Center for Understandable Performant Exascale Communication Systems

Tennessee TECH

RAPIDS Channel API: Improved Persistent Communication

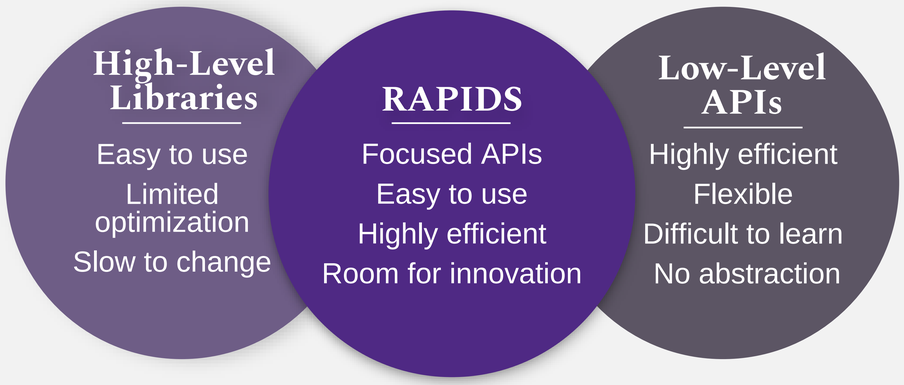
Riley Shipley[1] Anthony Skjellum[1] Purushotham Bangalore[2] Patrick Bridges[3] | [1]Tennessee Tech; [2]U. of Alabama; [3]U. of New Mexico

What is RAPIDS?

RAPIDS (Reduced API Data-transfer Specifications) addresses the following key issues present in existing communication libraries:

- **Performance:** General-purpose libraries sacrifice performance to offer semantics and capabilities to suit many applications
- **Progress:** Catering to a wide audience makes innovation in these libraries untenable
- **Pace:** Even widely agreed upon changes are outpaced by other parts of industry

By dividing functionality from existing models into separate but composable APIs, RAPIDS will provide more performant alternatives for existing models and a venue for innovative new approaches.

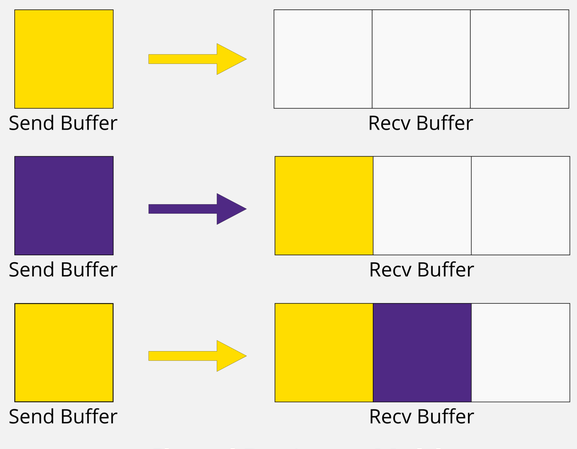


RAPIDS combines the best parts of high-level libraries and low-level APIs

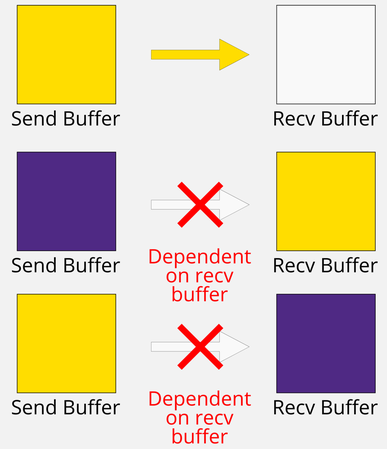
Channel API

The Channel API offers a streamlined persistent communication model implemented using RMA to move data directly, eliminating the need for a matching queue on the receiving process.

One of the most important features of the Channel model is that the remote buffer it creates is segmentable, meaning that users have complete freedom in how much of the remote buffer they use. This allows for multiple messages to be written to different buffer segments without needing to synchronize with the remote process, as seen below.



Channel Persistence Model



MPI Persistence Model

```
// Write to segment starting at offset
SendChannel::i_put(offset, count);

// Wait for segment write to complete
SendChannel::wait(offset, count);

// Wait for incoming segment at offset
RecvChannel::wait(offset, count);

// Give write access to sender
RecvChannel::release(offset, count);
```

Core Channel functions

Development Timeline

Below are currently planned RAPIDS APIs in the order they will be developed.

Summary

The RAPIDS initiative seeks to deliver innovative and performant communication APIs. The first of these is the Channel API, improving persistent communication by implementing it via RMA and allowing remote buffers to be segmented to fit user needs.

Acknowledgements

This work was performed with partial support from the National Science Foundation under Grants Nos. CCF-2405142 and CCF-2412182, the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966, and Tennessee Technological University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the DOE NNSA.

