MPI implementation optimization for Slingshot network

Adam Lavely
awlavely@lbl.gov
Lawrence Berkeley National Lab
Berkeley, California, USA

Afton Geil
angeil@lbl.gov
Lawrence Berkeley National Lab
Berkeley, California, USA

Neil Mehta
namehta@lbl.gov
Lawrence Berkeley National Lab
Berkeley, California, USA

Rahulkumar Gayatri
rgayatri@lbl.gov
Lawrence Berkeley National Lab
Berkeley, California, USA

Brandon Cook
bgcook@lbl.gov
Lawrence Berkeley National Lab
Berkeley, California, USA

Abstract

The Message Passing Interface (MPI) has long served as the de facto standard for distributed parallel programming. Historically, support for emerging network architectures has primarily been driven by vendor-specific implementations, such as Cray MPICH for the Slingshot interconnect. However, optimizing open-source MPI implementations for these new networks remains essential, as open-source MPI libraries are frequently required in containerized workflows, third-party software deployments, and diverse user environments. In this paper, we evaluate the performance characteristics of multiple MPI implementations on the Perlmutter supercomputer using a the OSU suite of micro-benchmarks. Results indicate that, with default configuration options and run-time parameters, the open-source MPICH implementation exhibits significantly lower performance compared to the vendor-optimized Cray MPICH. In contrast, OpenMPI demonstrates variable performance relative to MPICH under certain message-size regimes with some MPI Procedures. Our analysis identifies several causes for these performance disparities, including differences in build-time configuration settings, network protocol threshold parameters, and algorithm selection heuristics while indicated the further optimization is possible in both Cray MPICH, the open source MPICH and OpenMPI.

Keywords

MPI, Cray-MPICH, MPICH, OpenMPI, Slingshot

ACM Reference Format:
Adam Lavely, Afton Geil, Neil Mehta, Rahulkumar Gayatri, and Brandon Cook. 2025. MPI implementation optimization for Slingshot network. In . ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Conference'25, New York, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Message Passing Interface (MPI)[10] is a widely adopted distributed-memory parallel programming model. Most enterprise-scale supercomputers provide support for one or more MPI implementations, typically recommending vendor-provided MPI libraries optimized specifically for their underlying network infrastructures. However, certain practical scenarios necessitate the use of alternative MPI libraries, including:

- Access to new or experimental MPI features unavailable in vendor implementations,
- Dependencies arising from containerized workflows,
- Workflow-specific requirements tied to particular MPI implementations.

To address these requirements, the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC) provides two open-source MPI libraries—MPICH[12] and OpenMPI[11]—in addition to the vendor-optimized Cray MPICH[7] tailored for the Slingshot interconnect[8, 9]. Although Cray MPICH is the officially recommended MPI library due to its optimization for the Slingshot network, some users may desire alternative implementations, particularly in scenarios involving NVIDIA containers, which often rely on OpenMPI.

Argonne's MPICH serves as the upstream implementation for several other MPI libraries, including Cray MPICH. OpenMPI is commonly deployed on InfiniBand-based systems and is often included in NVIDIA's distributions and has recently introduced support for the Slingshot network starting from its 5.x release series.

This article compares the default out-of-the-box performance of Cray MPICH, Argonne MPICH and OpenMPI on the Perlmutter system at NERSC. We find that, with default configuration options and run-time parameters, the open-source MPICH implementation exhibits significantly lower performance compared to the vendor-optimized Cray MPICH. In contrast, OpenMPI demonstrates variable performance relative to MPICH under certain message-size regimes with some MPI Procedures.

An MPI library's performance depends on several components, such as the algorithm implementation efficiency, algorithm selection heuristics, underlying network hardware, network software stack, and interfaces to system and node topology. The analysis presented here primarily focuses on the MPI layer itself and does not systematically examine the lower-level components. However, we note that most of these building blocks remain consistent across

MPI implementations, with the notable exception of PMI: Open-MPI utilizes the `pmix` interface[6], MPICH relies on `pmi2`, and Cray MPICH employs the proprietary `cray_shasta` PMI implementation.

