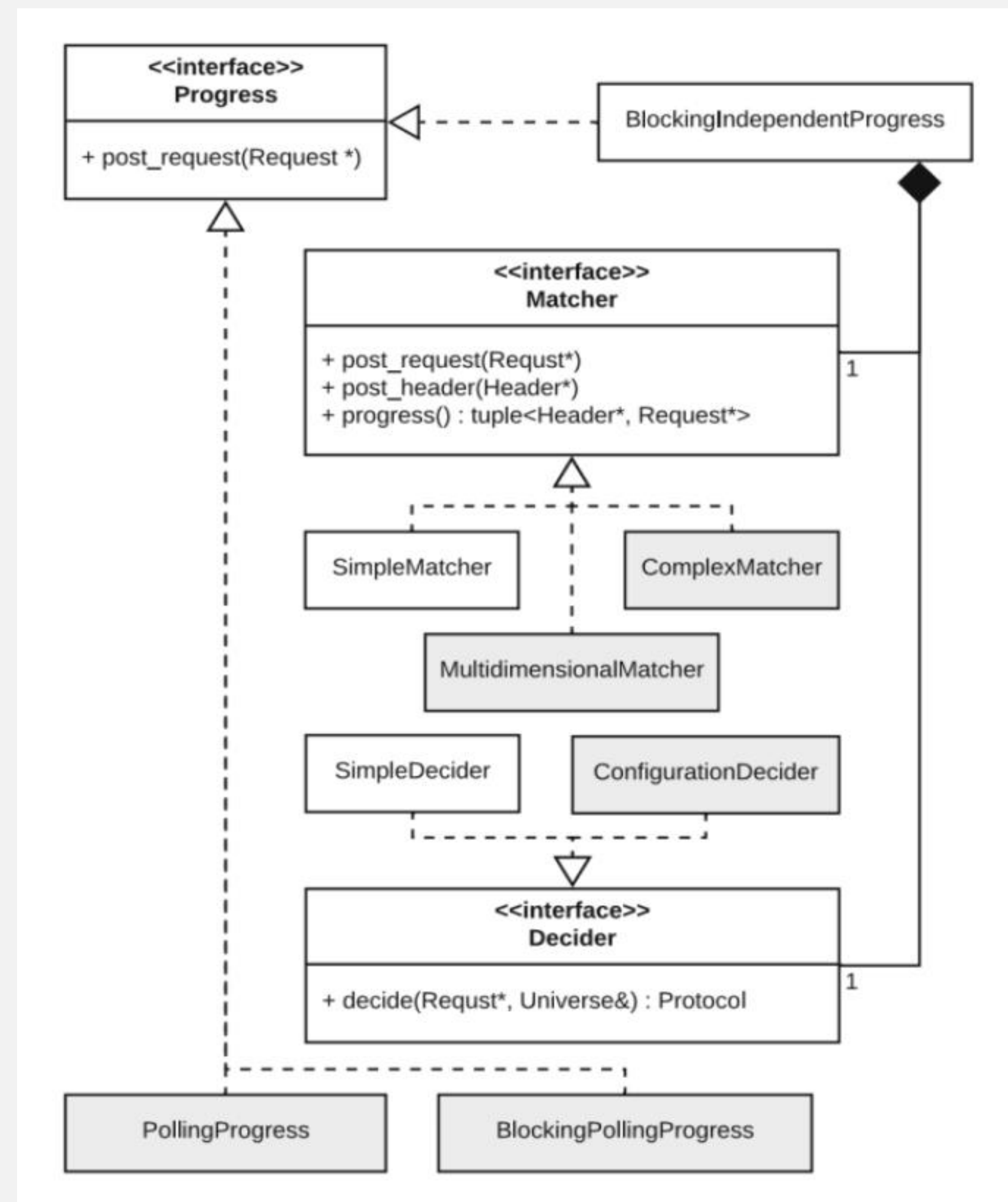


Figure 1: Structure of the progress layer of ExaMPI [2].

Strong Progress Model

- MPI RMA is often associated with bad performance, but it can be made better by adding strong progress.
- Figure 1 shows the three core parts of ExaMPI's strong progress engine: the Transport, Algorithm, and Matcher components.
- Transport is responsible for moving data, Algorithm sets the parameters Transport uses, and Matcher matches posted receives with incoming messages.
- Matching is unused, as it is unnecessary for RMA but necessary for message passing.

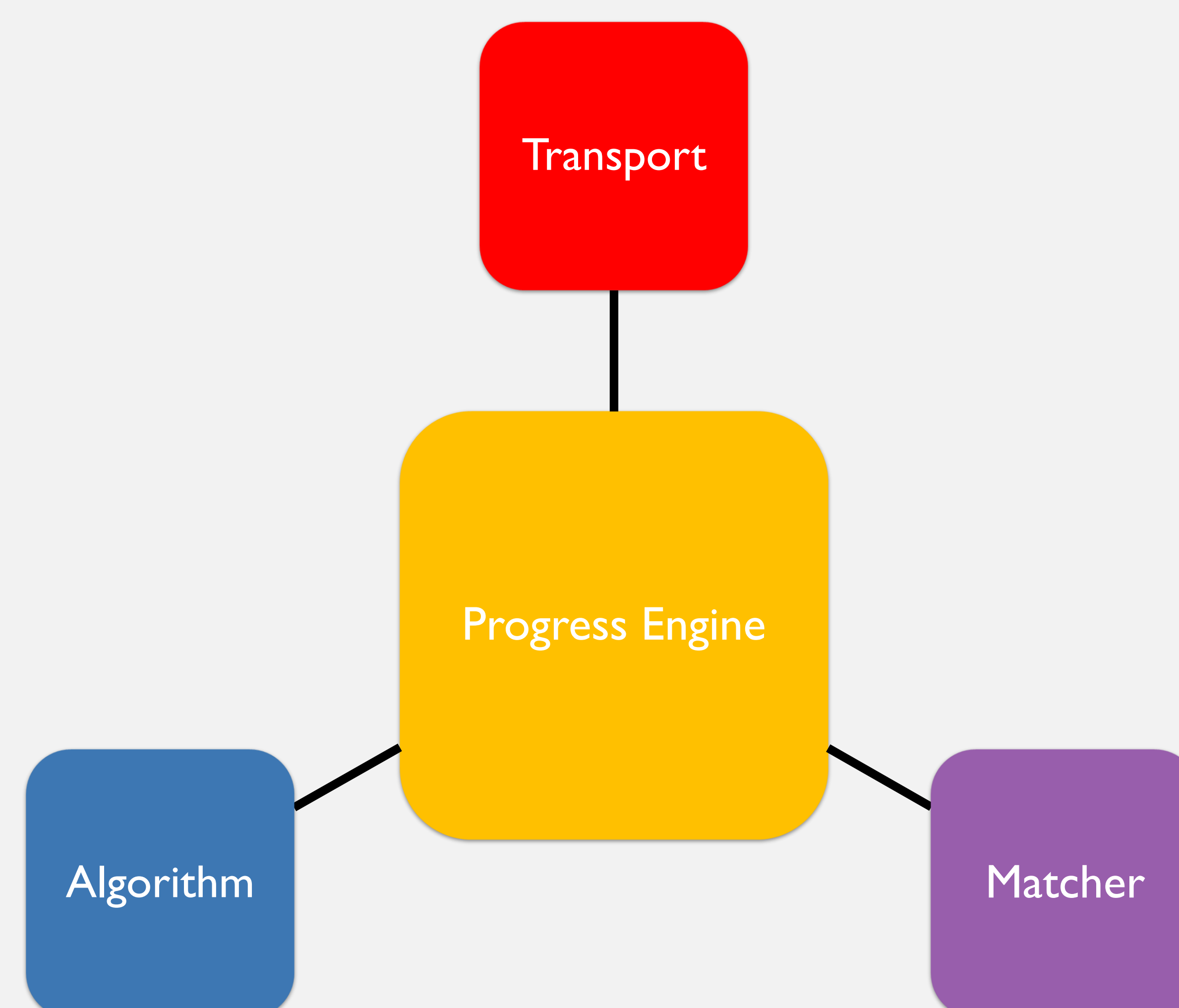


Figure 1: The core parts of the ExaMPI progress engine.