Center for Understandable Performant Exascale Communication Systems

Tennessee TECH

Enabling Stream-Triggered MPI+X Backends for Cabana Benchmarks

Evan Drake Suggs[1], Patrick Bridges[2], Derek Schafer[2], Anthony Skjellum[1] | [1]Tennessee Technological University, [2]University of New Mexico

Background

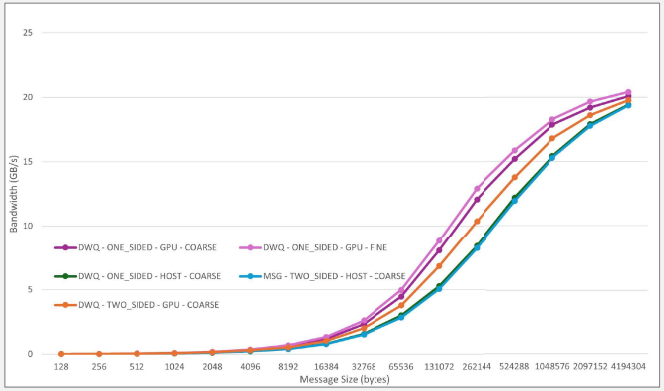
- We have previously surveyed various stream-triggered MPI interfaces. They are currently implemented by several vendors via separate and incompatible APIs across many different GPUs and backends in often mutually incompatible ways.
- CUP-ECS's MPI Advance stream-triggering library works with several backends (CUDA, CXI, etc) to create a portable interface for stream triggering
- We are in the process of integrating this library as a backend in Cabana and CUP-ECS's CabanaGhost benchmark suite:
  - Cabana is a performance-portable particle simulation library built on Kokkos with an optional MPI backend.
  - CabanaGhost is a bulk synchronous parallel regular mesh benchmark for teaching and exploring parallel and performance portable frameworks, consisting of implementations of the Game of Life and the Jacobi method.

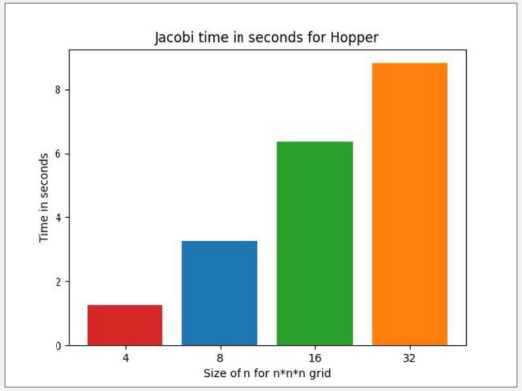
```
void Distributor::Distributor(ExecutionSpace &e)
{
    // Create non-blocking send and receive operations
    for ( int n = 0; n < num_n; ++n ) {
        ...
        auto recv_subview = Kokkos::subview(recv_buffer, recv_bounds);
        auto send_subview = Kokkos::subview(send_buffer, send_bounds);
        MPI_Recv_init( recv_subview.data(), recv_subview.size(), ...,);
        MPI_Send_init( send_subview.data(), send_subview.size(), ...,
    }
    // Pair up send/recv buffers, create a queue for starts and waits
    MPIX_Matchall(halo_ops.data(), halo_ops.size());
    MPIX_Queue_init(&queue, MPIX_QUEUE_TYPE_HIP, &e.hipStream());
}

void Distributor::distributeData(AoSoA_t& src, AoSoA_t& dst)
{
    // All operations are enqueued to the stream, which enqueue
    // them to the progress engine associated with the queue
    MPIX_Enqueue_startall(queue, halo_ops.data(), num_n);
    Kokkos::parallel_for(pack_buffer_func, src, send_buffer );
    MPIX_Enqueue_startall(queue, halo_ops.data() + num_n, num_n);
    MPIX_Enqueue_waitall(queue);
    Kokkos::parallel_for( unpack_recv_func, dst, rcev_buffer);
}
```

Methods

- The MPI Advance stream-triggering library works with CUDA, CXI, and default thread implementations.
- A halo exchange shares data between MPI processes that are nearby, forming an overlapping halo over a submatrix using scatter and gather functions.
- We created a branch of Cabana with a version of the grid subpackage halo object called the StreamHalo object that front-loads the initialization of Send/Recvs, uses enqueue variants of scatter and gather, and uses stream-triggering's queue function to schedule and wait on the underlying communication.
- Once this was done, we refactored Jacobi and Game of Life to utilize the enqueueScatter and enqueueGather.
- The code snippet shows an idealized interface for stream-triggered MPI that does not break current MPI semantics, so current programs remain valid.

Results





- The first graph is a packing ping-pong with the MPI API on the HPE CXI Libfabric that integrates Cabana and Kokkos mini-apps on LLNL's Tioga system.
- The second uses versions of CabanaGhost and Cabana integrated with the MPI Advance stream-triggering library on the UNM Hopper condo cluster.
- Hopper has 53 compute nodes with a combined 1,760 Intel Xeon Gold CPUs and 8.2 TB of RAM. We allocated two nodes and two GPUs under the CUP-ECS partition.
- We ran tests with the threaded version of the stream-triggered benchmark using the Jacobi test.

Conclusions and Future Work

- **Conclusions**
  - This project successfully integrated MPI Advance stream-triggering with Cabana and CabanaGhost on Hopper
  - The results are promising but some issues remain with memory on Hopper
- **Future Work**
  - Stream-triggered CabanaGhost working with CXI on the Tioga system and CUDA on Hopper in the near-future
  - involves creating similar work for the creation of a Kokkos-level stream-triggered interface and further experiments with MPI stream-triggering
  - Port into the KokkosComm library to enhance MPI+Kokkos integration

Acknowledgements

Funding in part is acknowledged from these NSF Grants CCF-2405142 and CCF-2412182, as well as OAC-2103510 and the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966.

