Poster Lightning Introductions

September 28th, 2023

CUP ECS

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF TENNESSEE CHATTANOOGA

Simulating the Impact of Network Interference on Collective Performance

Nicole Avans[1], Riley Shipley[1], and Anthony Skjellum[1,2] 1. UTC 2. TTU

Background

The escalating network traffic interference on high-performance clusters has prompted a critical examination of its impacts on both discrete applications and specific collectives. This research focuses on simulating network interference and load imbalance while measuring execution data to analyze the latency tolerance of MPI collective communication functions. To gather and manipulate detailed execution metrics, we utilized multiple LLNL tools including: Caliper, Adiak, Hatchet, and Thicket. Development of these benchmarks is an ongoing endeavor with the aim of illustrating whether certain collectives offer more robust performance under these operating conditions.

Caliper

Caliper is a versatile performance profiling framework. It allows for the dynamic instrumentation of code to gather a wide range of performance data. Caliper provides a flexible and lightweight approach to measuring program behavior. [1]

Adiak

Adiak is a dynamic performance instrumentation and measurement tool. Users may insert performance measurement points into their code without the need for recompilation; particularly useful for obtaining fine-grained performance data from existing applications. [2]

Hatchet

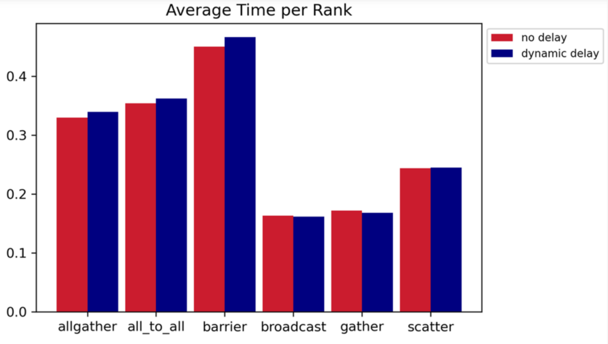
Hatchet is a hierarchical performance data visualization tool designed to depict performance metrics in a tree-like structure, making it particularly useful for understanding how different components of a program contribute to overall performance. [3]

Thicket

Thicket is a performance data analysis and visualization tool that complements Hatchet. It specializes in visualizing and analyzing hierarchical performance data, making it easier to identify patterns, anomalies, and areas for optimization in complex codebases. [4]

Results

- The introduction of delays before executing collective functions leads to varying impacts on the communication completion time. The plot below illustrates the completion times for specific collective functions, comparing the average times per rank with no delays and dynamic delays.
- When tested within the ExaMPI implementation, simulated network interference resulted in a modest, variable impact on execution times. This finding underscores the nuanced influence of network traffic interference on overall collective performance.



The average time spent executing each rank with no delay, and with a dynamic delay. This figure includes the added buffer times.

Methods

- To simulate network interference and load imbalance, delays are introduced prior to the execution of collective functions. These delays can either be configured as a set static delay or as a random selection from a set delay interval. To begin the study, we are evaluating simulated interference on ExaMPI, a modern research subset implementation of MPI-4.
- The benchmarks are fully customizable using a variety of parameters, implemented as command line arguments. Users may specify the number of ranks, epochs, message size, and a variety of delay options, in addition to specifying which collectives or set of collectives to evaluate.

```
hatchet v2023.1.0

1.785 main
├─ 0.341 all_to_all
│  └─ 0.332 mpi_alltoall.cc::MPI_Alltoall
│     └─ 0.308 toolbox.hpp::start_wait_complete
│        ├─ 0.014 basiccommunicator.cc::start_request
│        │  └─ 0.005 blockingprogress.cc::post_request
│        ├─ 0.264 blockingprogress.cc::wait_for_complete
│        │  └─ 0.254 request.cc::wait_for_complete
│        └─ 0.010 toolbox.hpp::finalize_request
├─ 0.373 allgather
│  └─ 0.364 mpi_allgather.cc::MPI_Allgather
│     └─ 0.339 toolbox.hpp::start_wait_complete
│        ├─ 0.014 basiccommunicator.cc::start_request
│        │  └─ 0.005 blockingprogress.cc::post_request
│        ├─ 0.295 blockingprogress.cc::wait_for_complete
│        │  └─ 0.284 request.cc::wait_for_complete
│        └─ 0.011 toolbox.hpp::finalize_request
├─ 0.451 barrier
│  └─ 0.442 mpi_barrier.cc::MPI_Barrier
│     └─ 0.422 toolbox.hpp::start wait complete
```

Thicket compiles multiple Caliper files to view various ranks execution time simultaneously.