Implementation Design

Libfabric is the current networking stack used to implement RMA. ExaMPI's architecture is based primarily on the Request object, which stores all of the necessary information to inform progress engine operations. The progress engine consists of a fixed loop around the Algorithm, Matcher, and Transport progress functions. Each progress function reads values from the Request and acts accordingly. This model only works because the MPI_Fence definition allows for early completion of operations during each Window epoch [3].

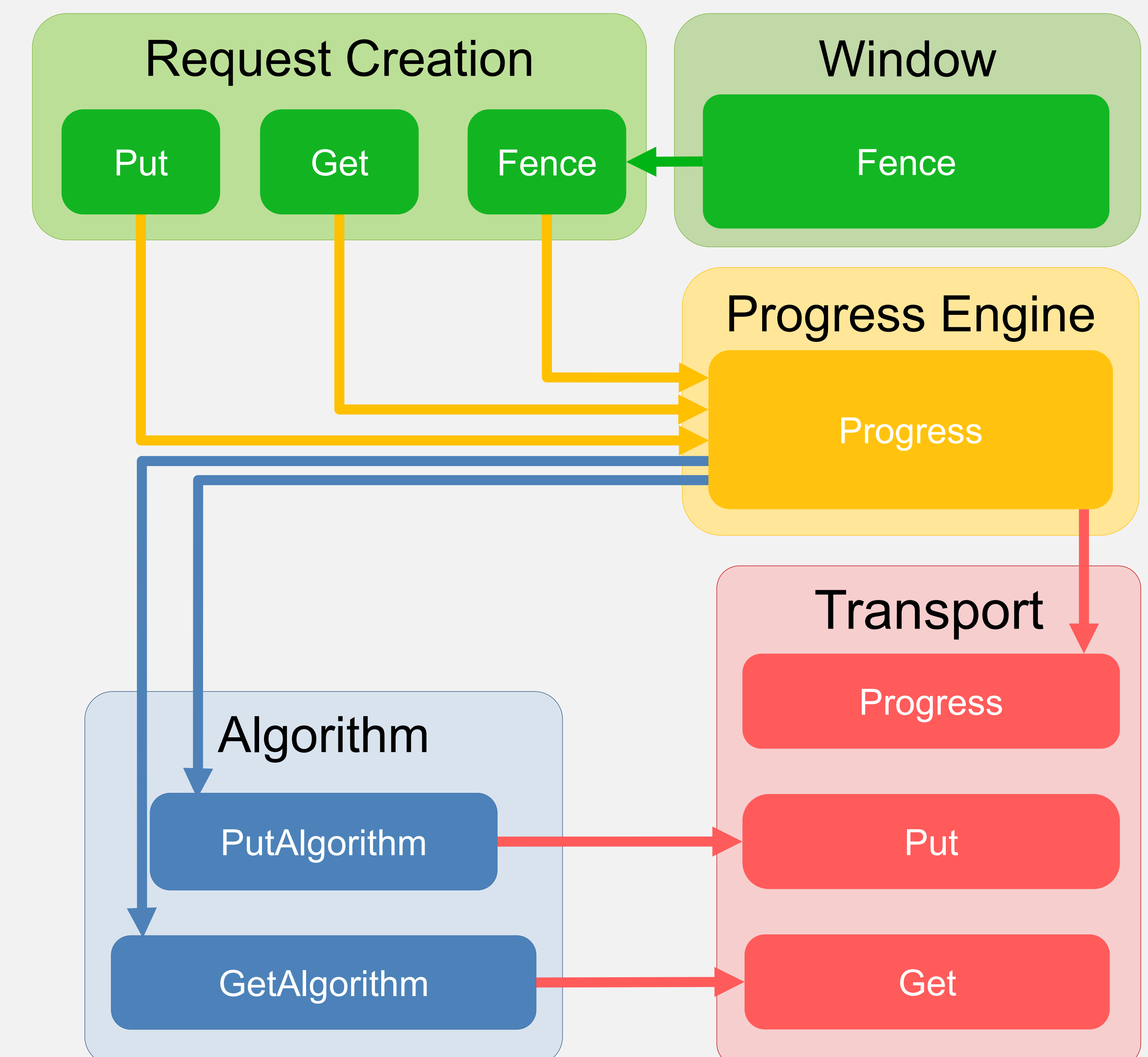


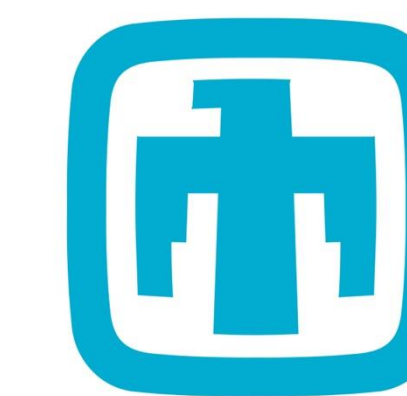
Figure 3: Flow of an RMA operation in ExaMPI.

References

- R. P. Dimitrov, "Overlapping of Communication and Computation and Early Binding: Fundamental Mechanisms for Improving Parallel Performance on Clusters of Workstations." Order No. 3016616, Mississippi State University, United States -- Mississippi, 2001.
- A. Skjellum, M. Rüfenacht, N. Sultana, D. Schafer, I. Laguna, and K. Mohror, "ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation," *Communications in Computer and Information Science*, pp. 153–169, 2020, doi: 10.1007/978-3-030-41005-6_11.
- Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.1 (2015). <https://www.mpi-forum.org/docs/mmpi-3.1/mmpi31-report.pdf>

Acknowledgements

Thanks to Derek Schafer and Thomas Hines for providing advice and review. This work was supported in part by the U.S. National Science Foundation under awards 2511818, 2514054, 2450093, 2450092, 2412182, and by the U.S. Department of Energy, National Nuclear Security Administration, under the Predictive Science Academic Alliance Program, award DE-NA0004267 (PSAAP-IV). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Energy.



Multi-Strategy Adaptive Optimization of High-Performance Communication

Shannon Kinkead, Advisor: Amanda Bienz, Mentor: Matthew Dosanjh | Department of Computer Science