Center for Understandable Performant Exascale Communication Systems

Tennessee TECH

Enabling Performant Inter-Node Communication for Kokkos Views

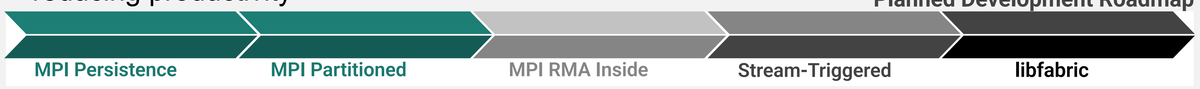
C. Nicole Avans[1,2], Carl Pearson[1], Jan Ciesko[1], Evan Suggs[2], Stephen L. Olivier[1], Anthony Skjellum[2]
[1]Sandia National Laboratories, [2]Tennessee Technological University

Introduction

- **The Kokkos Comm library introduces a new communication interface integrated with Kokkos Views**
- A unified performance-portable ecosystem for on- and off-node parallel programming
- Kokkos Comm optimizes movement of data by abstracting implementation-specific details and marshaling and unmarshaling of data away from the end-user programmer

Design for Performance, Portability, and Productivity

- Enable the fastest path for data to move without changing the program
- Provide intuitive support for neighbor collectives by managing data as subviews
- Native multi-transport communication support for performance and portability without reducing productivity

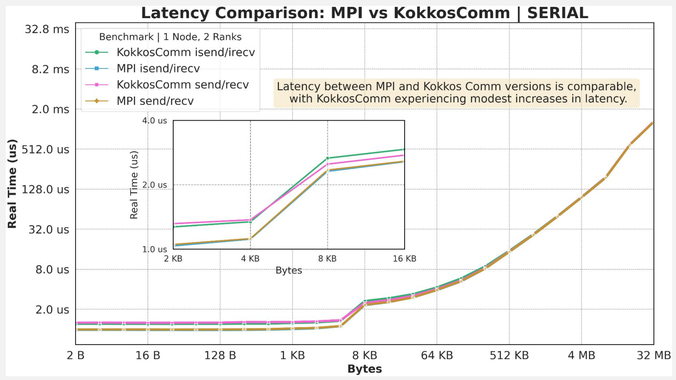


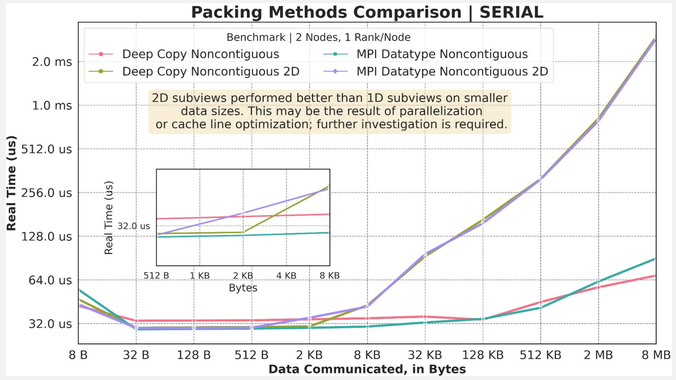
Current Challenges to Inter-Node Communication

- Programming with non-contiguous data in MPI typically requires a serialization strategy using data repacking or MPI Datatypes
- Platform-specific optimization for processors and interconnects presents undue burden to scalable application developers
- Heterogeneous architectures complicate data exchange and reduce performance-portability

Addressing Challenges

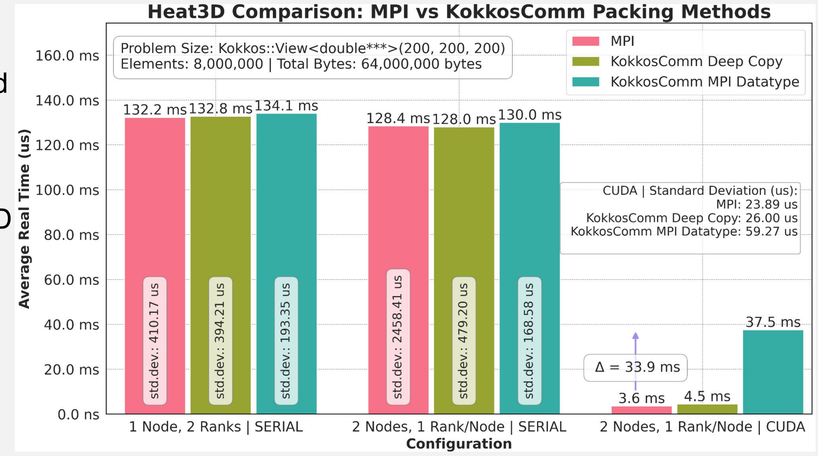
- Serialization and deserialization is managed and abstracted into two supported types: Deep Copy, and MPI Datatype
- GPU memory space communication is supported based on Kokkos enabled backends (e.g., CUDA, HIP)





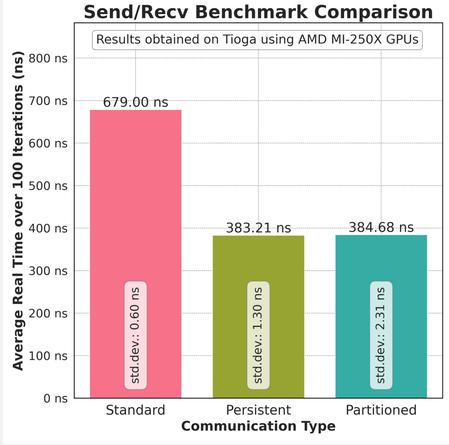
Methods

- Experiments were executed to optimize performance and identify bottlenecks on multiple nodes of SNL Weaver (NVIDIA Tesla V100) and LLNL Tioga (AMD MI250X)
- Packing method performance, latency, and real-time of execution were measured via a Heat3D miniapp and standard OSU benchmarks



