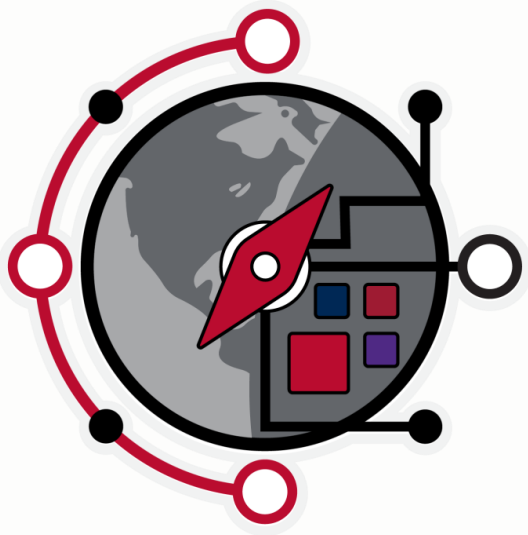




Center for
Optimized Modern
Parallel Adaptive
System Software

Communication Optimization

Prof. Anthony Skjellum · Tennessee Technological University
COMPASS Annual Review Meeting · October 29, 2026

Communication Optimization: overview

- Through the lens of the student presentations; roadmap items keyed on each poster's first slide
- When GPU transfer-mode wins: GPU-enhanced communication
 - MPI Advance in brief [Suggs; Amanda Bienz covers more later]
 - CPU-free stream triggering: interface, Tuolumne latency, packing ping-pong [Schafer]
 - Cabana halo exchange and CabanaGhost strong scaling [Suggs]; stream-triggered Kokkos Comm [Avans]
 - Stream triggering in applications: Cabana, aCG; kernel-initiated transfers with OpenRAPIDS Cascade [Hines]
- When CPU-transfer mode is better: CPU-enhanced communication
 - ExaMPI strong-progress RMA: what, design, why it matters [Martin, Humphrey]
 - Earlybird collectives on MPI RMA: speedup, delay absorption, against system MPI [Wesley]
 - Locality discovery: NUMA regions from ping-pong timings [Kinhead]
 - Topology-aware collectives: message mode, RMA mode, RAPIDS hot loops
- Common themes across the student work
- Low-level access, next steps: CXI and the limits of libfabric; TORRENTs and bespoke kernels
- Conclusions; Q&A

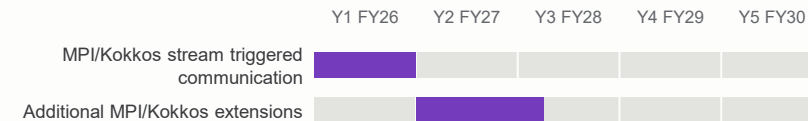
SECTION 01

When GPU transfer-mode wins

Stream triggering and kernel-initiated transfers: the GPU drives the communication; the host is out of the hot loop.



MPI Advance



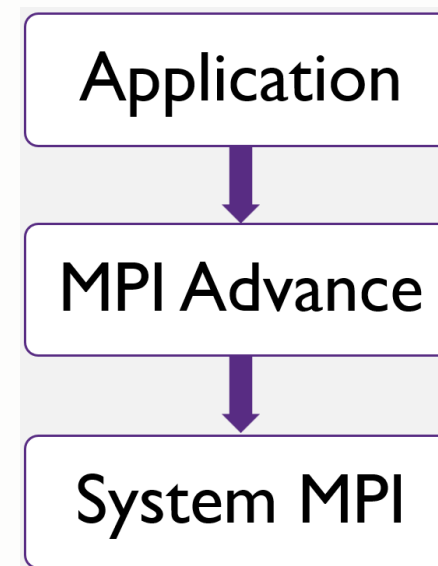
What it is

- Performance optimizations and experimental features: performance control, non-standard interfaces, standard-but-unimplemented ones
- A paradigm for new MPI interfaces without learning an MPI implementation
- Proofs of concept for the MPI community [and the MPI Forum]

Using it

- Link one more library. Prefixes MPIP_, MPIL_, MPIS_, MPJA_ stay clear of MPI_ and MPIX_
- A meta-project builds some or all libraries in one step; shared boilerplate [requests, management] lives in one place

MPIPCL	Locality-aware collectives	Stream triggering
MPI 4.0 partitioned communication	Collective algorithms for irregular communication [Amanda Bienz, later today]	MPI-style GPU-stream communication; CPU-free on Slingshot



Evan Drake Suggs, Derek Schafer, C. Nicole Avans, Amanda Bienz, Anthony Skjellum [TTU, UNM]. SC26 poster.

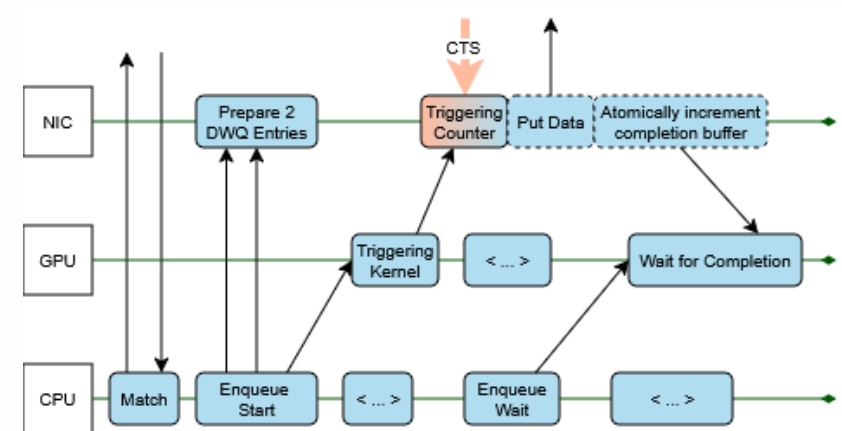
CPU-free stream triggering

Three back ends in MPI Advance

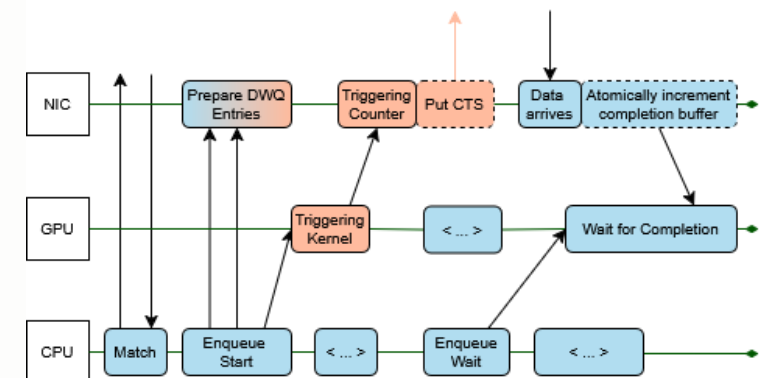
- Progress-thread version, mostly for testing; requires explicit GPU synchronization from the user
- HIP and CUDA memops version; a CPU thread coordinates communication
- CPU-free back end on the CXI libfabric provider and its extensions, after HPE's whitepaper

Optimizations since the last meeting

- HIP-IPC support for on-node ready sends
- Credit-based protocol for eager, regular sends
- Fewer kernel launches: one kernel per batch of start triggers, one per wait batch