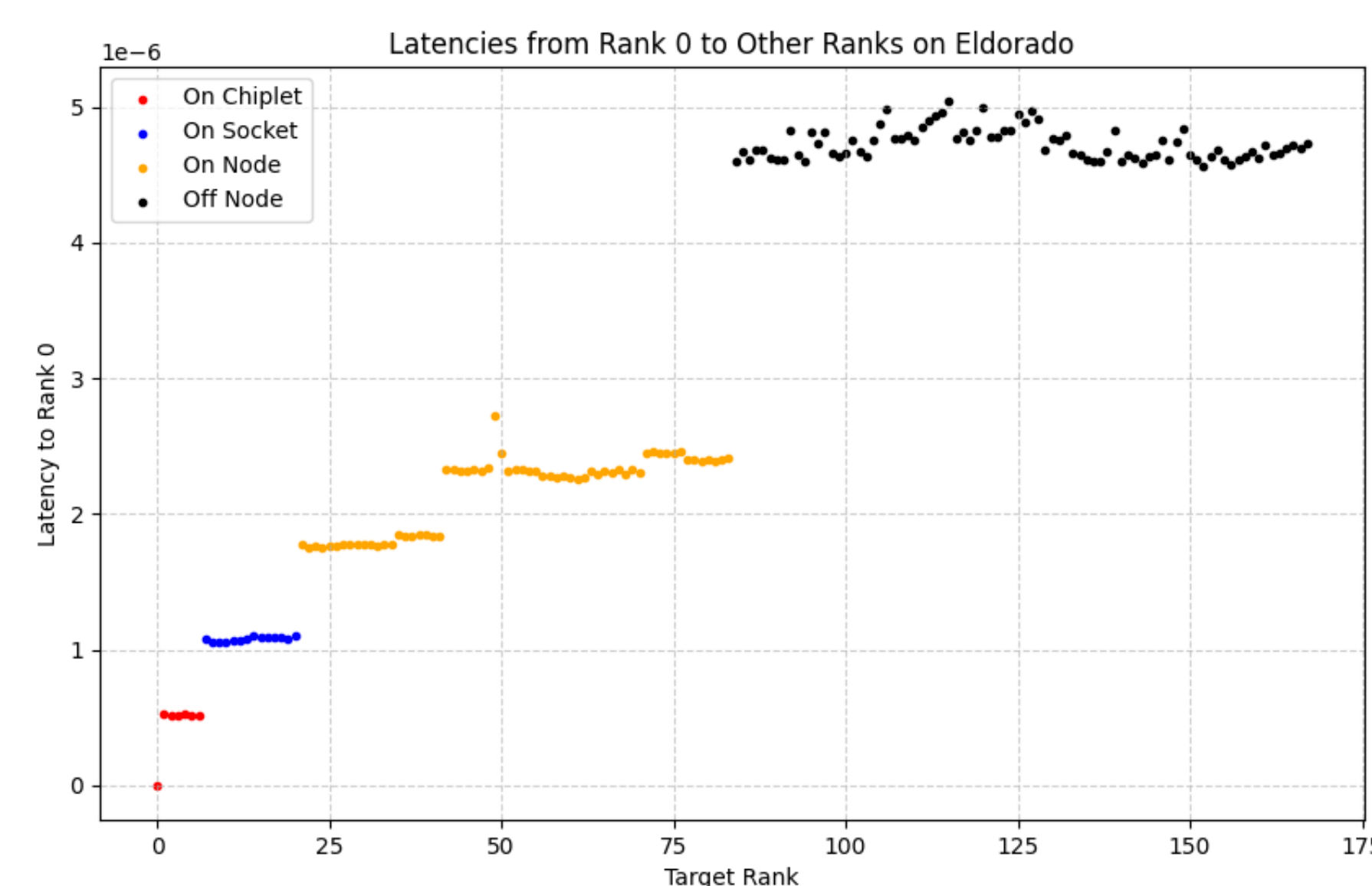
Dynamic Topology Discovery through Mahalanobis-Based Clustering

- Can we dynamically determine node socket/NUMA region membership based on ping-pong timings between each pair of ranks and balanced Lloyd's aggregation?
- On previous systems, on-node communication was less expensive than off-node, but on modern architectures, often on-node is more expensive
- Optimal locality-levels are architecture dependent
 - Determining the best level of locality requires running targeted benchmarks
- Allow MPI to determine which locality level is the best

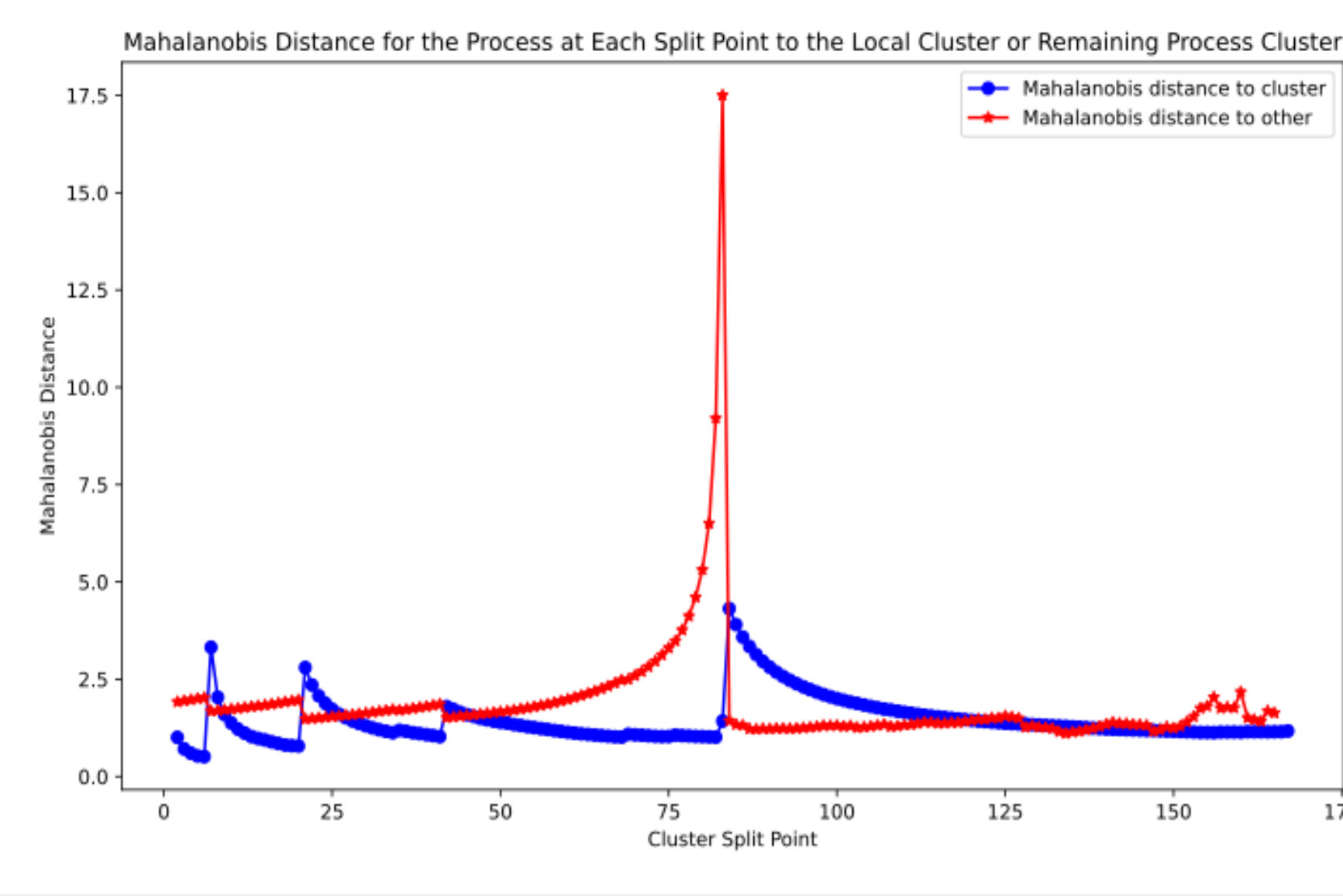
Clustering Algorithm

- Round-robin ping pong between every pair of ranks to build an adjacency matrix of latencies
- Initial rank cluster contains the current rank, and the closest rank
- Ranks are added to a cluster if the Mahalanobis distance of that rank's latency to the current cluster is less than 1.96.
- Union of the local clusters for all common ranks in the clusters are taken to ensure all ranks are in a cluster.
- Hierarchical clusters are built by adding the first process where the Mahalanobis distance exceeds 1.96, then adding subsequent processes if their Mahalanobis distance is less than the previous or less than 1.96