Highlights and Conclusions

- Implemented a Channel object that manages half-duplex persistent and single-partition partitioned point-to-point communication in Kokkos Comm, to be expanded upon
- Added explicit HIP support to Kokkos Comm, to complement existing CUDA support
- Kokkos Comm has been fully instrumented for use with Kokkos Tools, enabling rapid identification of performance bottlenecks and the ability to trace program behavior



Future Opportunities

- Enhance support to include arbitrary Kokkos View types and `mdspan`
- Add new communication space backends (e.g., NCCL) to extend Kokkos Comm beyond MPI-based communication for higher performance
- Add support for stream-triggered communication and libfabric to enable optimizations for system-specific performance via interface with external libraries Cabana and RAPIDS

Acknowledgements

This project wishes to acknowledge the fruitful collaboration with our colleagues at Université Paris-Saclay: Gabriel Dos Santos, Cédric Chevalier, Hugo Taboada, and Marc Pérache.

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-NA003525.

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

Center for Understandable Performant Exascale Communication Systems

Tennessee TECH

Optimized RMA-based MPI_Alltoallv Operations

Evelyn Namugwanya[1], Amanda Bienz[2], Matthew Dosanjh[3], Anthony Skjellum[1] | [1]TN TECH, [2]UNM, [3]SANDIA

Introduction

The MPI_Alltoallv operation is critical in scientific workloads involving irregular communication. RMA (Remote Memory Access) promises asynchronous, one-sided communication, and could out-perform message-based implementations of collective operations that are specified in the MPI Standard.

The goal is to design and develop more efficient and faster RMA-based Alltoallv operations and compare them with traditional MPI_ Alltoallv against MPI-Advance's two RMA variants over a range of scalability.

This poster compares the performance of three RMA_Alltoallv implementations: the default MPI implementation of MPI_Alltoallv, MPI-Advance's RMA-based Alltoallv with MPI_Win_fence, and RMA-based Alltoallv with MPI_Win_lock.

Experiments were conducted on 2, 4, and 8-node configurations using LLNL's Dane and Lassen HPC Supercomputers.

Algorithms developed and Compared

- **MPI_Alltoallv**:
  Standard MPI implementation.
- **RMA_winfence**:
  Uses MPI_Win_fence for synchronization.
- **RMA_winlock**:
  Uses MPI_Win_lock/MPI_Win_unlock for finer-grained control.
  Uses MPI_Win_flush for completions.

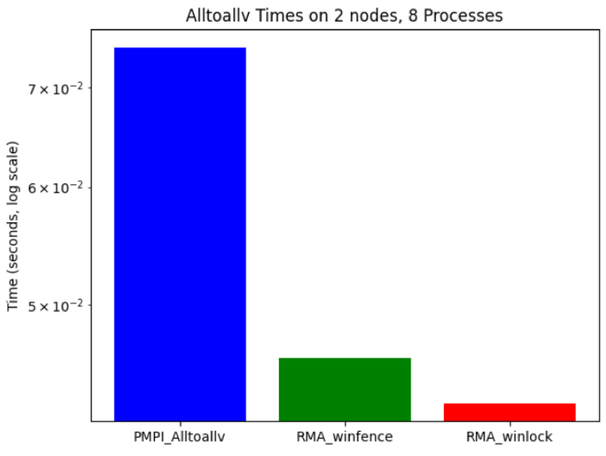
Results

- The default MPI implementation performs better on two nodes, but the advantage narrows as nodes and processes increase.
- RMA_Winlock is consistently slightly faster than RMA_Winfence.

Insights

- RMA approaches reduce overhead in large process counts and large messages but not on fewer nodes.
- Synchronization method matters: winlock outperforms fence marginally.