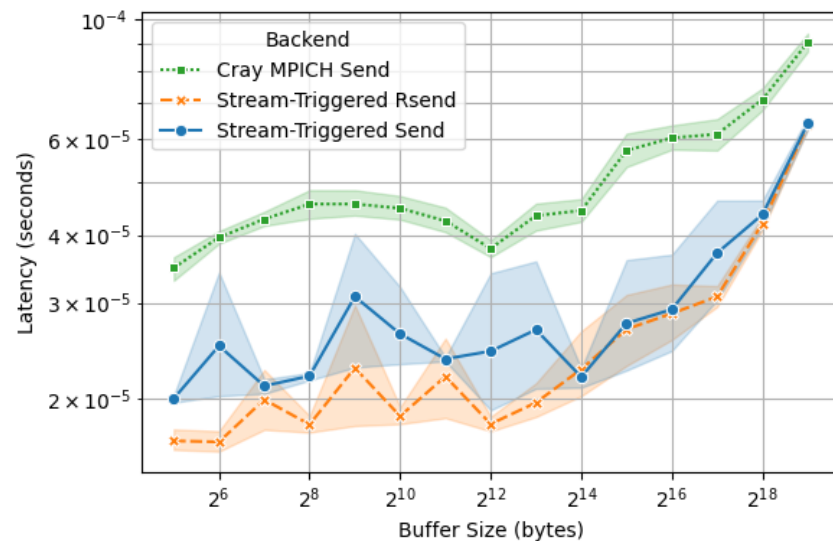
Sending side: CPU, GPU, NIC. Orange: the non-ready send path



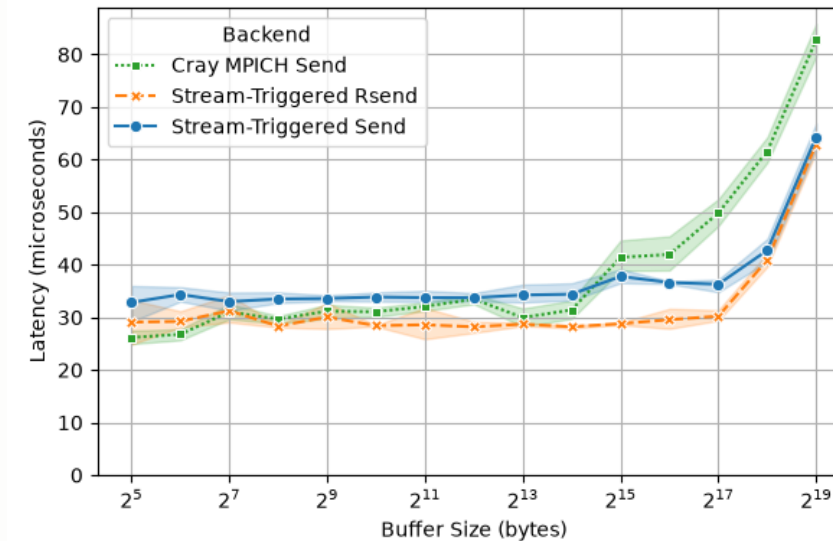
Receiving side. Derek Schafer and Patrick Bridges [UNM]

Packing ping-pong: ideal use case, before and after

Original runs



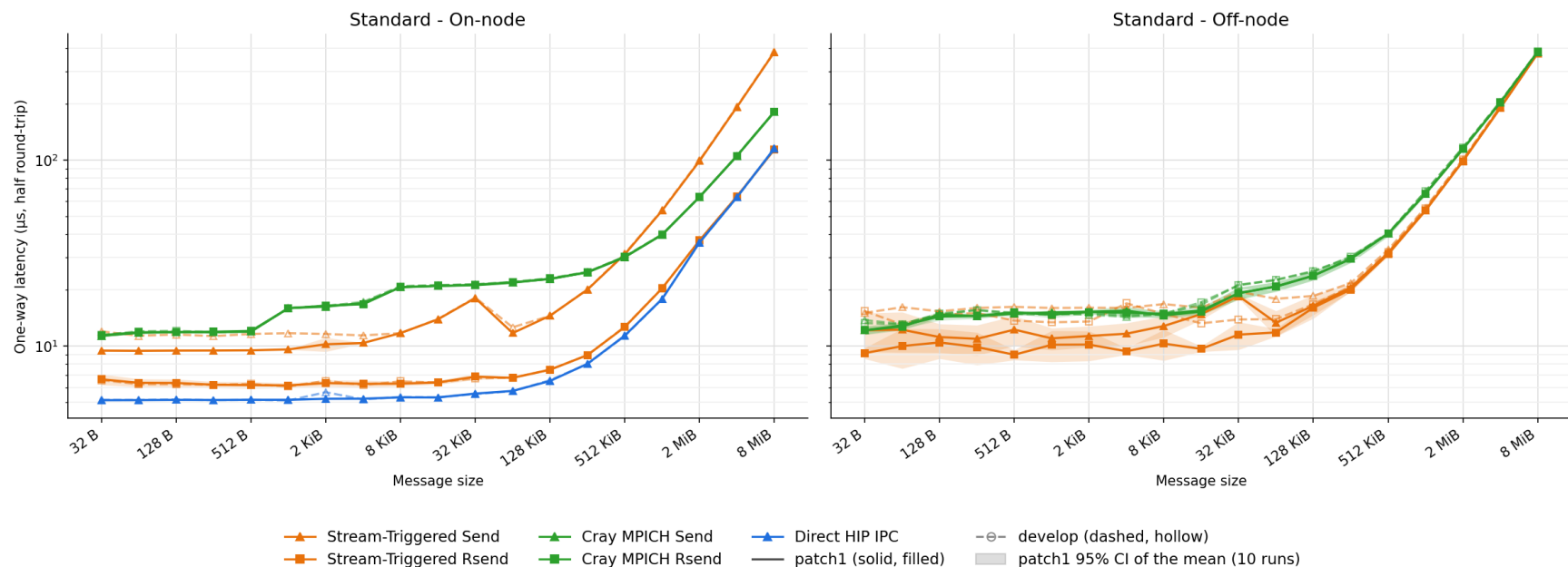
After optimizations and the stack upgrade



- Benchmark: pack on the GPU, send [ping], partner unpacks on the GPU, send back [pong]; a cycle over message sizes
- Original runs: 12–49% lower latency than Cray MPICH
- Since then: Cray MPICH 9+, libfabric 2.3.1, ROCm 7+. New runs: parity at small messages, about 25% gain at 32 KiB and above
- See Derek Schafer's Poster for more implementation details
- Next: stream-triggered collectives; MPI Forum standardization; non-HPE networks

Stream triggering on Tuolumne: best-case latency

Tuolumne ping-pong latency (up to 8 MiB) - means, patch1 (10 runs) vs develop (5 runs)



Takeaways

- On-node: regular sends improved; ready sends improved considerably, now close to a pure IPC baseline
- Off-node: close to the MPI baseline, with a cleaner separation in the middle sizes
- Ready send is the fastest path on and off node

Packing Ping-pong one-way latency on Tuolumne, up to 8 MiB, mean over several runs across multiple communication interfaces
Dotted lines represent code that was used for Cluster paper; solid lines represent latest improvements
Send variants for Cray-MPICH are overlapped on chart due to essentially the same times

Cabana halo exchange, CPU-free

Design

- Cabana Grid Halo [1] with MPI Advance stream triggering as a back end; CabanaGhost is the benchmark
- Stream-triggered Scatter and Gather [enqueueScatter, enqueueGather], with regular send and ready-send options; Cray MPICH as the default
- Start the receives, pack the send buffers, unpack after the receives complete. No fences [the default Cabana path has them]
- See Evan Sugg's poster for more application integration details

Why it matters

- GPU codes bottleneck on CPU-GPU coordination; MPI Advance queues the GPU operations instead
- A sample case: the stream-triggering API is performant and fits an existing performance-portable library