Results



Latency of each rank to rank 0, colored by relative locality on the node



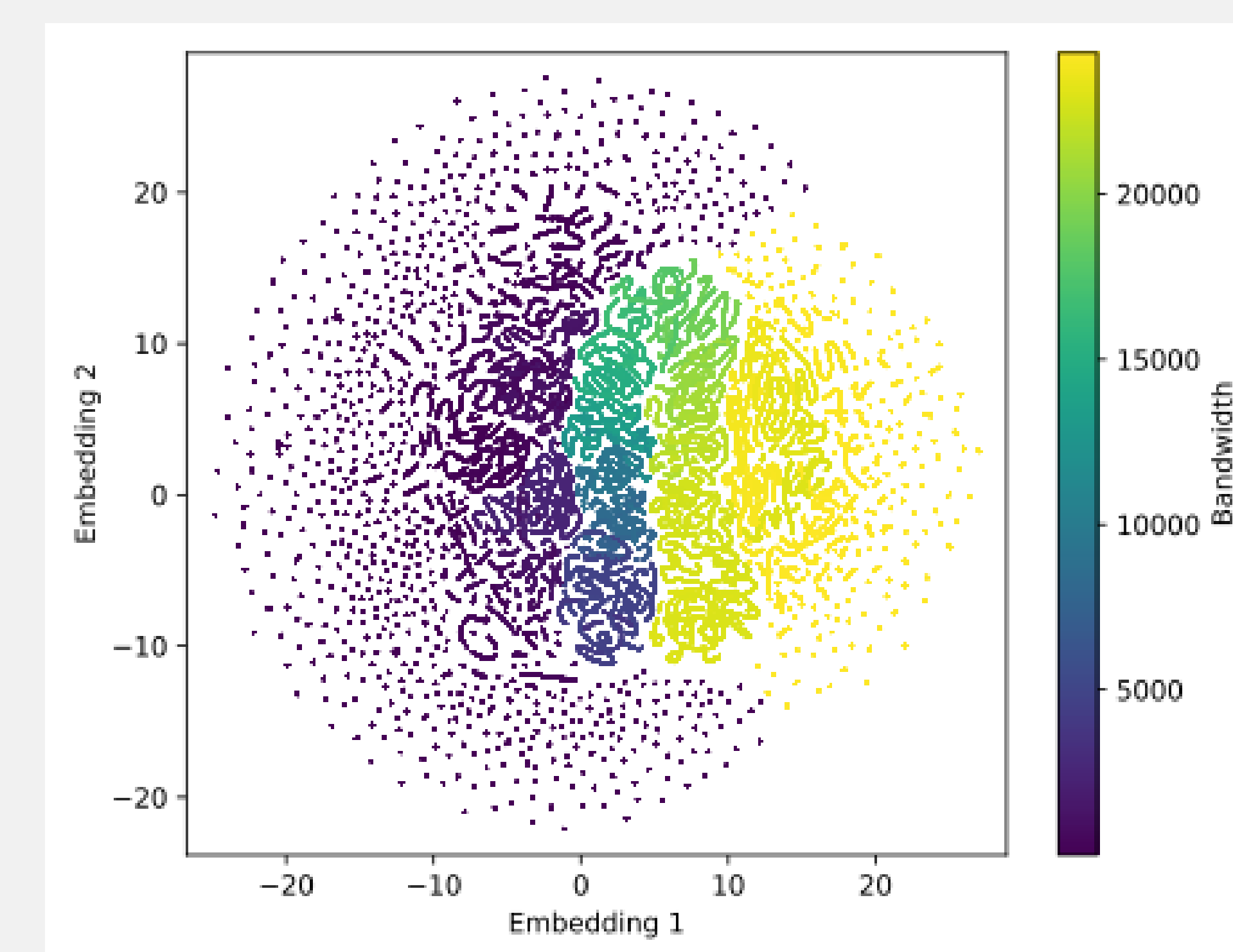
Mahalanobis Distance Calculated for each new rank, n , from the cluster being built (blue) and the cluster of all remaining processes (red)

Manifold-Aware Joint Optimization of MPI Tuning Parameters

- Can we determine optimal settings for multiple MPI tuning parameters at one time?
- Tuning MPI performance is a complicated process with many factors
- Complex network stack incorporating many libraries
- Multiple communication styles
- Heterogenous architectures
- Large number of tuning parameters, often with interacting impacts
- Approach:
 - Reduce the problem dimensionality
 - Optimize the reduced dimensions using derivative-free optimization techniques

Manifold Learning

- Performs dimensionality reduction of complex, non-linear surfaces
- Creates a lower-dimensionality embedding with closest topological structure to the original manifold



2 Dimensional UMAP Embedding of MPI Tuning Parameters and Bandwidth

Bayesian Optimization

- Derivative-Free optimization technique
- Effective on noisy objective functions
- Uses active learning and a surrogate model to identify additional points to sample
 - Often Gaussian processes
- Past objective function evaluations inform future point selection, reducing the data needed to find an optimal solution

Acknowledgements

This work was performed with partial support from NSF CCF-2151022, NSF CCF-2338077, DOE NNSA DE-NA0003966, and Sandia. Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA-0003525.

SAND# SAND2026-28191PE

This work was performed with partial support from the National Science Foundation under Grant Nos. CCF-2338077, CNS-2450092, and OIA-2521103 and the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program Award DE-NA0004267.