Future Research

- The present work establishes a highly extensible benchmarking suite poised to evaluate diverse MPI implementations and to support a greater range of collective communication algorithms. This enables more comprehensive evaluations to pinpoint the most effective algorithms in various scenarios.
- Looking forward, our foremost goal is enhancing customization of delays to allow for flexible comparison of disparate algorithms. Currently delays are introduced before execution and are neatly encapsulated in their own Caliper regions, while future versions will also support delays at multiple execution points. Additional custom parameters will be implemented to allow users to augment how random delays are distributed.

Conclusions

- The architecture of the collective benchmarks program established thus far in the course of this study provide a foundation for rapid, focused expansion of utility with meticulous control over a useful selection of variables.
- This extensibility empowers researchers to adapt and extend the benchmarks to more comprehensively explore collective communication performance under varying network conditions for one or more selected communication algorithms.
- The goal of this and continuing research is to identify more resilient and efficient communication strategies in distributed computing environments. The benchmark suite can be enhanced with additional parameters for fine-grain control over minute details to better suit specific needs depending on the codebase, enabling bespoke analysis for even the most specific of circumstances.
- As traffic increases on high performance clusters, it will impact the performance of certain applications. Data gathered on the Quartz cluster at LLNL exhibited variable execution time depending on simulated latency. Monitoring which methods accommodate these delays most efficiently will assist in developing programs with critical performance needs on shared systems.

References

[1] D. Boehme *et al.*, "Caliper: Performance Introspection for HPC Software Stacks," *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, USA, 2016, pp. 550-560, doi: 10.1109/SC.2016.46.

[2] LeGendre, Matthew P., Bohme, David, and Poliakoff, David Z. Adiak. Computer Software. https://github.com/llnl/adiak. USDOE National Nuclear Security Administration (NNSA). 28 Aug. 2019. Web. doi:10.11578/dc.20191025.1.

[3] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. 2019. "Hatchet: pruning the overgrowth in parallel profiles." *SC'19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* Association for Computing Machinery, New York, NY, USA, 2019, Article 20, 1–21. doi: 10.1145/3295500.3356219

[4] Stephanie Brink, Michael McKinsey, David Boehme, Connor Scully-Allison, Ian Lumsden, Daryl Hawkins, Treece Burgess, Vanessa Lama, Jakob Lüttgau, Katherine E. Isaacs, Michela Taufer, and Olga Pearce. 2023. "Thicket: Seeing the Performance Experiment Forest for the Individual Run Trees." *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*. Association for Computing Machinery, New York, NY, USA, 281–293. doi: 10.1145/3588195.3592989

Acknowledgements and Funding

Thanks to the ExaMPI team, our CUP-ECS collaborators, and the Caliper, Adiak, Hatchet, and Thicket teams at LLNL for their technical support. This work was performed with partial support from the National Science Foundation under Grant Nos. 1918987 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.

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's National Nuclear Security Administration.

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF TENNESSEE CHATTANOOGA

MPI Implementation Profiling for Better Application Performance

Riley Shipley and Garrett Hooten | SimCenter

Introduction

Until this point, profiling and optimization efforts in the HPC community have been focused almost entirely on applications themselves, neglecting the impact that using a poorly optimized communication library within an application can have.

This work demonstrates two effective methods of profiling and optimizing communication libraries that each produced considerable improvements when used on ExaMPI, a research focused sub-implementation of the MPI library created by UTC in collaboration with Auburn University, UNM, and the University of Alabama [1].

To collect profiling data, this study uses Caliper, a flexible profiling and tracing library that allows users to add custom profiling regions to their application code. Caliper can also be built with an MPI target in order to produce timing data for every MPI binding used over the application run, a crucial feature for this project [2].

Comparison-based Profiling

- Set up an MPI-aware profiling tool for use with two MPI implementations (e.g., Caliper)
- Run an MPI application multiple times with each implementation using the profiling tool
- Compare average run times for each MPI call to determine which areas can improve

Internal Profiling

- Use a region-profiling and visualization tool to mark regions of the MPI implementation
- Run an MPI application to collect profiling data and create a visual profile
- Inspect the profile for repetitious/problematic behavior, and focus efforts in those areas

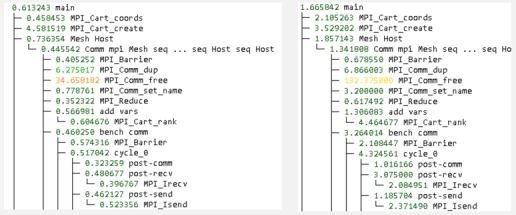
Analysis

In our comparison-based method, the resulting data from running COMB with both ExaMPI and Spectrum MPI was funneled through Hatchet, which allows for easy comparison of timing data [3].

The example shown is achieved by dividing Spectrum's completion times by ExaMPI's for all the same regions, resulting in a reliable way to discern large performance differences at-a-glance.