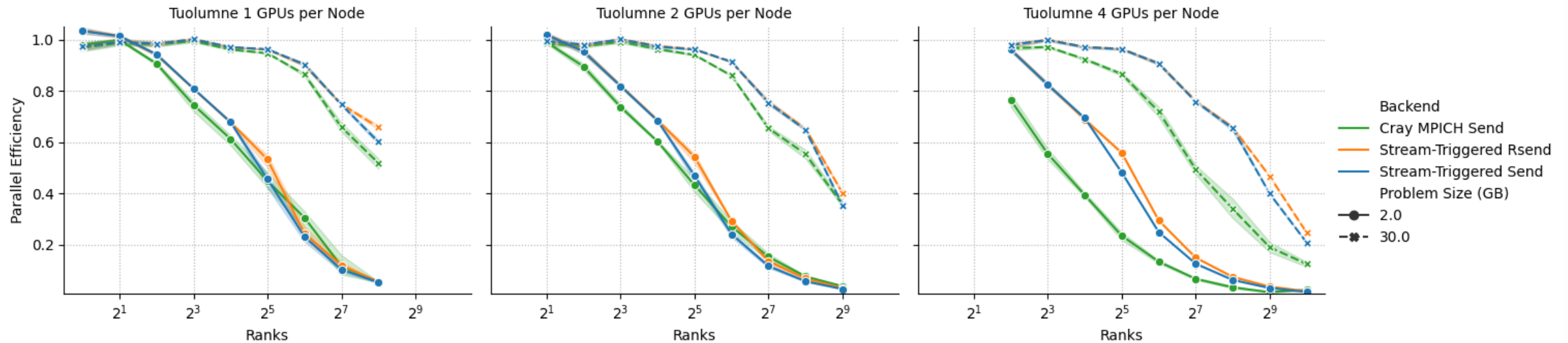
```
template <class... ArrayTypes>
void enqueueGather(
    const ArrayTypes&... arrays)
{
    // Enqueue starting the receives
    MPI_Enqueue_startall( _my_queue,
        num_n, &_requests[0] );
    // Enqueue packing the data to send
    enqueuePackBuffers( _owned_buffers,
        _owned_steering,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

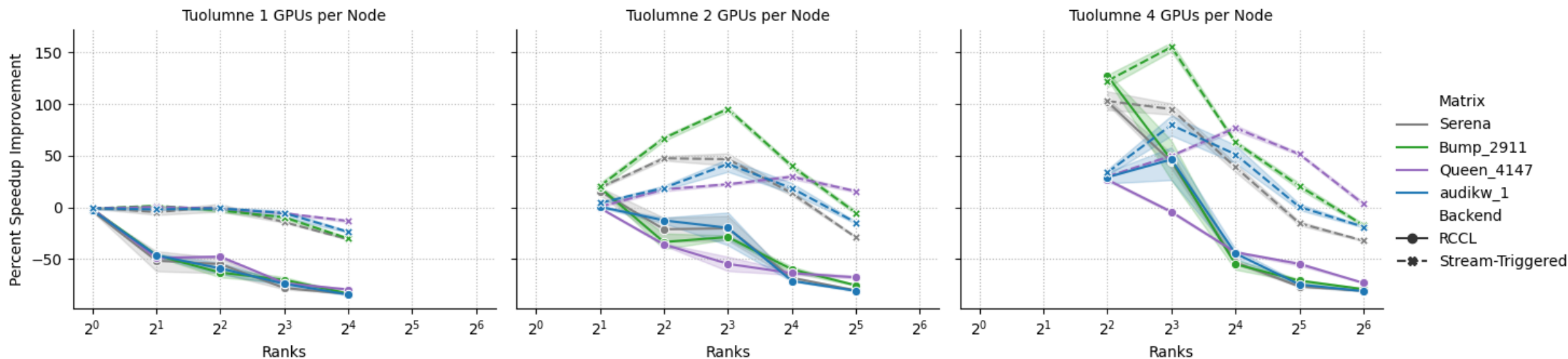
enqueueGather on the MPI Advance queue. Evan Drake Suggs, Derek Schafer, Anthony Skjellum [TTU]; paper at Cluster 2026 [arXiv]

CabanaGhost strong scaling on Tuolumne



- Halo-exchange strong-scaling test, 2 GB and 30 GB problems; parallel efficiency vs ranks, split by GPUs per node
- Four GPUs per node: stream triggering faster than Cray MPICH at every rank count. Cray MPICH is always slower at four per node for a fixed GPU count
- One or two GPUs per node: Cray MPICH wins at the largest scales [Tuolumne-specific MPICH optimizations; not seen on Frontier]
- Most runs match or beat Cray MPICH. Open issues: the limited number of resource handles; larger rank counts on four or more nodes
- Next: AllReduce and more collectives; Kokkos Comm paper; Trilinos integration

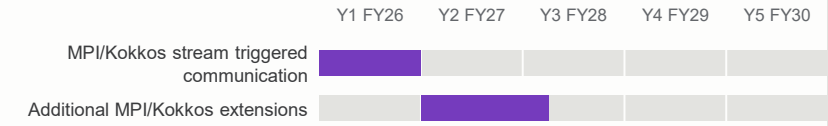
MPI Advance stream triggering in applications



aCG conjugate gradient on Tuolumne, four matrices: percent speedup over the MPI build, RCCL vs stream-triggered [poster Fig. 3; Trotter et al., SC '25]

- Accelerated CG from SC 25 paper that already featured a mode that supported using RCCL and/or GPU streams for communication
- Adding in our stream triggering support was fairly simple and provided a more elevated test case
- aCG: mixed. MPI ahead of stream triggering overall; stream triggering ahead of RCCL. Four GPUs per node is the favorable case
- Disabling some on-node Cray-MPICH optimizations helped bring stream triggering performance scale better than MPI longer

Stream-triggered Kokkos Comm

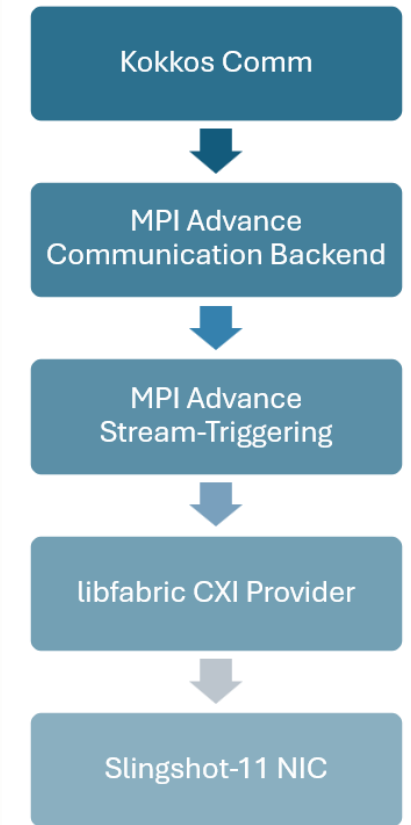


What it is

- A prototype stream-triggered communication interface for Kokkos Comm, built on MPI Advance
- Device streams initiate the operations; no CPU orchestration. The route to overlap of communication and computation
- MPI standardizes no such interface yet; several designs and implementations fill the space
- See Nicole Avans's Poster for more details

Evaluation on Tuolumne

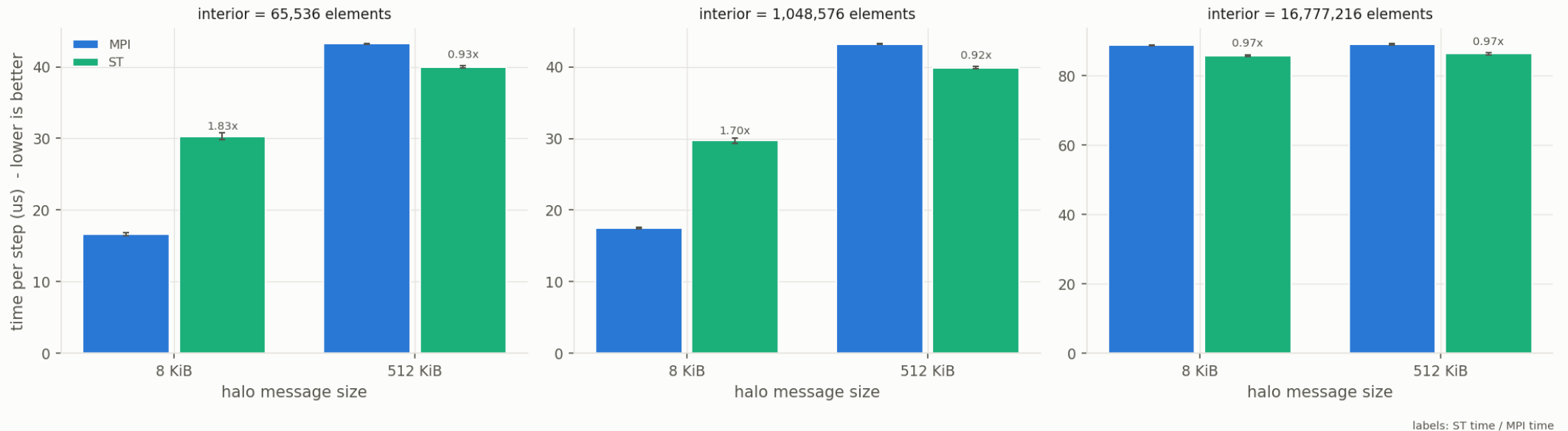
- Slingshot-11 with the CXI libfabric provider; AMD MI300A APUs
- Bandwidth, latency, concurrent bidirectional exchange [halo model], and a pipelined kernel–exchange–kernel loop



C. Nicole Avans, Evan Drake Suggs, Derek Schafer, Anthony Skjellum [TTU]

Kokkos Comm stream triggering: where it adds performance

Pipeline: kernel -> exchange (+overlapped interior kernel) -> kernel, time per step