Optimizations for a CPU-free Stream Triggered Communication Interface

Derek Schafer, Patrick Bridges | University of New Mexico

Our Stream Triggering Implementation

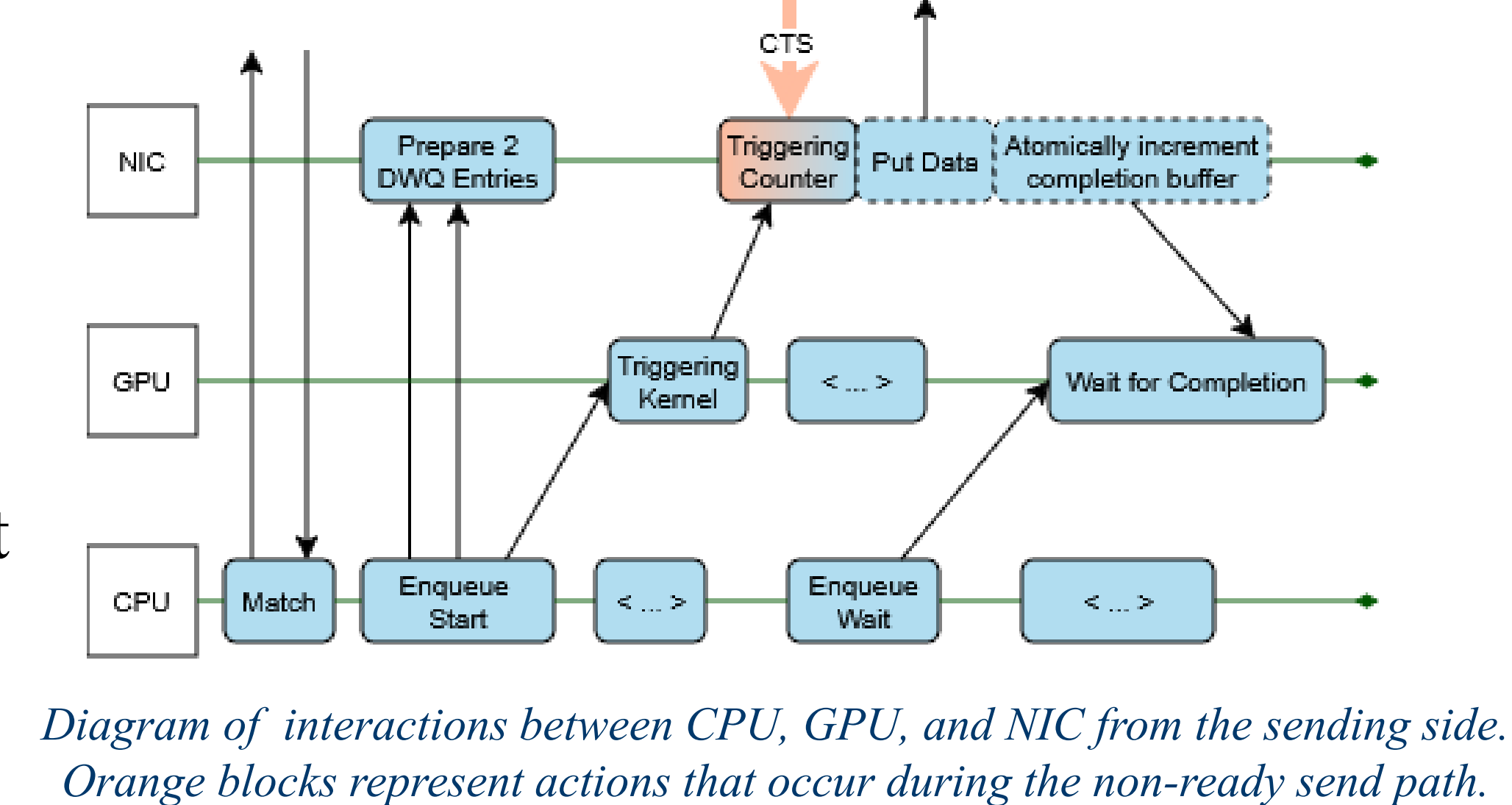
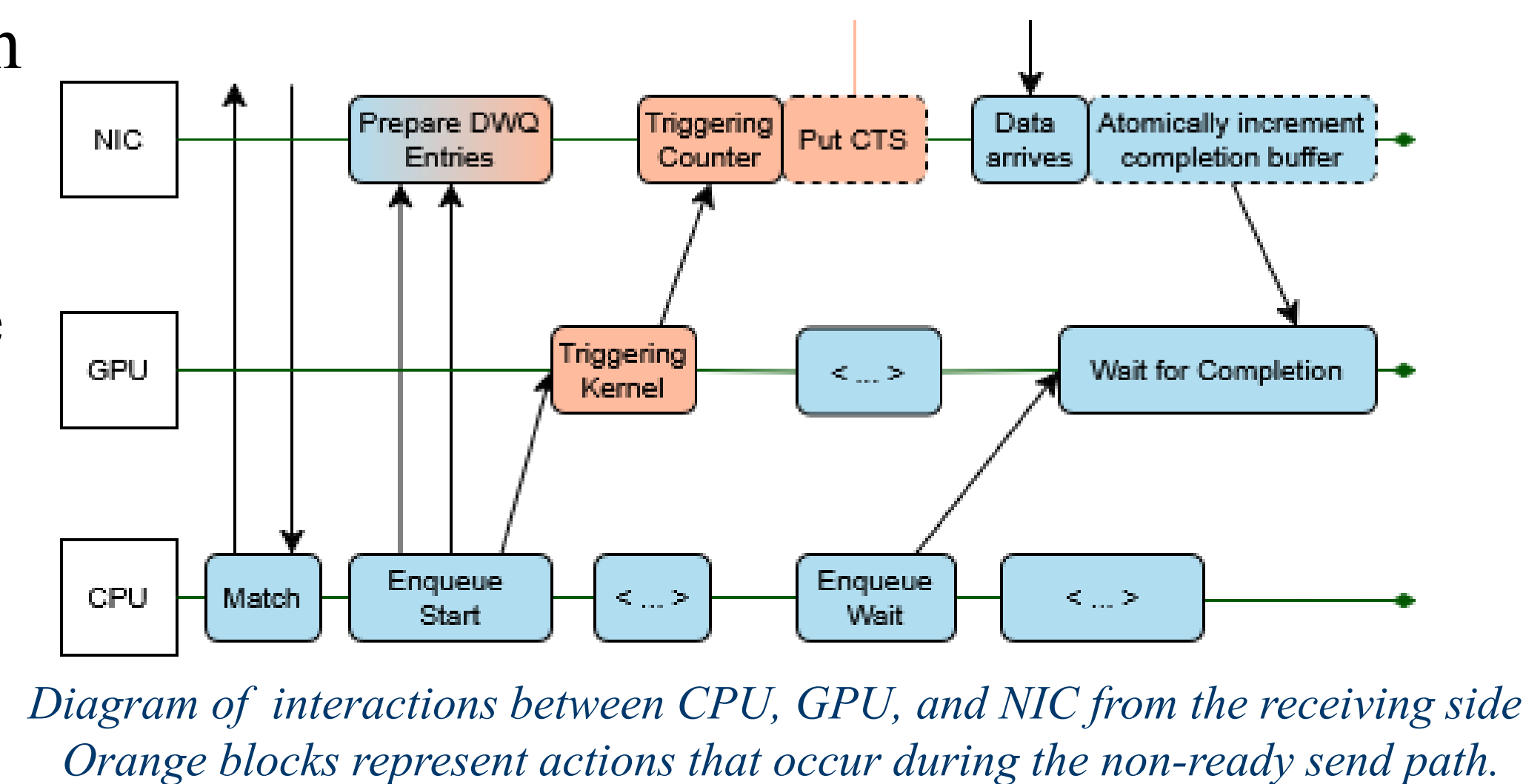
Prior work shows a stream triggering interface can improve performance, but we wanted to examine the tradeoffs and better understand the hardware. Our library is available publicly via MPI Advance (with a Spack package) and offers three different back-end implementations:

- A simple, progress thread version mostly used for testing. Requires more synchronization from the user.
- A HIP and CUDA memops version, using a CPU thread to coordinate communication
- A CPU-free backend using the CXI libfabric provider and its extensions, based on the approach used by HPE in their whitepaper [1]

Our design for each direction of the point-to-point operations are shown in the figures on the right.

Both the ready and non-ready variants are shown.

[1] N. Namashivayam et. al, "Exploring Fully Offloaded GPU Stream-Aware Message Passing," Jun. 2023, arXiv:2306.15773 [cs].



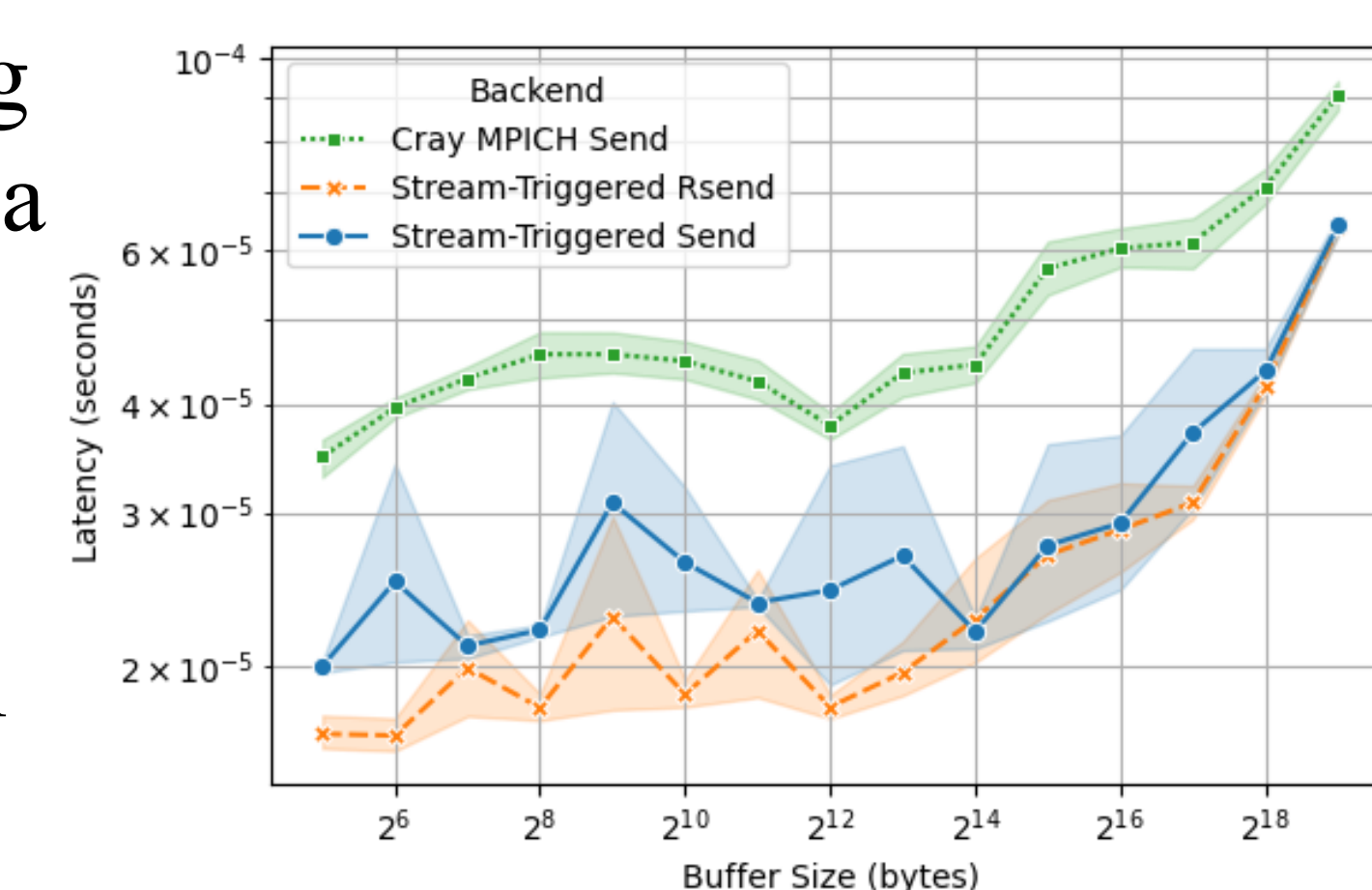
Examining the Performance of Stream Triggering