Weak Scaling Results

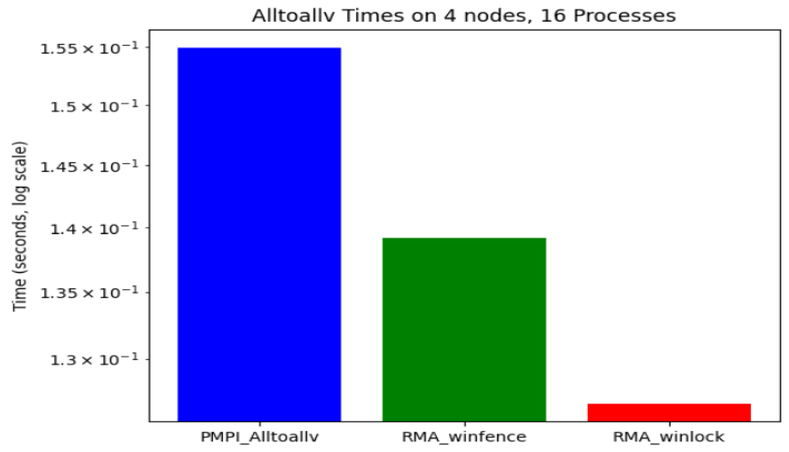


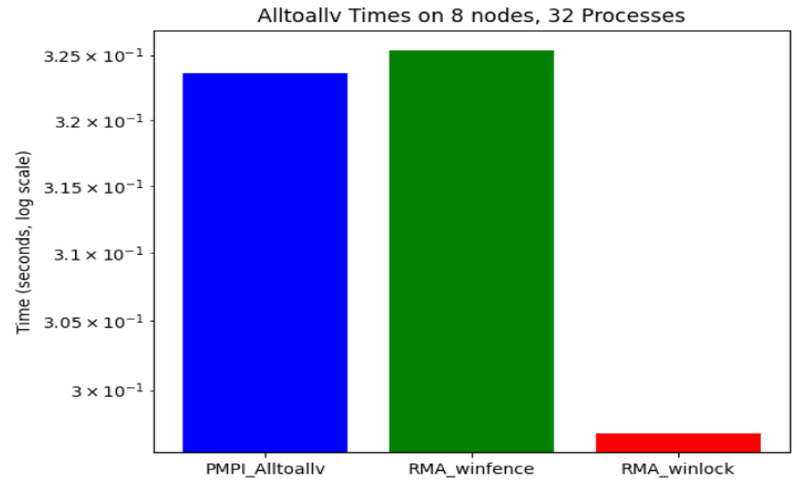
For all graphs, low bars mean less time taken and high bars mean more time taken (in seconds)

Conclusions

- The default MPI_Alltoallv offers strong performance on small process counts.
- RMA variants, especially with fine-grained locking, scale better with more nodes and process count.
- This can be attributed to its flexibility as compared to MPI_Win_fence.
- Future work includes Persistent RMA Alltoallv and OpenSHMEM instead of MPI RMA.

Message size = 33,554,432 bytes





Acknowledgements

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

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF NEW MEXICO

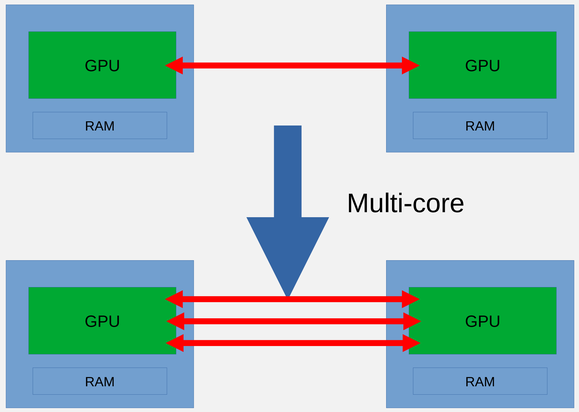
GPU Allreduce using Interprocess Communication

Michael Adams and Amanda Bienz | Department of Computer Science

Background

Applications such as deep neural netwiork training rely on the Allreduce operation for data communication and use GPUs for efficient linear algebra implementations. In deep neural network training, the Allreduce is used to average the backpropagation gradients across all GPUs.

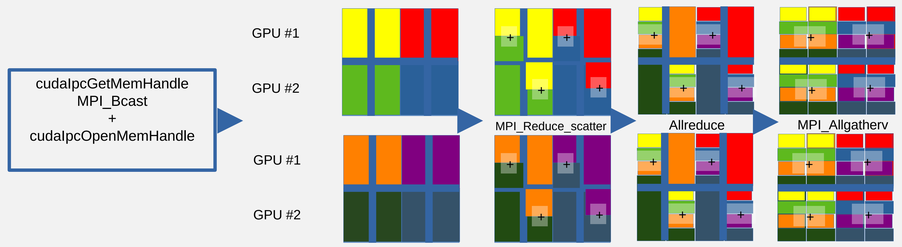
It is noted, however, that communication is dependent on the CPU and network configuration of the system, even in the GPUDirect RDMA case. Thus, we propose to take advantage of growing core counts on supercomputers, such as NCSA Delta, with a lane-aware approach [1] to utilize all cores during communication via CUDA Interprocess Communication to yield further speedup.



Methods

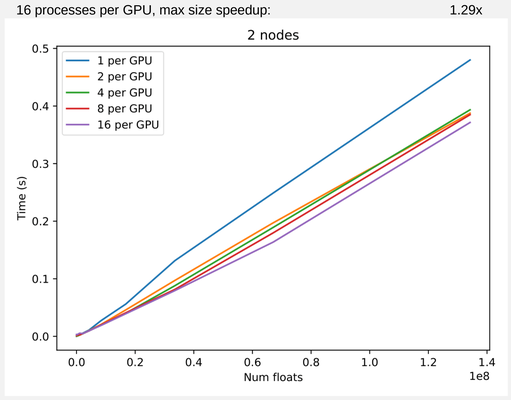
The optimization of the Allreduce collective operation relies on multiple MPI ranks being launched by the scheduler per GPU. After the send and receive buffers are allocated per GPU, all ranks associated with each GPU are given a pointer to the buffer using the CUDA IPC API; a root process per GPU passes the buffer to cudaIpcGetMemHandle to get a handle to broadcast to share the buffer with its on-node ranks, then these ranks retrieve the pointer with cudaIpcOpenMemHandle.

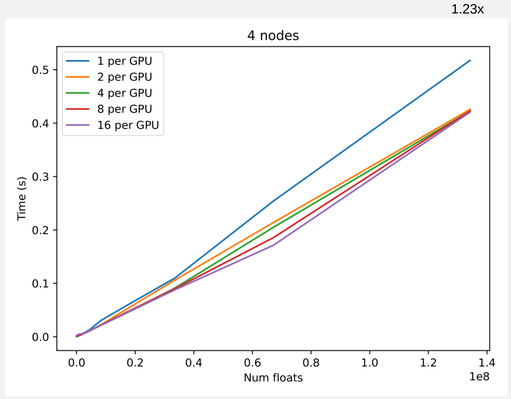
Then, as seen below, the reduction proceeds in a lane-aware manner [1]. A Reduce scatter first divides each buffer into chunks equal to the number of ranks per node, reducing the chunk size associated with each rank per GPU on a node. Then, an inter-node Allreduce occurs to ensure all ranks on every node have a completely reduced chunk of the solution. Finally, the chunks on a node are assembled into the solution with an Allgatherv. It should be noted that displacements relative to the start of the buffer are used by each GPU's ranks in order to ensure each rank stays within bounds of its chunk to allow synchronization to be ignored.