- Pipeline: kernel, exchange with an overlapped interior kernel, kernel. Time per step, stream-triggered vs MPI [labels: ST time / MPI time]
- Large halos or large interiors: stream triggering wins, 0.92–0.97× of the MPI time. Small halo with a small interior: a fixed cost dominates [1.7–1.8×]
- Microbenchmarks show the same floor: about 13 μ s per stream-triggered operation; parity with MPI from 512 KiB [latency, bandwidth, bidirectional exchange]
- Next: non-contiguous data, ready sends, coarse vs fine-grain coherence; collectives; kernel-triggered communication; MPI Forum standardization

Cascade: kernel-initiated transfers

Lower middleware [*CCLs, MPI implementations] needs a few data-transfer operations, done fast.

OpenRAPIDS [Reduced API Data-transfer Specifications]: 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

Cascade: Thomas Hines's design and implementation, with Riley Shipley.
Target: Slingshot-11 with AMD GPUs; LLNL Tuolumne, MI300A.

Poster: OpenRAPIDS Cascade: Kernel-Initiated Data Transfers on Slingshot-11 [Hines, Skjellum, Shipley].

Why Cascade

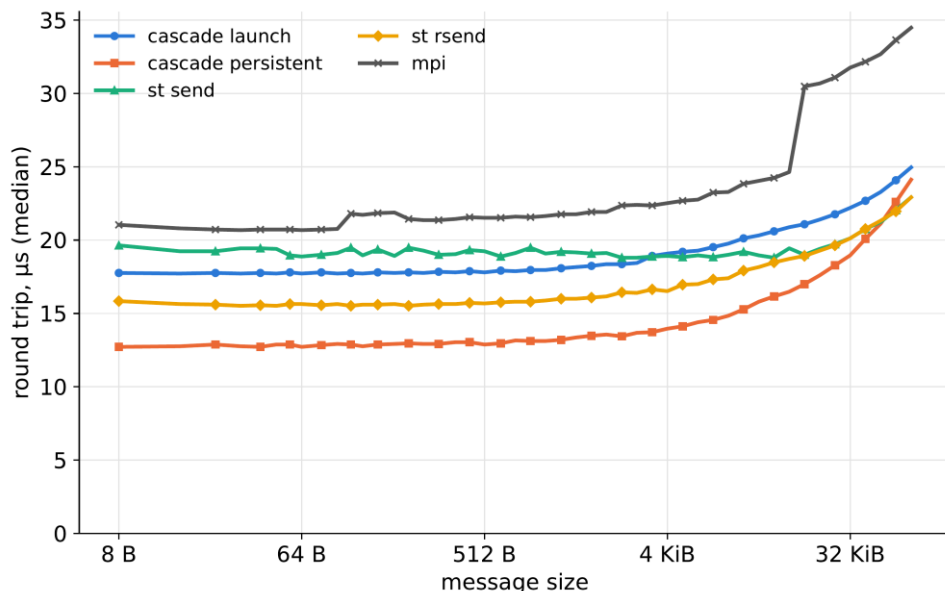
- Slingshot-11, AMD GPUs: existing paths have the host build the network work in advance
- GPU computes, host communicates: exchanges cross GPU and host
- Cascade removes the crossing. The kernel that produced the data sends it

How Cascade works

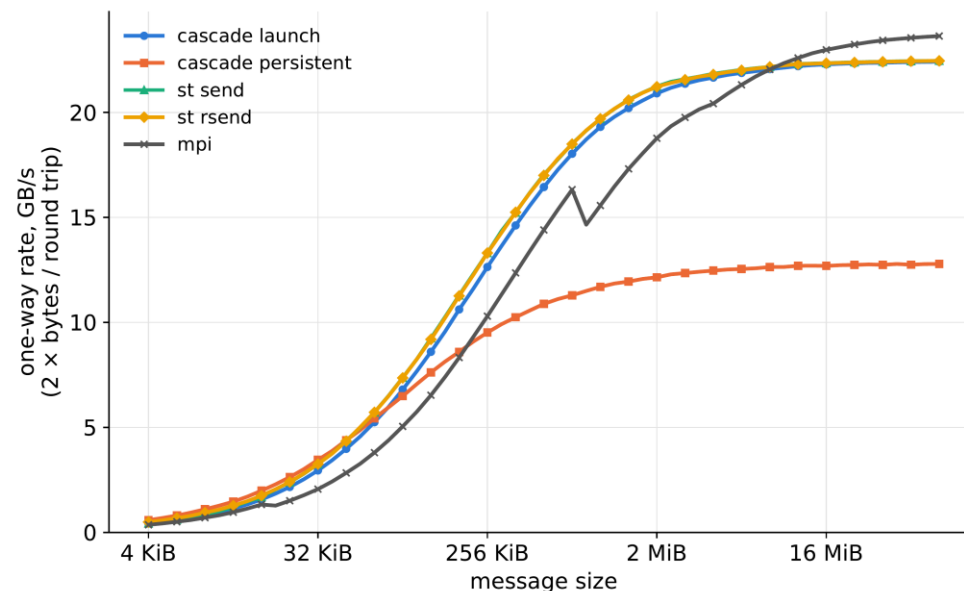
- A running kernel writes its own NIC commands into a Cassini command queue through libcxr [no host on the transfer path]
- Commands prebuilt once. A post patches address, length, slot; one 128-byte store
- Flow-controlled slots: delivery frees the source, a credit frees the slot
- Slots held across kernels: communication overlaps computation
- One node: the same kernel code copies GPU to GPU

Cascade on Tuolumne: latency and bandwidth, two nodes

Latency between two nodes



Bandwidth between two nodes



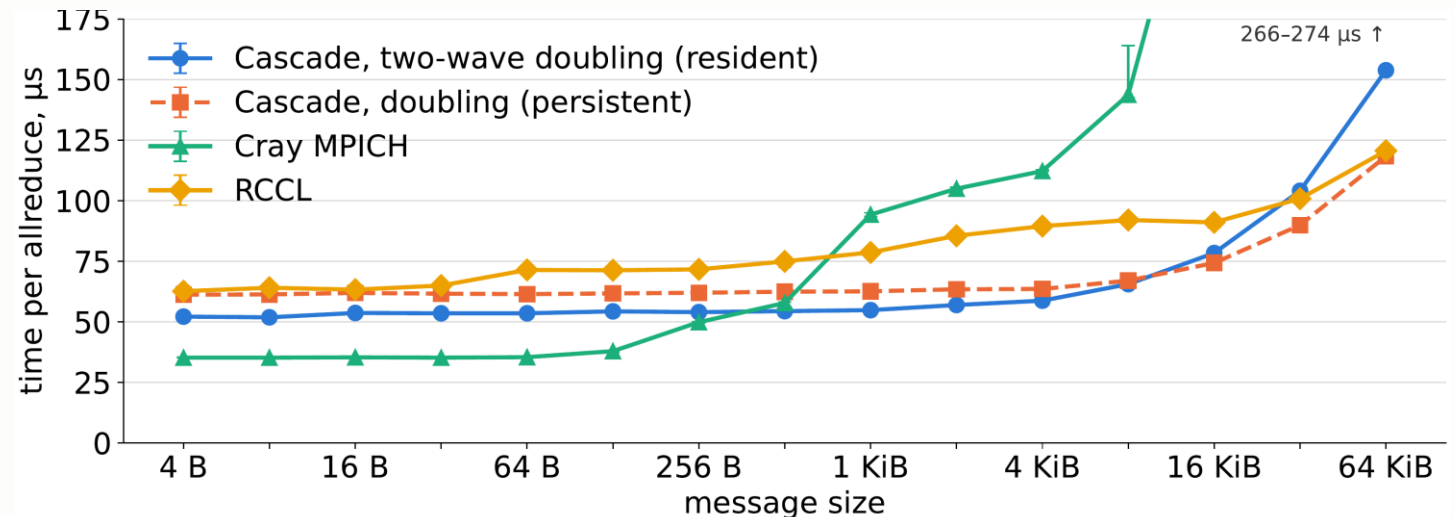
- 8 B round trip: Cascade persistent 12.7 μ s; Cray MPICH 19.8–24.6 μ s. 98% of persistent round trips within 12.6–13.0 μ s
- Bandwidth: Cascade launch tracks stream-triggered MPI to the Cray MPICH asymptote. Persistent levels off near 12.8 GB/s [its single block also does the add]
- Rally: GPU ping-pong; each side adds to the vector before returning it. cascade launch: a kernel per turn. persistent: one resident kernel. st: MPI Advance stream triggering. mpi: Cray MPICH 9.1.0, host waits each turn. Tuolumne, MI300A, Slingshot-11; median of 3 runs

Allreduce at 128 GPUs

128

GPUs on Tuolumne
[32 nodes × 4 MI300A]

- Cascade beats RCCL and Cray MPICH from 256 B through 32 KiB [each in its best configuration]
- Fastest of all four, 512 B through 8 KiB. Cray MPICH leads at 256 B and below
- f32 sum; a compute kernel between calls; slowest rank over 10,000 calls; median of 3 runs