2 Related Work

Khorassani et al. [15] evaluated different MPI implementations on the Slingshot-10 network, including OpenMPI, MVAPICH2-GDR, Cray MPICH, and RCCL and found significant variation between the implementations across collectives for different message sizes. They also performed some early evaluations of MPI implementations for CPU-only nodes on a Slingshot-11 network.

Beebe et al. [3] previously compared the performance of Cray MPICH, MPICH, and Open MPI on Slingshot-11 for the OSU benchmarks[5]. They found that Cray MPICH outperformed the open source MPI implementations, but predicted that this performance gap would shrink over time as the open source implementations had time to be optimized. They also analyzed the performance of the LAMMPS and WarpX applications using different MPI libraries, finding that the choice of MPI library had little impact on LAMMPS, but a significant effect on the communication-bound WarpX application.

The open source community is constantly working to improve MPI implementations for new networks. Shehata et al. [20] describe their recent work to support Open MPI on Slingshot-11, including the development of a new libfabric provider, LINKx, to optimize both intra-node and inter-node communication. Guo [14] details recent developments in Argonne MPICH to address challenges for exascale machines.

3 Hardware and Software Overview

The experiments presented in this paper were conducted on the Perlmutter supercomputer located at the Lawrence Berkeley National Laboratory. Perlmutter comprises both CPU-only and GPU-accelerated compute nodes. Specifically, the system contains 3,072 CPU nodes, each equipped with two AMD EPYC "Milan" processors[2], with each processor providing 64 cores and supporting two hardware threads per core. These CPU nodes are interconnected via a single HPE Cassini Network Interface Card (NIC).

Additionally, Perlmutter includes 1,536 GPU-accelerated nodes, each consisting of one AMD EPYC CPU and four NVIDIA Ampere GPUs[19]. GPU nodes are interconnected with other nodes through four HPE Cassini NICs per node, while within-node GPU pairs are interconnected using four third-generation NVLinks.

All compute nodes on Perlmutter are interconnected via HPE's Slingshot network, arranged in a dragonfly topology[16].

The software environment utilized in this study is based on HPE's Cray Programming Environment (CPE), specifically version CPE/24.07. Our evaluation includes the latest available versions (at the time of writing) of three MPI implementations: Cray MPICH 8.1.30, MPICH 4.3.0, and OpenMPI 5.0.7. A consistent network software layer, libfabric version 1.20[13], was used across all three MPI implementations.

4 Methodology

To evaluate the performance of different MPI implementations, we utilize benchmarks from the OSU benchmark suite. Given the extensive nature of the full benchmark suite, we focus our analysis on a representative subset of point-to-point and collective communication routines, which are commonly employed in distributed-memory workflows.

For point-to-point communication studies, we measure the bandwidth achieved when exchanging messages between two MPI tasks. In the case of collective communication routines, we evaluate the latency incurred when performing single-rooted operations, such as `scatter` and `gather`[17], with progressively increasing message sizes. Additionally, we examine the performance differences in multi-rooted collective routines, such as `alltoall`[4], as well as multi-rooted collectives involving computation, such as `allreduce`.

Our analysis places particular emphasis on MPI routines that involve GPU buffers motivated by the fact that Perlmutter nodes are equipped with four NVIDIA A100 GPUs per node, which collectively account for more than 70% of the system's total computational power. Consequently, most workflows running on Perlmutter are specifically designed to leverage GPU acceleration. Therefore, we concentrate most of our detailed analysis specifically on MPI routines utilizing GPU buffers, as these scenarios provide more meaningful insights into performance distinctions between MPI implementations for most workflows. Accordingly, all subsequent results presented in this paper are derived exclusively from experiments conducted on Perlmutter's GPU-equipped nodes, with each MPI task explicitly involving GPU buffers.

Our results focus on the case of 1 MPI Process per GPU, a very common configuration in NERSC's workload. GPU affinity of processes is handled by a wrapper script that sets `CUDA_VISIBLE_DEVICES` in conjuction with Slurm's `--gpu-bind none` in order to ensure cgroups associated with Slurm tasks do not interfere with device to device IPC.