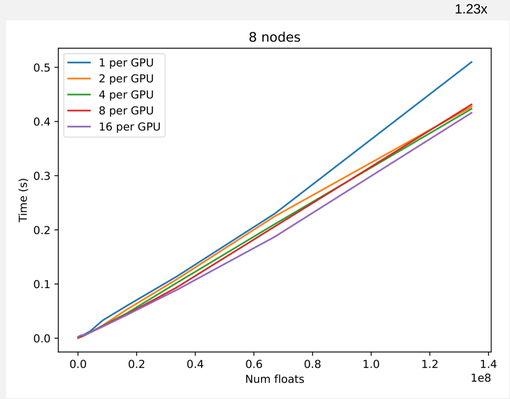


Results

16 processes per GPU, max size speedup: 1.29x 1.23x 1.23x







Conclusions

Speedup for large buffer sizes using 2 to 16 ranks per GPU is seen on two, four, and eight nodes of NCSA Delta where each node has 4 NVIDIA A100 and 64 AMD EPYC CPU cores. Careful consideration of rank placement evenly throughout all NUMA nodes (one per GPU) provided initial validation.

Further validation on additional systems such as PSC Bridges-2 and Purdue Anvil is needed. Scaling studies will be performed to show applicability to scientific and deep learning problems.

Acknowledgements

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the research computing resources used in this work.
In addition, we would like to thank the PSAAP program, funded in part by the United States Department of Energy, for accces to the Lassen, Tioga, and Chicoma platforms used in this work.
Finally, we would like to thank the NSF Access program for access to NCSA Delta, Purdue Anvil, and PSC Bridges-2

References

1. J. L. Traff and S. Hunold, Decomposing mpi collectives for exploiting multi-lane communication, 2020 IEEE International Conference on Cluster Computing (CLUSTER), (2020)

2. T. T. Nguyen, M. Wahib and R. Takano, Topology-aware Sparse Allreduce for Large-scale Deep Learning, 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC), (2019)

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF NEW MEXICO

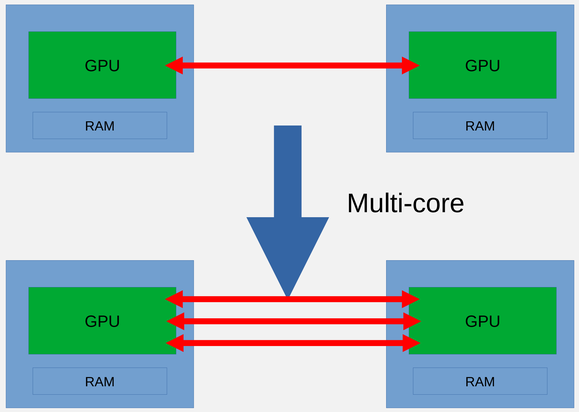
GPU Allreduce using Interprocess Communication

Michael Adams and Amanda Bienz | Department of Computer Science

Background

Applications such as deep neural netwiork training rely on the Allreduce operation for data communication and use GPUs for efficient linear algebra implementations. In deep neural network training, the Allreduce is used to average the backpropagation gradients across all GPUs.

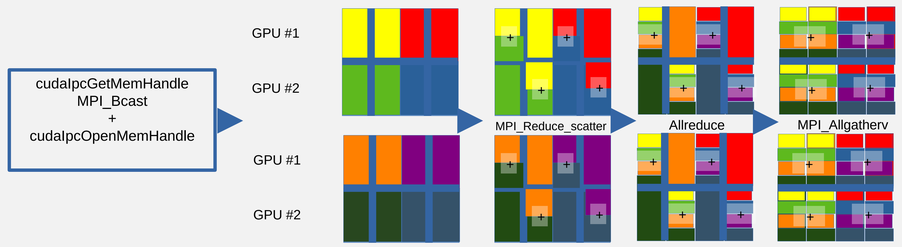
It is noted, however, that communication is dependent on the CPU and network configuration of the system, even in the GPUDirect RDMA case. Thus, we propose to take advantage of growing core counts on supercomputers, such as NCSA Delta, with a lane-aware approach [1] to utilize all cores during communication via CUDA Interprocess Communication to yield further speedup.



Methods

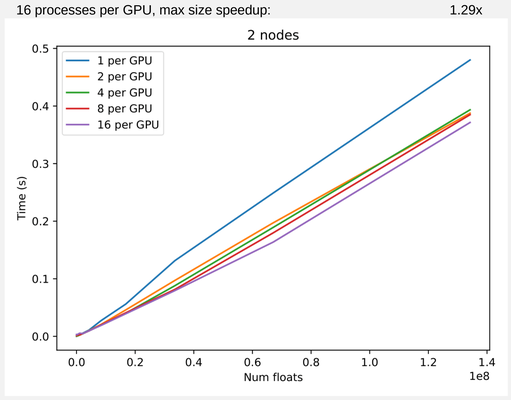
The optimization of the Allreduce collective operation relies on multiple MPI ranks being launched by the scheduler per GPU. After the send and receive buffers are allocated per GPU, all ranks associated with each GPU are given a pointer to the buffer using the CUDA IPC API; a root process per GPU passes the buffer to cudaIpcGetMemHandle to get a handle to broadcast to share the buffer with its on-node ranks, then these ranks retrieve the pointer with cudaIpcOpenMemHandle.