- Current work: OpenCCL on Slingshot-11 calls Cascade's kernel-side verbs directly



Allreduce (f32 sum) on Tuolumne: 128 ranks, 32 nodes × 4 MI300A, a compute kernel between calls.
Slowest rank over 10,000 calls; median of 3 runs, whiskers min to max.

SECTION 02

When CPU-transfer mode is better

Strong progress and Earlybird collectives: communication that moves while the application computes.



ExaMPI: strong-progress RMA

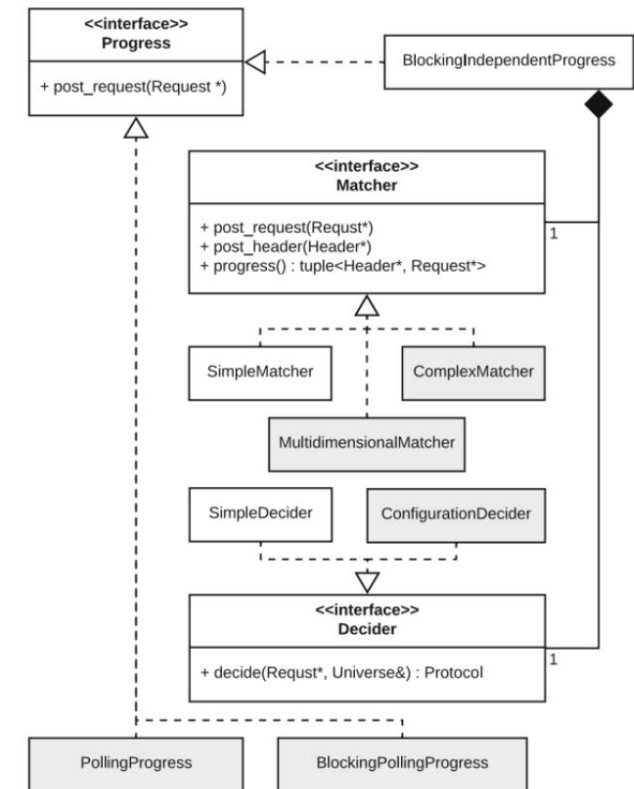
What ExaMPI is

- C++ MPI-subset implementation; a collaboration of universities across current and prior PSAAP centers
- Testbed for experimental MPI features

Strong progress model

- Progress engine follows Dimitrov's strong progress model [2001]: communication progresses in the background; the user thread never progresses it explicitly
- Three core parts. Transport moves data. Algorithm sets Transport's parameters. Matcher pairs posted receives with incoming messages [unused for RMA]
- MPI RMA is often associated with bad performance. Strong progress can make it better

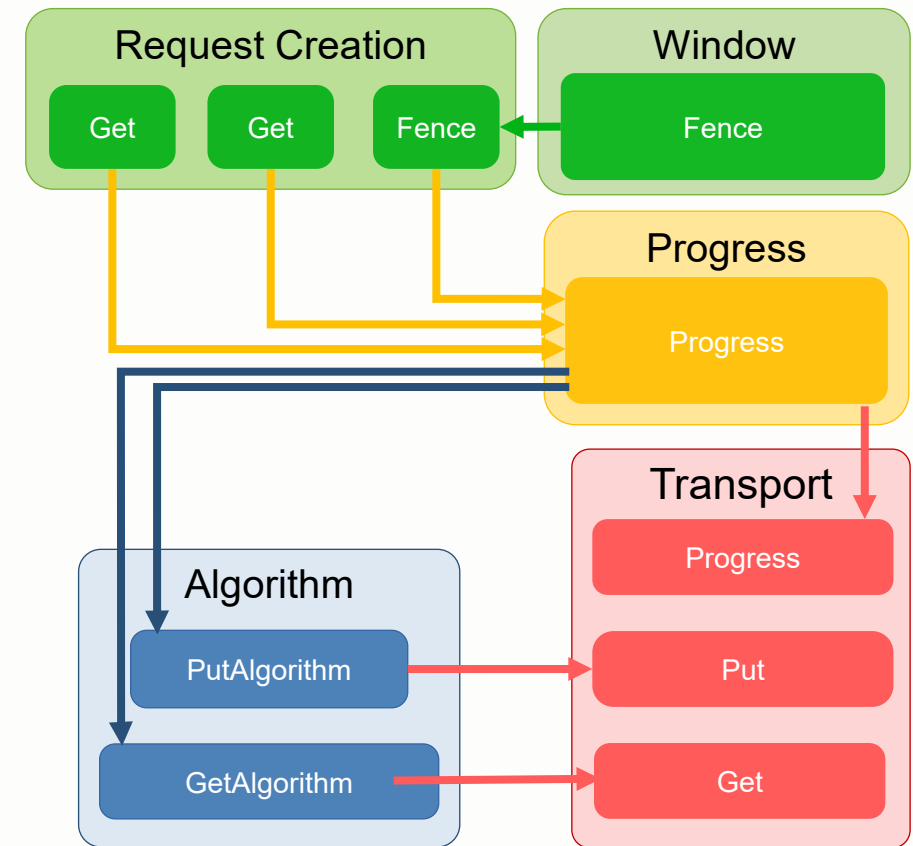
Javontae Martin and Kayla Humphrey, with Riley Shipley. Advice and review: Derek Schafer, Thomas Hines.



Structure of the progress layer of ExaMPI [Skjellum et al., 2020]

RMA in ExaMPI: implementation design

- Libfabric: the networking stack under RMA today
- Everything hangs on the Request object. It carries what each progress function needs
- Progress engine: a fixed loop over the Algorithm, Matcher, and Transport progress functions. Each reads the Request and acts
- Matching unused: unnecessary for RMA, necessary for message passing
- Put and Get each get an Algorithm; Transport carries Put, Get, and Progress
- Works because MPI_Win_fence allows early completion of operations within each window epoch [MPI 3.1]



Flow of an RMA operation in ExaMPI [poster Figure 3]

Why ExaMPI: strong progress, and what it enables

The gap

- Production MPIs still lack strong progress
- Overlap of communication and computation needs it; without it, the application thread does the MPI's work
- RMA shows the cost most clearly [the poster's target]

The plan

- Apply strong progress to RMA; then to MPI
Advance's topology-aware collectives
- Show the impact of strong progress on COMPASS workloads
- Deliver into Trilinos later, once the importance is demonstrated
- Help MPI implementations get strong progress [out of scope for PSAAP; DOE/NNSA should be driving vendors to deliver better MPI]

Strong progress is the requirement for every project in this talk: the application computes while the MPI communicates.

Earlybird collectives on MPI RMA

The problem

- Collectives need every rank. One late rank [OS noise, load imbalance] and every other rank waits
- Hierarchical collectives: copy on-node, exchange between node leaders, scatter on-node. Fence-based versions synchronize between every phase