In order to maintain legible plots we show results collected with 64 nodes, but the results discussed are representative of what we find at other scales. Our focus was not on full system level performance.

4.1 Point-to-Point

The `bibw` test from the OSU benchmark suite measures the bidirectional bandwidth between two MPI tasks. Figure 1 presents the bandwidth results obtained from this test, comparing two communication scenarios: (1) intra-node communication, in which both tasks reside on the same node, and (2) inter-node communication, in which each task resides on a separate node.

For inter-node communication, where tasks reside on different nodes, all three MPI implementations achieve nearly identical bandwidth performance across the range of message sizes tested.

However, for inter-node communication, clear performance differences emerge at larger message sizes. While all three MPI implementations perform similarly at smaller message sizes, Cray-MPICH achieves the highest bandwidth at larger message sizes, followed by MPICH, and then OpenMPI, which exhibits the lowest bandwidth among the three implementations.

5 Collectives

In this section, we compare the performance of `MPI_Scatter`, `MPI_Gather`, `MPI_Alltoall` and `Allreduce`.

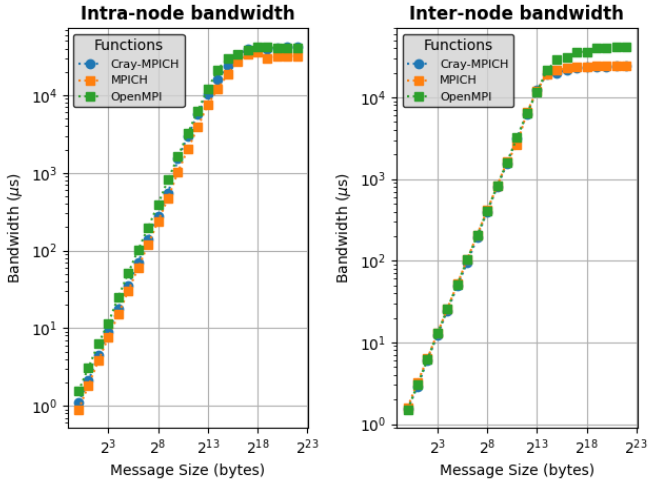

Figure 1: Bidirectional bandwidth achieved between two MPI processes on Perlmutter GPU nodes.

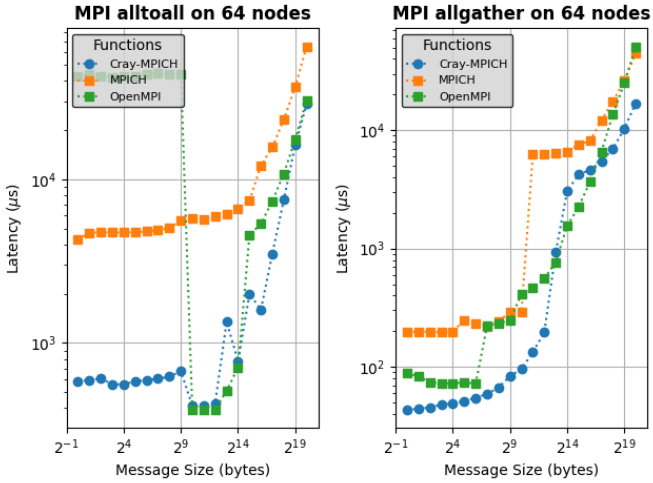

Figure 3: Latency incurred by multi-rooted MPI collective operations (`alltoall` and `allgather`) executed on 64 Perlmutter GPU nodes.

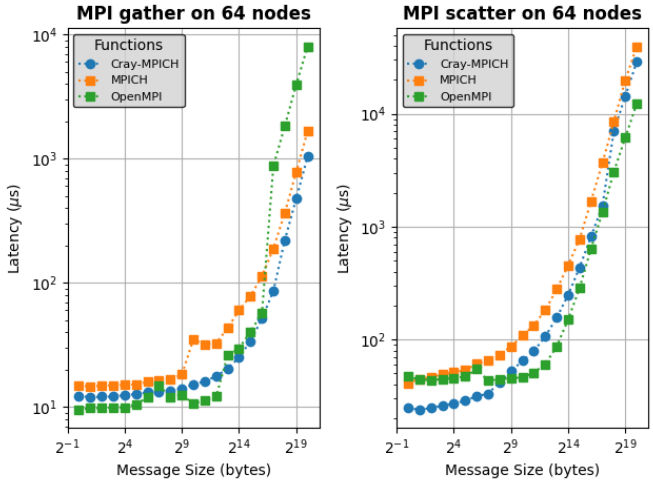

Figure 2: Latency incurred by single-rooted MPI collective operations (`gather` and `scatter`) on Perlmutter GPU nodes.