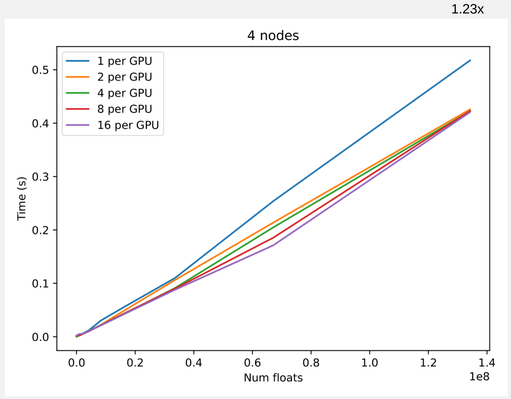
Then, as seen below, the reduction proceeds in a lane-aware manner [1]. A Reduce scatter first divides each buffer into chunks equal to the number of ranks per node, reducing the chunk size associated with each rank per GPU on a node. Then, an inter-node Allreduce occurs to ensure all ranks on every node have a completely reduced chunk of the solution. Finally, the chunks on a node are assembled into the solution with an Allgatherv. It should be noted that displacements relative to the start of the buffer are used by each GPU's ranks in order to ensure each rank stays within bounds of its chunk to allow synchronization to be ignored.

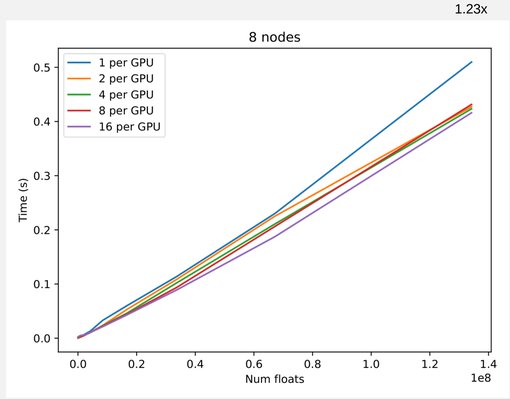


Results

16 processes per GPU, max size speedup: 1.29x 1.23x 1.23x







Conclusions

Speedup for large buffer sizes using 2 to 16 ranks per GPU is seen on two, four, and eight nodes of NCSA Delta where each node has 4 NVIDIA A100 and 64 AMD EPYC CPU cores. Careful consideration of rank placement evenly throughout all NUMA nodes (one per GPU) provided initial validation.

Further validation on additional systems such as PSC Bridges-2 and Purdue Anvil is needed. Scaling studies will be performed to show applicability to scientific and deep learning problems.

Acknowledgements

We would like to thank the UNM Center for Advanced Research Computing, supported in part by the National Science Foundation, for providing the research computing resources used in this work.
In addition, we would like to thank the PSAAP program, funded in part by the United States Department of Energy, for accces to the Lassen, Tioga, and Chicoma platforms used in this work.
Finally, we would like to thank the NSF Access program for access to NCSA Delta, Purdue Anvil, and PSC Bridges-2

References

1. J. L. Traff and S. Hunold, Decomposing mpi collectives for exploiting multi-lane communication, 2020 IEEE International Conference on Cluster Computing (CLUSTER), (2020)

2. T. T. Nguyen, M. Wahib and R. Takano, Topology-aware Sparse Allreduce for Large-scale Deep Learning, 2019 IEEE 38th International Performance Computing and Communications Conference (IPCCC), (2019)

SAND2025-05968O

Sandia National Laboratories

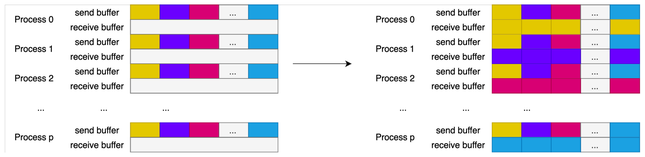
ANALYZING ALLTOALL ALGORITHMS ON MANY-CORE SYSTEMS

Shannon Kinkead (Sandia National Laboratories, University of New Mexico)
In collaboration with: Amanda Bienz (University of New Mexico)

Introduction

Alltoall

- Critical to matrix operations and machine learning
- Each process sends a portion of its data to every other process
- Multiple algorithms
- Performance depends on data size, process count, architecture, synchronization requirements, etc.



Methods

- 8 Alltoall algorithms compared
  - System MPI
  - **Pairwise Exchange** (uses MPI send/recv in pairs)
  - **Nonblocking** (uses MPI isend/irecv/waitall)
  - **Hierarchical** (algorithm 2, local comm is all ranks on node)
  - **Multileader** (algorithm 2, local comm is a group within the node)
  - **Node Aware** (algorithm 3, local comm is all ranks on node)
  - **Locality Aware** (algorithm 3, local com is a group within a node)
  - **Multileader Locality Aware** (multileader, replaces MPI alltoall with node aware)
- 3 HPC Systems
  - Amber (Sandia National Laboratories)
  - Dane (Lawrence Livermore National Laboratory)
  - Tachi (Sandia National Laboratories)
- 2 - 32 Nodes
- Message sizes from 4 bytes to 2 kb