- Earlybird: move data as soon as it is ready. Early ranks' data is already off-node when the late rank arrives

The method

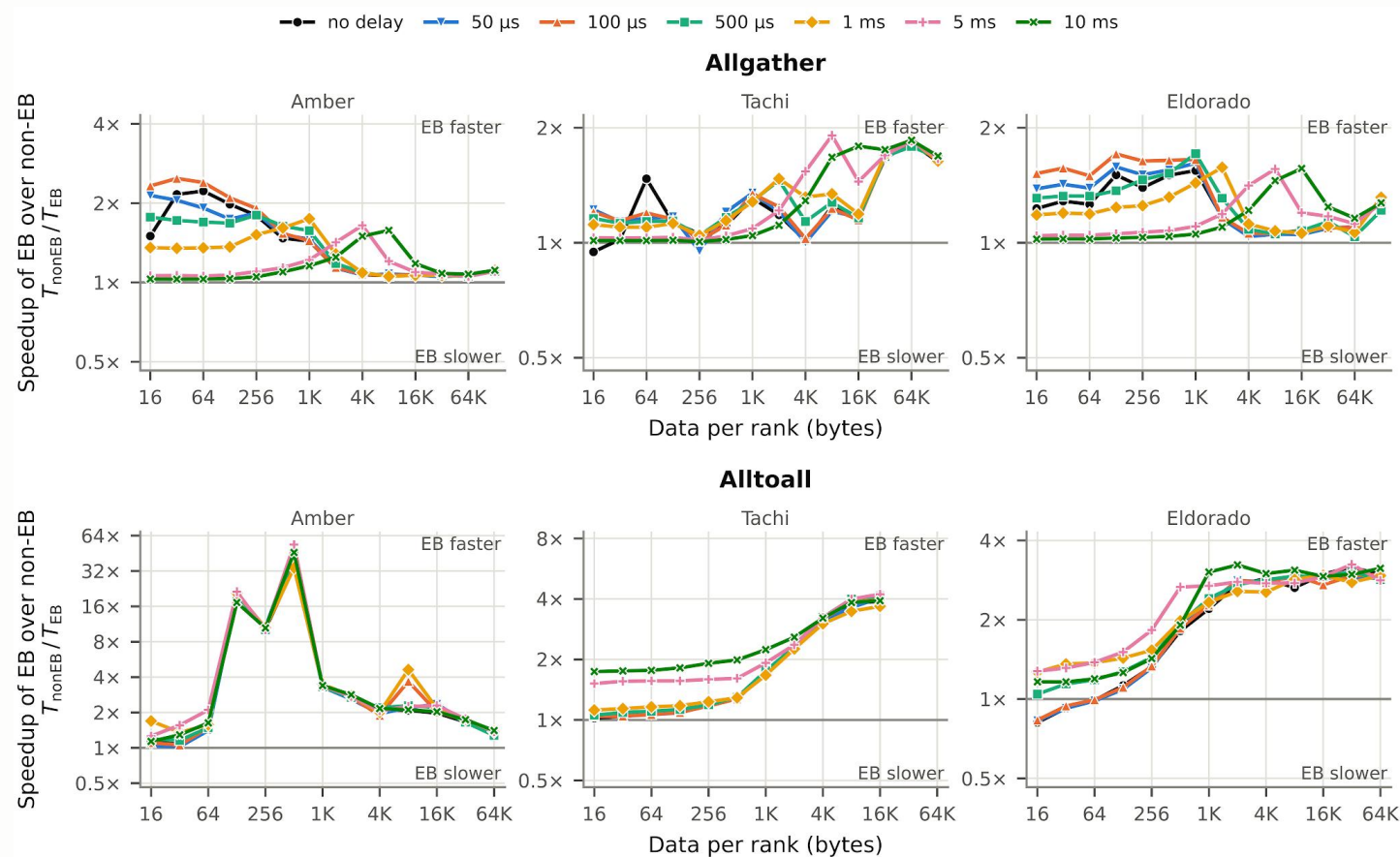
- Persistent Earlybird Allgather and Alltoall on MPI one-sided [RMA]
- Persistent API: init builds windows, communicators, flags once. Start/Wait per call
- Non-Earlybird: barrier and fence between phases
- Earlybird: each rank sets a generation flag; the leader MPI_Puts that rank's data as soon as the flag appears

System	CPU	Network	MPI
Amber	Xeon 8480+, 2 × 56 cores	Omni-Path, 100 Gb/s	Open MPI 4.1
Tachi	Xeon Max 9480, 2 × 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

Jackson Wesley [UNM], with Matthew Dosanjh and Whit Schonbein [Sandia] and Amanda Bienz [UNM].

Benchmark: 32 nodes × 64 ranks/node, 16 B to 64 KiB per rank; scaling to 64 nodes. Delay injection: one random non-leader sleeps before each collective.

Earlybird vs fence-based: speedup

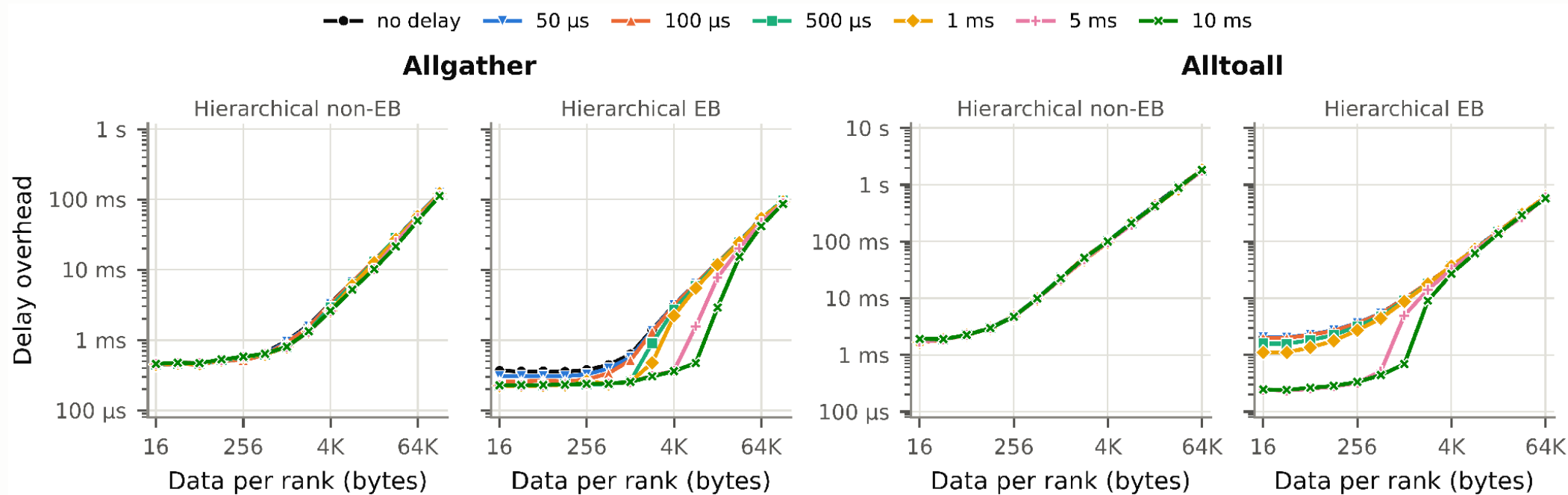


Takeaways

- Earlybird beats fence-based at almost every size, with or without delay
- Allgather: 1.2–2.6 $\times$ faster below 1 KiB
- Alltoall: 1.7–4 $\times$ faster from 1 KiB
- Setup is a one-time cost, recovered within a few calls in most cases

$T_{\text{nonEB}} / T_{\text{EB}}$, 32 nodes $\times$ 64 ranks/node, one line per injected delay [poster Fig. 4]

Absorbing delay, and performance vs. system MPI



	vs. non-Earlybird	vs. system MPI
Allgather	1.2–2.6× faster below 1 KiB	Faster at most sizes [6–20× at 1 KiB]; slower at ≤ 64 B on Cray
Alltoall	1.7–4× faster from 1 KiB	Faster at small to medium sizes only
P3DFFT	2.5–57.1% faster	Faster when per-destination messages are small

- Non-Earlybird pays the delay in full: every delay curve lies on the undelayed one
- Earlybird overhead falls as delay grows. Alltoall ≤ 64 B at 10 ms: 0.25 ms vs 2.0–2.3 ms undelayed
- Next: triggered operations; leader and data pooling

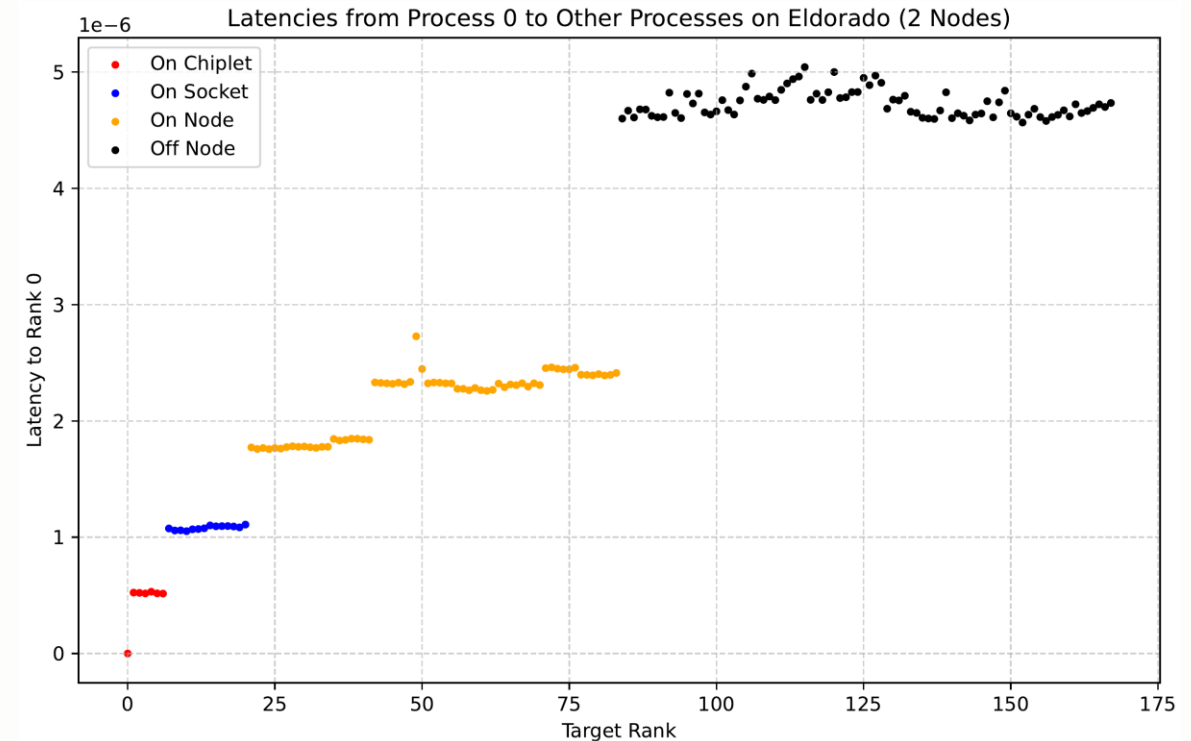
Locality from ping-pong timings

The question

- Can MPI find socket and NUMA-region membership on its own, from ping-pong timings between every pair of ranks?
- On modern nodes, on-node communication is often the expensive path; the best locality level depends on the architecture

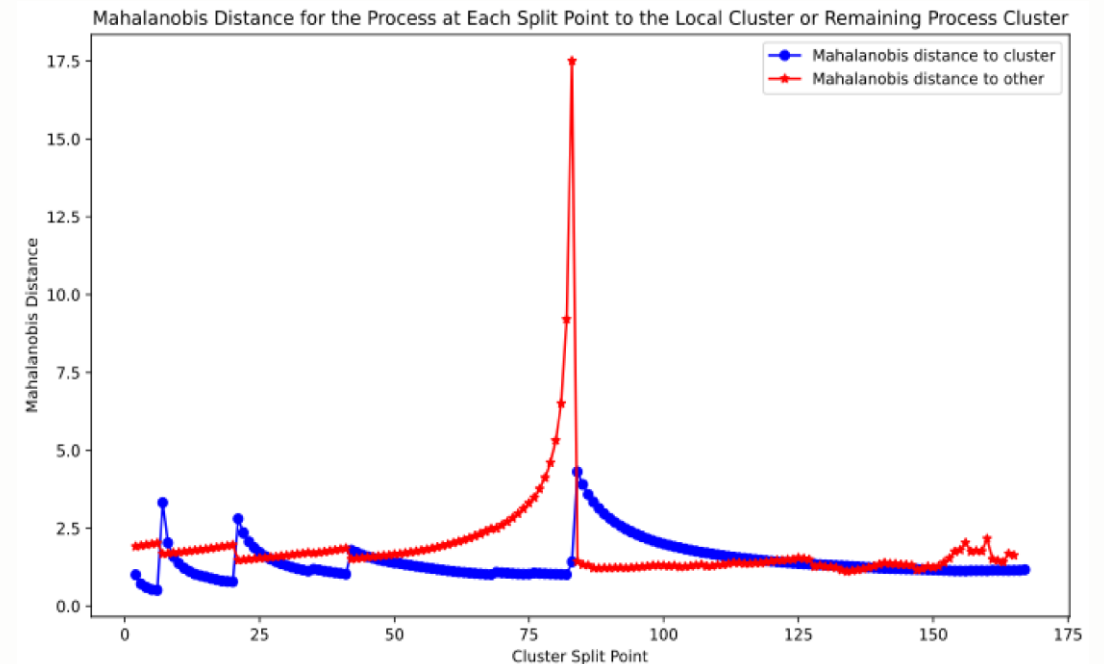
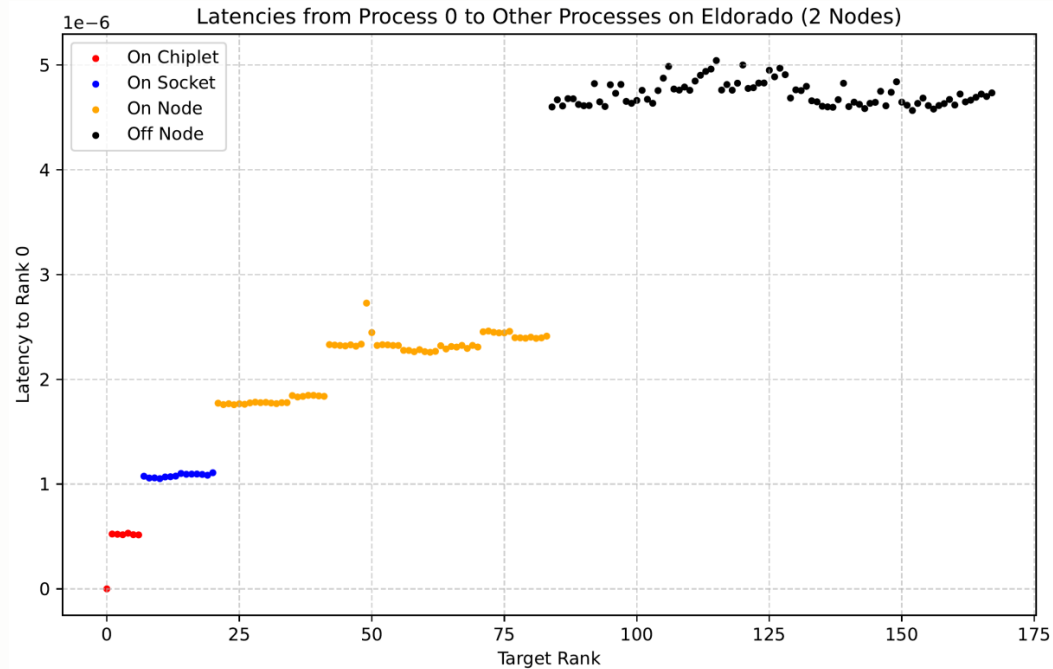
Clustering algorithm

- Round-robin ping-pong between every pair of ranks builds a latency adjacency matrix
- A cluster starts with a rank and its closest rank; a rank joins while its Mahalanobis distance to the cluster is below 1.96
- Local clusters are unioned over common ranks; hierarchy from the first rank whose distance exceeds 1.96



Latency from rank 0 to every other rank on Eldorado, colored by locality. Shannon Kinkead; advisor Amanda Bienz [UNM], mentor Matthew Dosanjh [Sandia]