5.1 Single rooted collectives

Single-rooted MPI operations, such as `gather` and `scatter`, which involve one root process communicating with multiple destination processes. These are often implemented via straightforward tree-based algorithms and have relatively fewer tuning parameters. Figure 2 presents the latency measurements obtained for MPI `gather` and `scatter` calls executed on 64 nodes across a range of message sizes.

For both `gather` and `scatter`, Cray-MPICH consistently achieves the lowest latency across all tested message sizes. In the case of `gather`, OpenMPI exhibits latencies comparable to Cray-MPICH for smaller message sizes (up to approximately 4KB). However, beyond 4KB, OpenMPI latency increases significantly, surpassing even MPICH. In contrast, MPICH demonstrates a more consistent latency profile, remaining slightly higher than Cray-MPICH across all message sizes.

For the `scatter` operation, both MPICH and OpenMPI exhibit higher latencies than Cray-MPICH at smaller message sizes, with MPICH performing slightly better than OpenMPI in several cases. However, the latency differences among the three implementations diminish as the message size increases. For the largest message size, OpenMPI outperforms both MPICH and Cray-MPICH.

In summary, Cray-MPICH provides the most efficient implementation of both `gather` and `scatter` operations out-of-the-box, clearly outperforming MPICH and OpenMPI. Notably, OpenMPI incurs substantial latency penalties for the `gather` operation at larger message sizes.

5.2 Multi rooted collectives

In multi-rooted MPI collective operations, where multiple root processes communicate simultaneously with multiple destinations. We focus on the `alltoall` and `allgather` routines. Figure 3 presents the measured latencies incurred by the three MPI implementations under consideration.

Cray-MPICH consistently achieves the lowest latency for both `alltoall` and `allgather` operations across all tested message sizes. Although Cray-MPICH occasionally exhibits irregular latency spikes at certain message sizes, these latencies remain notably lower than those observed with MPICH and OpenMPI. MPICH displays higher latencies compared to Cray-MPICH, although the latency differences decrease for larger message sizes. The OpenMPI implementation exhibits a more complex pattern. For lower message sizes up to 1KB, OpenMPI's latencies are significantly higher. However, between 1KB and 4KB, the implementation appears to be optimized, resulting in latencies that are even lower than Cray-MPICH for certain values. For larger message sizes, the average of latencieswmeasured shows OpenMPI's latency between that of Cray-MPICH and

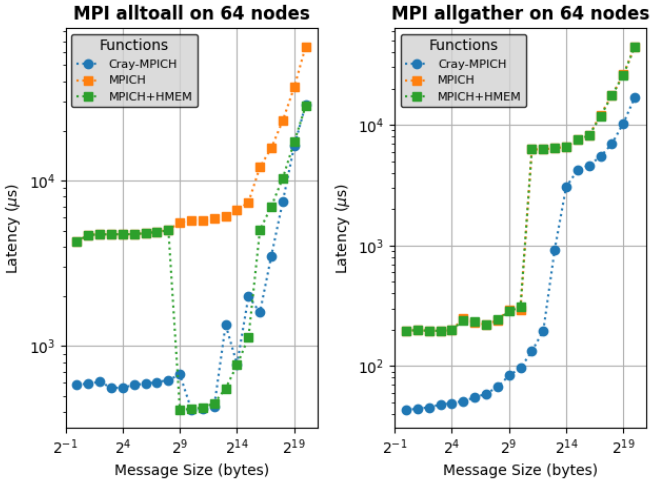

Figure 4: Impact of enabling HMEM (RDMA) on MPICH performance for multi-rooted MPI collectives (`alltoall` and `allgather`) executed on 64 Perlmutter GPU nodes.