To test if there are potential benefits to be gained from integrating stream triggering communication into an application, we created a "packing Ping-Pong" test that cyclically runs over a range of message sizes:

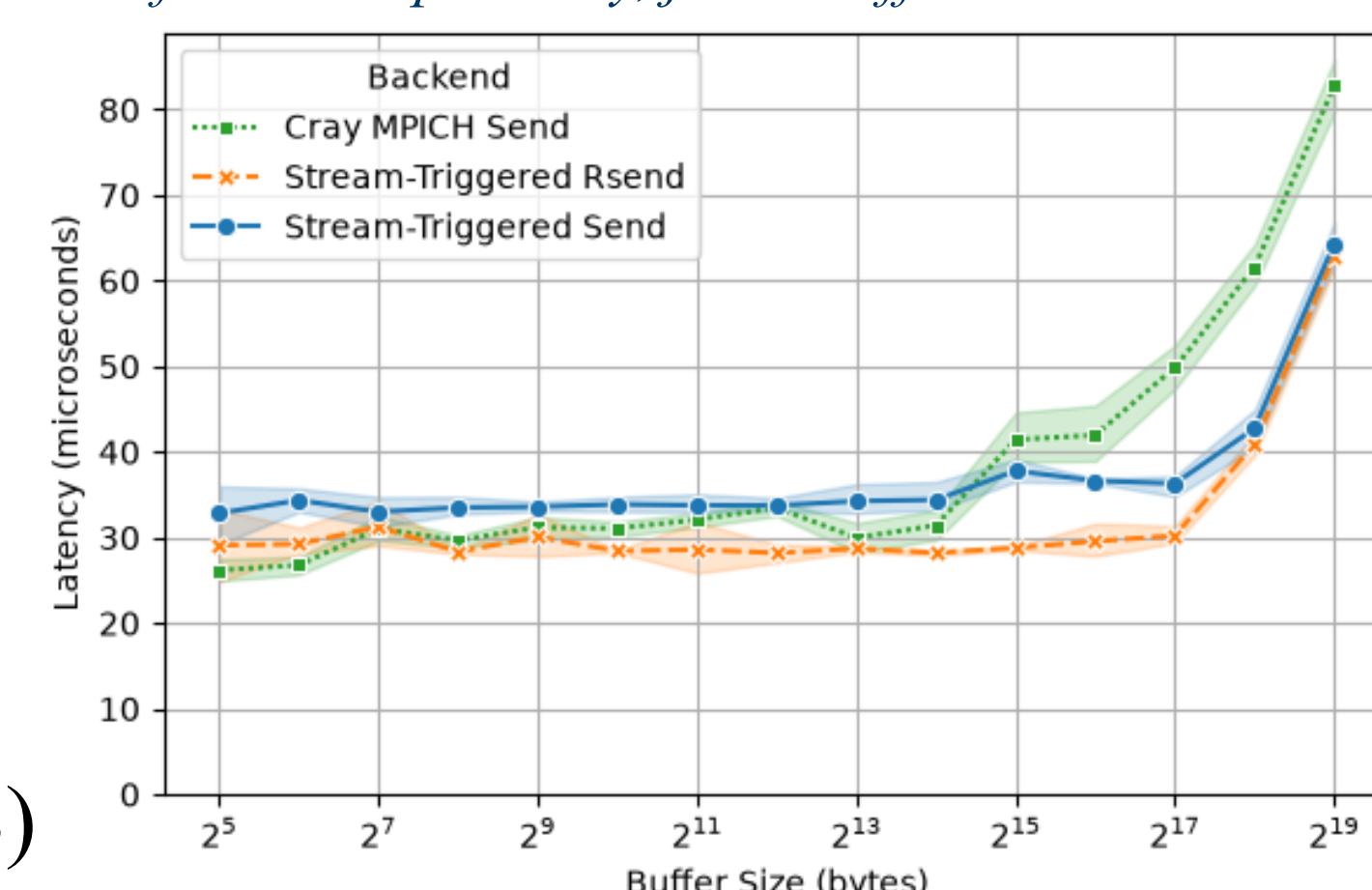
- Launches a GPU kernel to pack the send buffer
- Sends the data to its partner (ping), who launches a GPU kernel to "compute" (unpack) the data
- Sends the new data back to the original process (pong)

This benchmark represents the "best use-case" for stream-triggering (specific application results are discussed more on Evan Sugg's and Nicole Aven's Posters)

- Original runs (top), saw a considerable (12 - 49%) latency improvement compared to Cray MPICH
- Since then, the system software libraries have been upgraded (e.g., cray-mpich 9+, libfabric 2.3+)
- New runs (bottom) show a thinner gap. Here, we saw parity at smaller messages that grew into modest gains (~25% at ≥ 32 KiB)



Comparison between two versions of the stream-triggering implementation as the software stack improved (and our benchmark fixed). The y-axis are the same measurements of round-trip latency, just in different units.