Locality discovery on Eldorado: results



- Four latency tiers appear without any topology query: on chiplet, on socket, on node, off node [about 0.5, 1.1, 2, and 4.7 μ s to rank 0]
- The Mahalanobis distance from the cluster being built [blue] stays low until the boundary; the distance from the remaining ranks [red] spikes there. The spike is the split
- Result: MPI learns which locality level to exploit; collectives and neighborhood exchanges can choose the cheap path per node type
- Also on the poster: manifold-aware joint optimization of MPI tuning parameters [UMAP embedding, Bayesian optimization]

Topology-aware collectives: the path

1 · Message mode

- Strong progress under MPI Advance's topology-aware collectives, in message-passing mode [ExaMPI]
- Demonstrate the gain: the collective progresses while the application computes



2 · RMA mode

- The same collectives over MPI RMA with strong progress: windows and flags set up once, Put and Get in the loop [Earlybird's route]
- Strong progress makes RMA perform; the API to the application is unchanged



3 · RAPIDS hot loops

- The hot loops of these collectives run on RAPIDS primitives [Cascade and kin]; MPI leaves only those loops
- The rest of the application stays MPI-based

Each stage adds its gain into every user of these collective operations [Trilinos first]. The application does not change.

Common themes

- Optimize communication and computation together. Overlap is the goal in every project
- Regular MPI is not good enough: host-driven paths, weak progress, and the libfabric layer leave performance on the table
- Move the trigger to where the data is: the stream, the kernel, the arriving data, the progress engine
- Early-bird: act as soon as data is ready; never wait for the slowest rank or the host
- Persistence: set up once, run many times. Commands, windows, flags, requests built in advance
- Low-level access is the enabler. Each result rests on a primitive the stack tries to hide

	Trigger	What happens early	Needs from the stack
Stream triggering	GPU stream	MPI starts from the stream; host out of the loop	CXI libfabric extensions; triggered ops
Cascade	The producing kernel	The kernel sends as the data appears; commands prebuilt	libcxr command queues [CXI]
Earlybird collectives	Data arrival [generation flag]	Leader forwards early ranks' data while the late rank is delayed	MPI RMA Put; windows and flags built once
ExaMPI RMA	Progress engine, not the user thread	Communication progresses in the background	libfabric today [the constraint]

SECTION 03

Low-level access, next steps

Why CXI access matters, why libfabric falls short, and where TORRENTs and bespoke kernels take us.



Low-level access: CXI, and the limits of libfabric

Libfabric: too much and too little

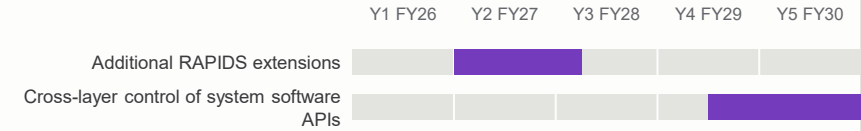