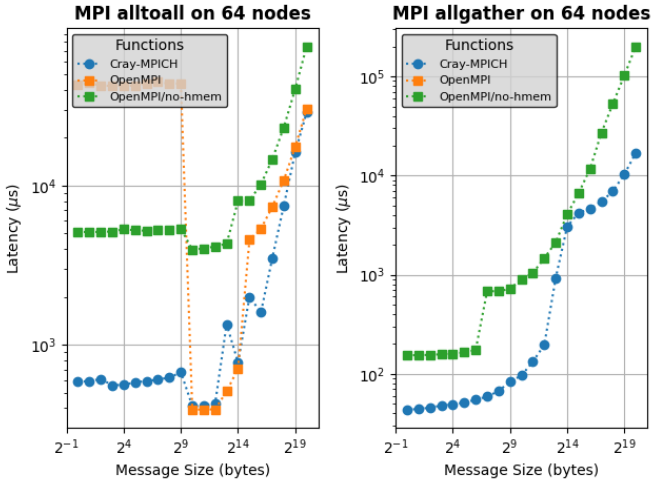

Figure 5: Impact of disabling HMEM (RDMA) in OpenMPI for multi-rooted MPI collectives (`alltoall` and `allgather`) executed on 64 Perlmutter GPU nodes.

MPICH. Examination of the performance of individual iterations shows that OpenMPI's behavior suffers from high variability.

In the case of the `allgather` operation, the relative differences among implementations are less pronounced compared to `alltoall`. Nevertheless, Cray-MPICH still provides the lowest latency, especially for message sizes up to 4KB. At larger message sizes, OpenMPI exhibits significantly higher latencies compared to both Cray-MPICH and MPICH.

Our analysis of collective operation performance reveals that there is no single MPI implementation that stands out as a clear winner across all operations. Instead, each implementation has its strengths and weaknesses. Cray-MPICH demonstrates consistent behavior across all collective operations, making it a reliable choice. MPICH's performance is comparable to Cray-MPICH, with some implementations coming close to matching its performance. OpenMPI excels in certain areas, such as gather and scatter operations, but its performance is more variable and chaotic for alltoall operations. Overall, the choice of MPI implementation for collective operations depends on the specific use case and requirements.

5.3 Impact of Heterogeneous Memory (HMEM) on MPICH and OpenMPI Performance

We found that by default MPICH module on Perlmutter does not enable the heterogeneous memory (HMEM) option, which facilitates Remote Direct Memory Access (RDMA). RDMA enables direct data transfers between GPU buffers without intermediate copies to host memory, potentially improving communication performance significantly. This feature can be enabled at runtime by setting the environment variable `MPIR_CVAR_CH4_OFI_ENABLE_HMEM`. This will be included by default in a future maintenance when modulefiles are updated.

As Figure 4 illustrates, enabling HMEM does not noticeably affect the performance of the `allgather` operation, it significantly improves the performance of the `alltoall` operation, particularly for message sizes of 512 bytes and larger. In fact, with HMEM enabled, MPICH's `alltoall` performance closely matches Cray-MPICH performance for these larger message sizes. Since enabling HMEM did not produce performance regressions in any tested case, we strongly recommend that system administrators enable this option by default to leverage RDMA capabilities and that the default behavior of the library be reviewed.

In contrast to MPICH, OpenMPI enables HMEM and RDMA by default. Given that OpenMPI's `alltoall` performance (as shown in Figure 3) was comparable to MPICH with HMEM disabled, we investigated the effect of explicitly disabling HMEM in OpenMPI. The results of this experiment are presented in Figure 5.

Disabling HMEM in OpenMPI, as shown in Figure 5, improves the performance of the `alltoall` collective for smaller message sizes, especially for those message sizes that previously exhibited unusually high latencies. However, even with this improvement, OpenMPI's performance remains significantly worse than Cray-MPICH across these message sizes. Disabling HMEM however also impacts the performance of OpenMPI for medium to larger message sizes that incurred very low latencies when HMEM was enabled.