Algorithms

Hierarchical

- Aggregates all data to be sent by each process to leader of local communicator
- Exchanges the data to be sent to other local communicators to group communicator
- Scatters data from other processes to local communicator
- Data is repacked between each communication step

Algorithm 2: Hierarchical

Input: p {process rank}
n {process count}
s_{size}, s_{type}, s_{buf} {send size, type, and buffer}
r_{size}, r_{type}, r_{buf} {recv size, type, and buffer}
`local_comm` {All processes local to region}
ppn, l {Size and rank of local_comm}
`group_comm` {All processes with equal local rank}

$s_{\text{buf}_{leader}} \leftarrow$ **buffer of size** $s_{\text{size}} \cdot n \cdot ppn$
$r_{\text{buf}_{leader}} \leftarrow$ **buffer of size** $r_{\text{size}} \cdot n \cdot ppn$

// Gather to leader
MPI_Gather(s_{buf}, $s_{\text{size}} \cdot n$, ..., $s_{\text{buf}_{leader}}$, ..., `local_comm`)

Repack Data

// Alltoall exchange between leaders
MPI_Alltoall($s_{\text{buf}_{leader}}$, $s_{\text{size}} \cdot ppn^2$, ..., $r_{\text{buf}_{leader}}$, $r_{\text{size}} \cdot ppn^2$..., `group_comm`)

Repack Data

// Scatter from leader
MPI_Scatter($r_{\text{buf}_{leader}}$, $r_{\text{size}} \cdot n$, ..., r_{buf}, ..., `local_comm`)

Hierarchical Alltoall algorithm

Locality-Aware

- Exchanges all data for processes outside group communicator within the group communicator
- Exchanges all data within a region in the local communicator
- Data is repacked between each communication step

Algorithm 3: Locality-Aware

Input: p {process rank}
n {process count}
s_{size}, s_{type}, s_{buf} {send size, type, and buffer}
r_{size}, r_{type}, r_{buf} {recv size, type, and buffer}
`local_comm` {All processes local to region}
ppn, l {Size and rank of local_comm}
`group_comm` {All processes with equal local rank}

$tmp_{\text{buf}} \leftarrow$ **buffer of size** s_{size}

// Inter-region Alltoall
MPI_Alltoall(s_{buf}, $s_{\text{size}} \cdot ppn$, ..., tmp_{buf}, $r_{\text{size}} \cdot ppn$..., `group_comm`)

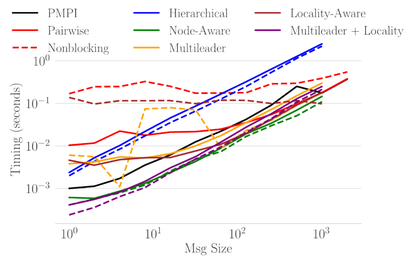
Repack Data

// Intra-region Alltoall
MPI_Alltoall(tmp_{buf}, $r_{\text{size}} \cdot ppn$, ..., r_{buf}, $r_{\text{size}} \cdot ppn$..., `local_comm`)

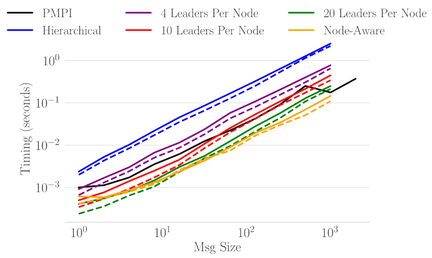
Locality-Aware Alltoall algorithm

Results

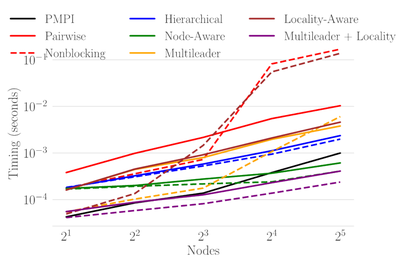
Dane



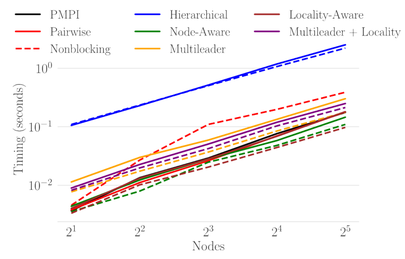
32 nodes, message scaling from 4b to 2 kb



32 nodes, message scaling from 4b to 2 kb, scaling leaders per node

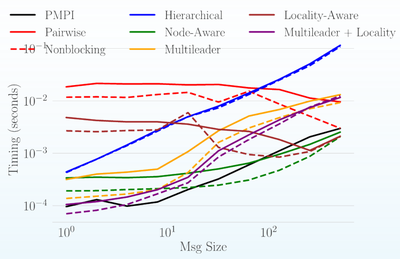


4b message, scaling nodes from 2 - 32



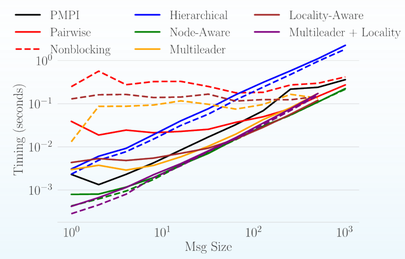
2kb message, scaling nodes from 2 - 32

Tachi



32 nodes, message scaling from 4b to 2 kb

Amber



32 nodes, message scaling from 4b to 2 kb

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.

References: N. Netterville, K. Fan, S. Kumar, and T. Gilray, "A visual guide to MPI all-to-all," in 2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW). IEEE, 2022, pp. 20–27.