- Not portable between providers
- Hides the special features of specific networks
- Creates new resource limitations
- Not designed with GPUs in mind
- Mid-level middleware: an abstraction barrier
- Cascade on CXI outperforms RCCL on Tuolumne. Libfabric alone is suboptimal even for AI applications

CXI deprecation: bad for DOE/NNSA

- DOE/NNSA must retain access to the lowest-level primitives. Press every network provider for them
- NVIDIA has low-level APIs. All should. CXI is a star example
- Hand-coded or AI-assisted: exploit the smartest features at the lowest level; drive them through kernel initiation
- Interfaces may change. HPE should conserve semantics and resources [mechanisms, counters, registers], not precise interfaces
- With generative AI, we adapt quickly

Beyond CXI: similar work with Cornelis; analogous work with BXI and Bull. Open question: a stream-ordered interface for callers without a kernel.

Next: TORRENTs, bespoke kernels



From RAPIDS to TORRENTs

- TORRENTs: partial RAPIDS, built on the lowest-level APIs each network offers [libcxl on Slingshot; libfabric is not our choice]
- Composed with AI assistants: one TORRENT per network, per primitive, per scale
- The vendor stack hides what the primitives need. We go under it [the CXI slide]
- Slingshot-12 matters: HPE promises more specialized NIC resources [DQEs]. More to exploit; more reason to sit at the lowest level

The GPU side already works this way [NVIDIA]

- Hopper: one elected thread of a producer warp issues the TMA bulk copies; consumer warpgroups [4 warps, 128 threads] issue asynchronous wgmma; mbarriers in shared memory carry stage and phase, and the kernel flips the phases itself
- Blackwell: one thread issues tcgen05.mma; accumulators live in Tensor Memory the kernel allocates and frees; completion is a commit on an mbarrier, then a fence before any read
- Register budgets move between producer and consumer warps at run time; stage and warpgroup counts come from profiling

Communication kernels follow: hyperparameters, optimizations, and kernel composition crafted for each network and each scale of a run. AI assistants make the bespoke affordable.

Conclusions

- Overlap of communication and computation is the goal. The trigger moves to where the data is: the stream, the kernel, the arriving data, the progress engine
- Regular MPI is not good enough. Production MPIs lack strong progress; host-driven paths and libfabric leave performance on the table
- Low-level access is the enabler [CXI today; Slingshot-12 next]. TORRENTs and AI-composed bespoke kernels are the path
- Next: stream-triggered collectives; strong progress into topology-aware collectives and Trilinos; MPI Forum standardization



NNSA PSAAP IV award DE-NA0004267; PSAAP III DE-NA0003966; NSF 2514054, 2450093, 2405142, 2450092