In summary, enabling HMEM (RDMA) significantly enhances MPICH's performance for multi-rooted collective operations, bringing it closer to the performance of Cray-MPICH. Conversely, OpenMPI's default configuration (with HMEM enabled) does not deliver comparable performance improvements for small message sizes, and disabling HMEM partially mitigates performance issues only for small message sizes in the `alltoall` procedure.

5.4 Upstream OpenMPI

In our efforts to understand the performance behavior of OpenMPI across different release versions, we discovered that the upstream (development) branch of OpenMPI has recently incorporated optimized implementations for certain collective operations.

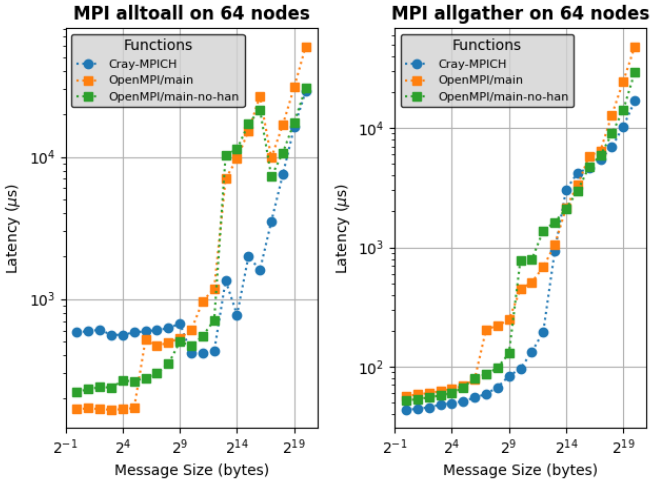

Figure 6: Performance of upstream OpenMPI (main branch, commit `ad35fd5c3b` from April 30, 2025) for multi-rooted MPI collectives (`alltoall` and `allgather`) executed on 64 Perlmutter GPU nodes.

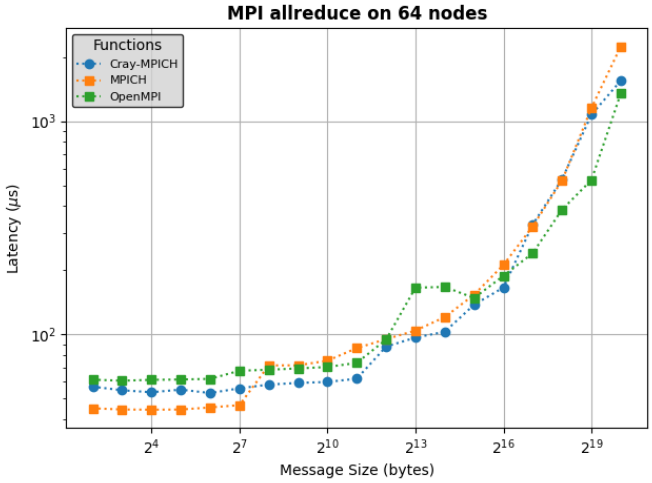

Figure 7: Latency incurred by the MPI `allreduce` operation executed on 64 Perlmutter GPU nodes.

Figure 6 presents the performance results of the multi-rooted collectives (`alltoall` and `allgather`) when using the upstream OpenMPI main branch (specifically, commit `ad35fd5c3b` from April 30, 2025). As shown in the figure, the upstream version significantly improves the performance of the `alltoall` collective compared to the last release version as of writing this paper. Notably, for small message sizes, the upstream OpenMPI implementation even outperforms Cray-MPICH.

In addition to enabling RDMA (HMEM) and using the upstream OpenMPI branch, we identified another configuration option that impacts performance: the Hierarchical Autotuned collectives for Networks (HAN). Previous studies [18] have noted that the HAN module does not currently account for GPU buffers, resulting in suboptimal performance when GPU memory is involved. Consequently, we disabled the HAN module using the following environment variable setting:

```
export OMPI_MCA_coll=^han
```

Figure 6 demonstrates the positive impact of disabling HAN, especially for small message sizes in the `alltoall` operation. In contrast, the effect of disabling HAN is less pronounced for the `allgather` operation, which exhibits relatively consistent performance with the upstream OpenMPI branch compared to the release version.