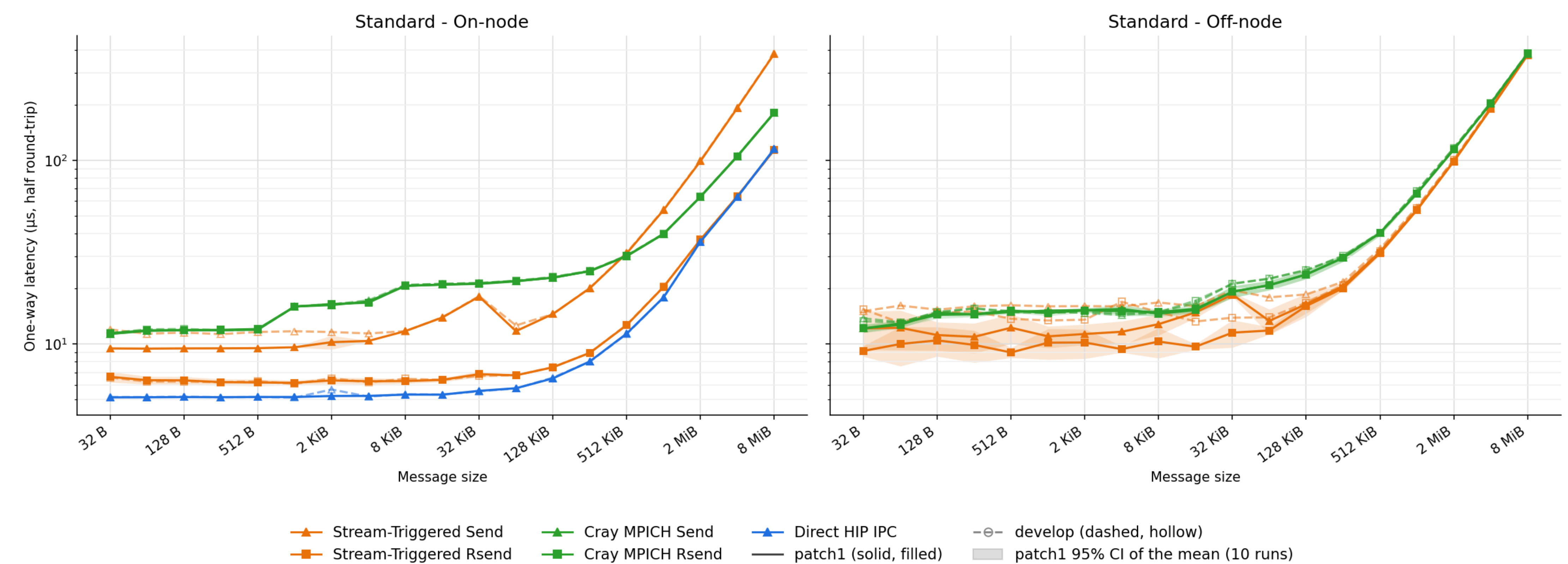
Optimizing Communication Protocol Performance

Our previous poster noted that there was room for improvement in some of our communication approaches. Since that poster, we have implemented the following optimizations:

- HIP-IPC support for on-node ready-send operations
- A credit-based protocol for eager regular send operations
- Fewer kernel launches: single kernel for per batch of start triggers, one per wait batch

We then re-ran the benchmark (on Tuolumne) and got a clearer picture of the "best-case" stream-triggering performance. From the chart we can see the following results:

- On-node performance improved for regular sends and considerably improved for ready sends.
- Compared to the results from the last meeting (top plot, bottom-left section), on-node ready-sends are the most improved, and now almost match a pure IPC-baseline.
- Off-node performance, while still close to the MPI-baseline, has a much cleaner separation in the middle message size range.



One-way latency results from running the "Packing Ping-Pong" experiment on Tuolumne with new optimizations and protocols. Lower is better. Dashed lines represent values from older runs before all of the optimizations were implemented.

Future Work

- Explore stream-triggered collectives and their trade-offs
- Explore an on-node HIP IPC approach for regular sends
- Standardization with the MPI Forum (ongoing)
- Implementation on non-HPE networks

arXiv Paper (Cluster 2026 paper out soon)



Acknowledgements

This work was performed with partial support from the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP III & IV), Awards DE-NA0003966 and DE-NA0004267. We'd like to thank Trey White (ORNL), Jack Lange (ORNL), Whit Schonbein (Sandia), Matthew Dosanjh (Sandia), and Naveen Ravi (HPE) for all their insights on our interface and debugging the CXI libfabric library.



Accelerated Data Simulations using Stream-Triggered MPI+Kokkos Comm

C. Nicole Avans, Evan Drake Suggs, Derek Schafer, and Anthony Skjellum | Tennessee Technological University

Introduction

- Stream-triggered communication enables operations to be initiated by device streams, **without needing CPU orchestration**. This is an effective strategy for achieving overlap between communication and computation.
- MPI does not currently standardize an approach, so multiple disparate designs and implementations compose the design space.
- This work documents the addition of a prototype stream-triggered communication interface for Kokkos Comm using MPI Advance.

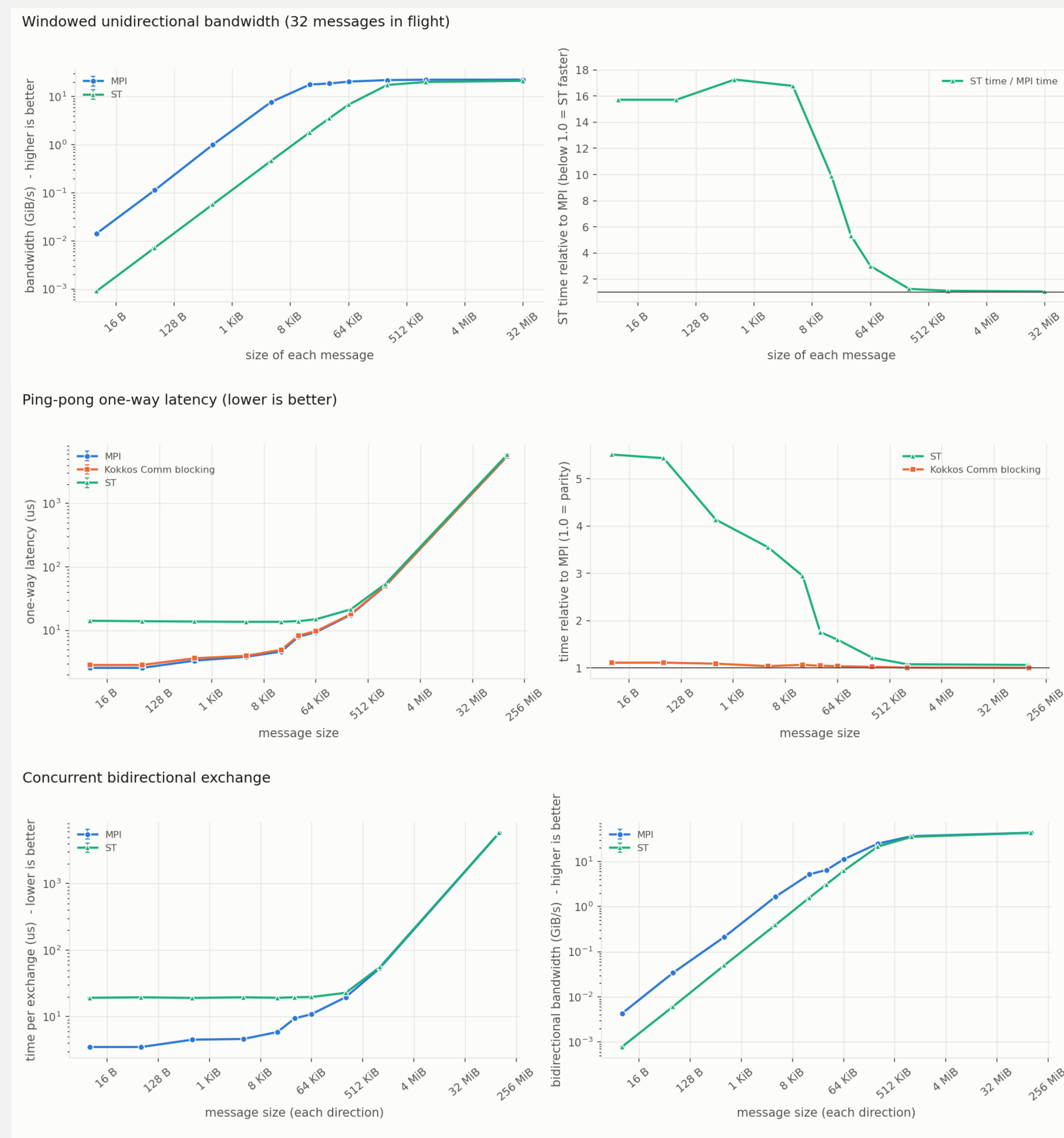


Fig. 1

Fig. 2

Fig. 3

Methodology

- Use Kokkos Comm as test platform for integration with MPI Advance building on previous work for Cabana.
- Investigate stream-triggered communication performance.

Evaluation

- Experiments conducted on LLNL's Tuolumne cluster, a sibling system to El Capitan. (Slingshot-11 interconnect with CXI libfabric interface and AMD MI300A APUs).
- Bandwidth (Fig. 1) and latency (Fig. 2) considered, in addition to concurrent exchange modelling halo exchange (Fig. 3) and pipelined communication (Fig. 4).

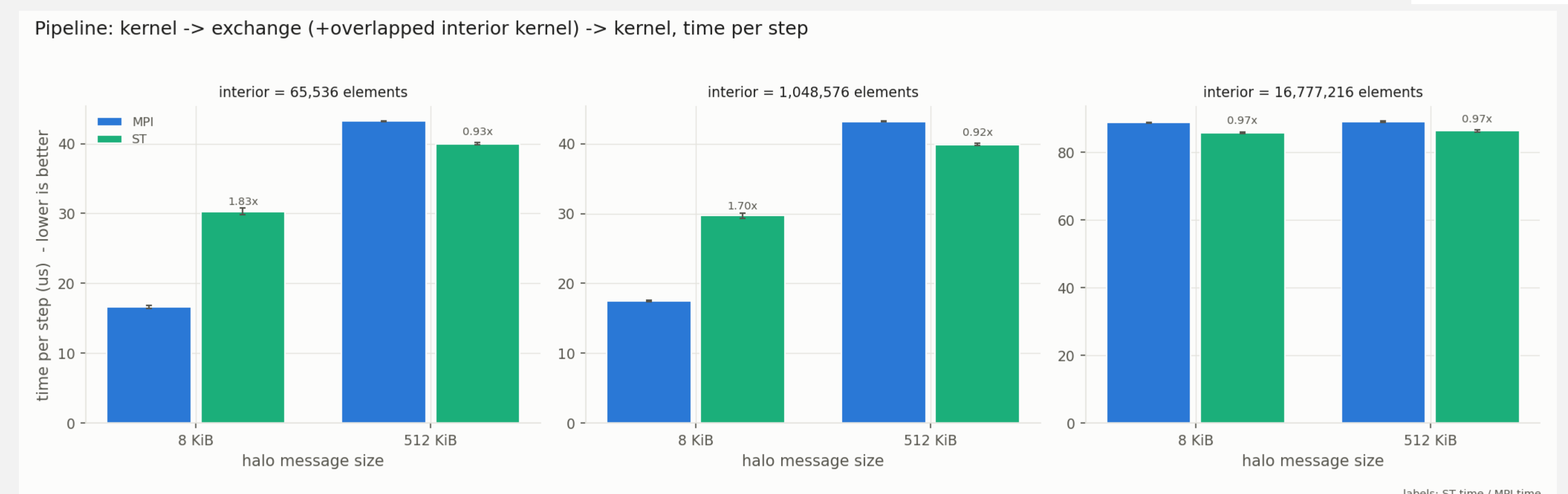
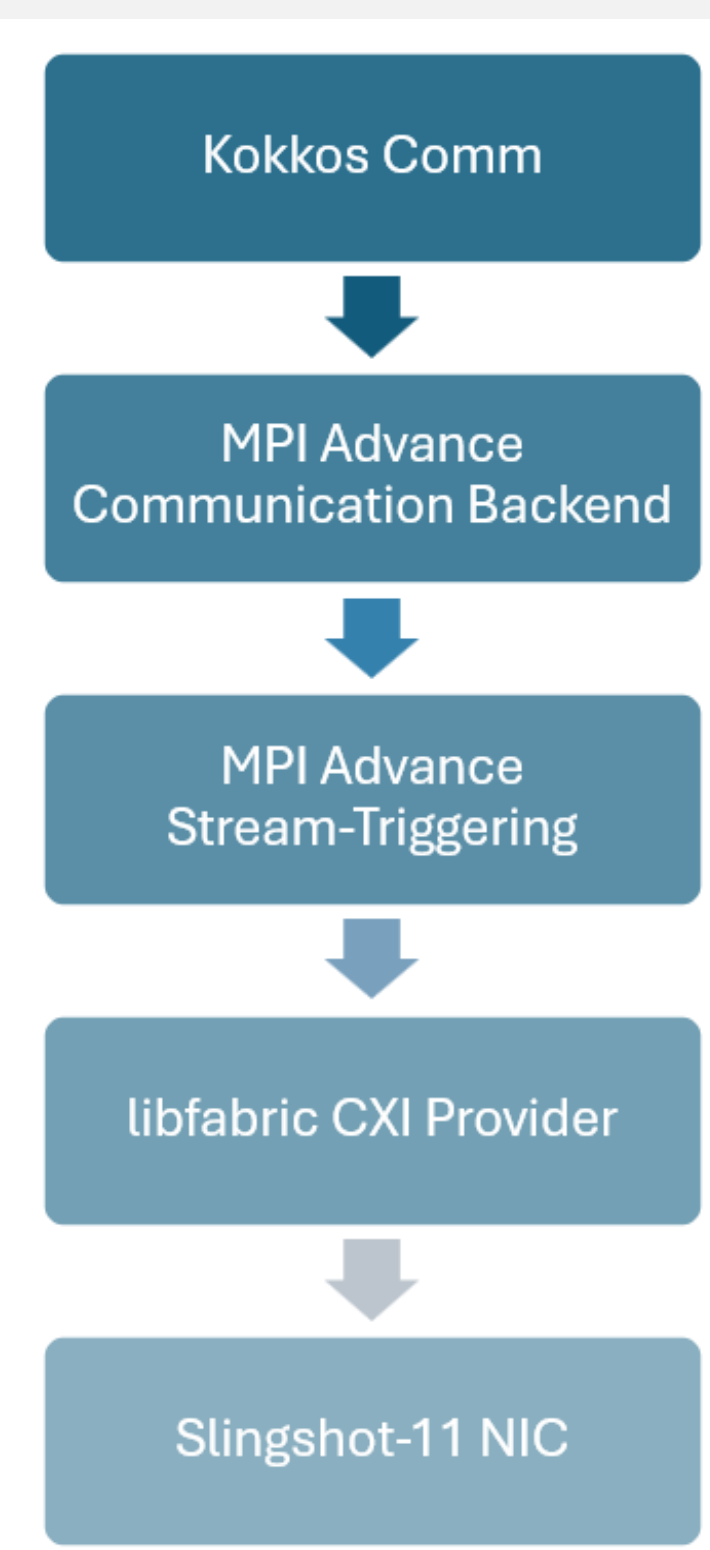


Fig. 4

Conclusions and Future Work

- Investigation of performance optimizations and advanced functionality support such as non-contiguous data handling, ready-sends (where the send blocks only long enough to send, and the matching receive function must be ready), and switching between coarse vs. fine-grain memory coherence (memory only being coherent after synchronization versus during kernel run-time).
- Implementation of additional collectives in Kokkos Comm stream-triggering
- Support for kernel-triggered communication prioritized.
- Collaboration with the MPI Forum on standardization.

QR Code



Acknowledgements

This work was performed with partial support from NSF Grants 2514054, 2450093, 2405142 and 2412182 and from the U.S. Department of Energy's Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP III & IV), Awards DE-NA0003966 and DE-NA0004267.

We'd like to thank Trey White (ORNL), Jack Lange (ORNL), Whit Schonbein (Sandia), Matthew Donsanjh (Sandia), and Naveen Ravi (HPE) for all their insights on our interface and debugging the CXI libfabric library.

References

- [1] S. Slattery, S. T. Reeve, C. Junghans, D. Lebrun-Grandi'e, R. Bird, G. Chen, S. Fogerty, Y. Qiu, S. Schulz, A. Scheinberg, A. Isner, K. Chong, S. Moore, T. Germann, J. Belak, and S. Mniszewski, "Cabana: A Performance Portable Library for Particle-Based Simulations," Journal of Open Source Software, vol. 7, no. 72, p. 4115, Apr. 2022. [Online]. Available: <https://joss.theoj.org/papers/10.21105/joss.04115>