In summary, at the time of writing this paper, the upstream OpenMPI development branch contains significant performance improvements for multi-rooted collective operations involving GPU buffers. These enhancements are expected to become available in future OpenMPI release versions. In a later section, we provide further details on how we identified and isolated the specific updates in OpenMPI responsible for these performance differences between the release and upstream versions.

5.5 Collective updates in OpenMPI

To narrow down the specific updates incorporated into upstream OpenMPI that resulted in performance improvements for the `alltoall` collective, we developed a script that recursively executes the `git bisect` operation until we pinpoint the exact commit responsible for the improvement. A compact version of the script is provided in Listing 8.

We marked the upstream OpenMPI commit `ad35fd5c3b` from April 30, 2025 as the "GOOD" commit (showing improved performance), and the release version tagged as `5.0.7` as the "BAD" commit (prior to improvement). By recursively iterating through the commits generated by the `git bisect` command, we successfully identified the specific commit responsible for the observed performance improvement in the `alltoall` collective.

Ultimately, our analysis revealed that commit `da14c12` [1], which updates MPI implementations, is the key factor behind the improved collective performance observed in the upstream OpenMPI repository.

5.6 Multi rooted collectives with Computation

In the previous sections, we analyzed MPI collective operations that primarily involve communication. Such communication-only collectives typically appear less frequently in application workflows, or at least are overlapped with computation for optimal performance. In this section, we shift our focus to an MPI collective operation that explicitly combines communication and computation: the `allreduce` routine.

For the `allreduce` performance evaluation presented here, we apply all the optimization strategies identified in the previous sections. Specifically, for MPICH we enable the HMEM runtime variable, and for OpenMPI we use the optimized upstream version discussed previously with the HAN collective module disabled. Figure 7 presents the performance results for the MPI `allreduce` collective executed on 64 Perlmutter GPU nodes.

```
#!/bin/bash
set -euxo pipefail
BASE="$(pwd)}"
SRC_DIR="${BASE}/src"
INSTALL_BASE="${SCRATCH}/openmpi/"
git clone --recursive omp-repo.git $SRC_DIR
cd $SRC_DIR
# Configure, build, and install OpenMPI
build_ompi() {
 git clean -fdx
# Non release versions need to
# generate configure script
 ./autogen.pl
 ./configure \
   CC=gcc CXX=g++ \
   --prefix="${install_base}/$1" \
   --with-cuda="${CUDA_HOME}" \
   --with-libfabric=$libfabric_path \
   --with-pmix=/usr \
   --other-configure-options-as-necessary
   make && make  install
}
# Run MPI benchmark test
#and determine good/bad commit
run_mpi_test() {
 # Point to the right OpenMPI install.
 # Point to current compilers and libs
 export PATH=... && LD_LIBRARY_PATH=...
 #Build and run the benchmark and Check
 # If the commit is good continue running
 if grep -qi "success"; then
   git -C "${SRC_DIR}" bisect bad
   return 1
 else
   git -C "${SRC_DIR}" bisect good
   return 0
 fi
}
# Latest commit as initial reference
# We know the latest commit is a good commit
initial_commit=$(git rev-parse --short HEAD)
build_ompi "${initial_commit}"
run_mpi_test "${initial_commit}" || true
cd "${SRC_DIR}" # Start bisect
git bisect start
git bisect good "${initial_commit}"
git bisect bad v5.0.7
# Automated bisect loop
while true; do
 current_commit=$(git rev-parse --short HEAD)
 configure_build_install "${current_commit}"
 if ! run_mpi_test "${current_commit}"; then
   echo "Continuing bisect: GOOD commit."
 else
   echo "Continuing bisect: BAD commit."
 fi
done
```

Figure 8: Pseudo code for git bisect.

As illustrated in Figure 7, all three MPI implementations exhibit a similar overall latency trend. However, the relative performance ranking among the implementations varies significantly depending on the message size. Specifically, for small message sizes up to 128 bytes, MPICH achieves the lowest latency among the three implementations. For intermediate message sizes (from 128 bytes to 64KB), Cray-MPICH consistently outperforms both MPICH and OpenMPI. Finally, for large message sizes exceeding 64KB, OpenMPI demonstrates the lowest latency.

In summary, the performance of the `allreduce` collective depends strongly on the message size and the MPI implementation. MPICH excels at small message sizes, Cray-MPICH provides superior performance at intermediate message sizes, and OpenMPI is most efficient at large message sizes.

References

[1]
[2] AMD. AMD EPYC 7763.
[3] Beebe, M., Gayatri, R., Gott, K., Lavely, A., Haseeb, M., Cook, B., and Chen, Y. A performance analysis of GPU-aware MPI implementations over the Slingshot-11 interconnect. In *2024 IEEE High Performance Extreme Computing Conference (HPEC)* (2024), pp. 1–7.
[4] Bruck, J., Ho, C.-T., Kipnis, S., Upfal, E., and Weathersby, D. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems 8*, 11 (1997), 1143–1156.
[5] Bureddy, D., Wang, H., Venkatesh, A., Potluri, S., and Panda, D. K. Omb-gpu: A micro-benchmark suite for evaluating mpi libraries on gpu clusters. In *Recent Advances in the Message Passing Interface* (Berlin, Heidelberg, 2012), J. L. Träff, S. Benkner, and J. J. Dongarra, Eds., Springer Berlin Heidelberg, pp. 110–120.
[6] Castain, R. H., Hursey, J., Bouteiller, A., and Solt, D. Pmix: Process management for exascale environments. *Parallel Computing 79* (2018), 9–29.
[7] Cray HPE. Cray MPICH.
[8] De Sensi, D., Di Girolamo, S., McMahon, K. H., Roweth, D., and Hoefler, T. An in-depth analysis of the Slingshot interconnect. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis* (2020), pp. 1–14.
[9] Enterprise, H. P. HPE Slingshot interconnect. *Hewlett Packard Enterprise. Retrieved Jan* (2024).
[10] Forum, M. P. Mpi: A message-passing interface standard. Tech. rep., USA, 1994.
[11] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting* (Budapest, Hungary, September 2004), pp. 97–104.
[12] Gropp, W., Lusk, E., Thakur, R., Balaji, P., Gillis, T., Guo, Y., Latham, R., Raffenetti, K., and Zhou, H. MPICH user's guide. *Argonne National Laboratory* (February 2025).
[13] Grun, P., Hefty, S., Sur, S., Goodell, D., Russell, R. D., Pritchard, H., and Squyres, J. M. A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency. In *Hot Interconnects* (2015), IEEE Computer Society, pp. 34–39.
[14] Guo, Y., Raffenetti, K., Zhou, H., Balaji, P., Si, M., Amer, A., Iwasaki, S., Seo, S., Congiu, G., Latham, R., et al. Preparing MPICH for exascale. *The International Journal of High Performance Computing Applications* (2025), 10943420241311608.
[15] Khorassani, K. S., Chen, C.-C., Ramesh, B., Shafi, A., Subramoni, H., and Panda, D. K. High performance MPI over the Slingshot interconnect. *Journal of Computer Science and Technology 38*, 1 (2023), 128–145.
[16] Kim, J., Dally, W. J., Scott, S., and Abts, D. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture* (2008), pp. 77–88.
[17] Lewis, J. G., and Simon, H. D. The impact of hardware gather/scatter on sparse gaussian elimination. *SIAM Journal on Scientific and Statistical Computing 9*, 2 (1988), 304–311.
[18] Luo, X., Wu, W., Bosilca, G., Pei, Y., Cao, Q., Patinyasakdikul, T., Zhong, D., and Dongarra, J. Han: A hierarchical autotuned collective communication framework. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)* (2020), IEEE, pp. 23–34.
[19] NVIDIA. NVIDIA AMPERE A100.
[20] Shehata, A., Naughton, T., Bernholdt, D., and Pritchard, H. Bringing HPE Slingshot 11 support to Open MPI. *Concurrency and Computation: Practice and*

Experience 36, 22 (Oct. 2024).