Applying the internal profiling method to COMB, we observed multiple non-blocking calls taking longer than expected to complete. Digging deeper into the visualization revealed that these calls were forced to wait on the same lock as the progress thread (left), a problem that was fixed soon thereafter by adding a separate incoming message queue (right).

Comparison-based profiling – before changes (left) and after (right)

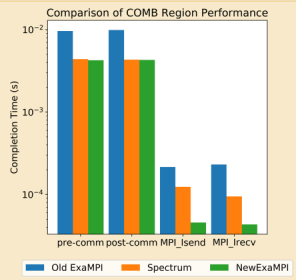


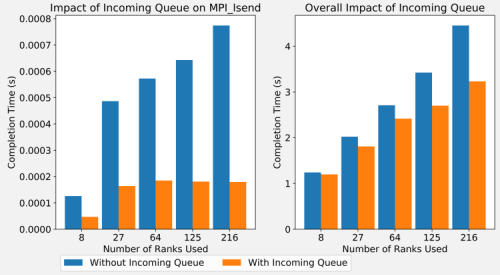
Ratio of Spectrum MPI performance to ExaMPI, values > 1 indicate superior ExaMPI performance

Internal profiling visualization – before (left) and after (right)

A visualization of two versions of ExaMPI completing an MPI_Isend call. Note the overlapping lock regions on the left, which do not exist in the improved version.

Results





In the comparison-based study, Hatchet analysis revealed that many regions of the benchmark, including non-MPI regions, were performing much worse in ExaMPI than Spectrum MPI. After resolving the issue, which stemmed from a process-scheduling issue, performance improved over 4.5x on MPI_Irecv and Isend calls, and matched Spectrum's performance on non-MPI regions.

For internal profiling, we resolved the lock-waiting issue by implementing separate queues for incoming messages and messages currently being processed, which reduced Isend completion time by more than 4x and overall application completion time by more than 1.25x in the largest test case of 216 processes.

Both methods were evaluated using COMB [4] on LLNL's Lassen cluster [5] with a maximum of 16 ranks per node.

Conclusions

Presented here are two powerful methods for assessing performance of communication libraries and finding areas where improvement is needed the most.

Comparison-based profiling excels at quickly identifying which regions need the most attention, at the application level, while internal profiling is more fine-grained and exposes patterns of behavior and performance anomalies to the observer .

Both offer differing but complementary methods for initiating investigations into the performance of communication libraries.

Acting on the insights these methods provide will improve the performance of not only the communication interface itself, but also the application.

References

[1] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. Exampi: A modern design and implementation to accelerate message passing interface innovation. In Juan Luis Crespo-Mariño and Esteban Meneses-Rojas, editors, *High Performance Computing*, pages 153–169, Cham, 2020. Springer International Publishing.

[2] David Boehme, Todd Gamblin, David Beckingsale, Peer-Timo Bremer, Alfredo Gimenez, Matthew LeGendre, Olga Pearce, and Martin Schulz. Caliper: Performance introspection for hpc software stacks. In *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 550–560, 2016.

[3] Abhinav Bhatele, Stephanie Brink, and Todd Gamblin. Hatchet: Pruning the overgrowth in parallel profiles. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Jason Burmark and USDOE National Nuclear Security Administration. Comb, 7 2018.

[5] "Lassen." *HPC @ LLNL*. https://hpc.llnl.gov/hardware/compute-platforms/lassen. (Jan. 2023)

Acknowledgements

Many thanks to Anthony Skjellum, Derek Schafer, Thomas Hines, et al for their expert advice on how best to integrate our profiling methods with ExaMPI, as well as allowing us to use ExaMPI for this in the first place.

Many thanks to David Boehme for his patience in answering our many questions on Caliper and for adding support specific to our experiment.

Finally, thanks to Olga Pearce and the rest of Livermore National Laboratory for partnering with us throughout this study.

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF TENNESSEE CHATTANOOGA

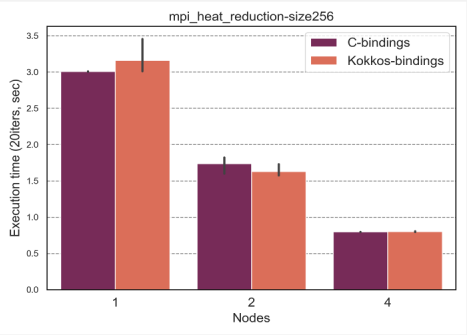
View-Aware Message Passing Through the Integration of Kokkos and ExaMPI

Evan Drake Suggs[1], Jan Ciesko[2], Stephen L. Olivier[2], and Anthony Skjellum[3] 1. UTC 2. Sandia National Labs 3. TTU

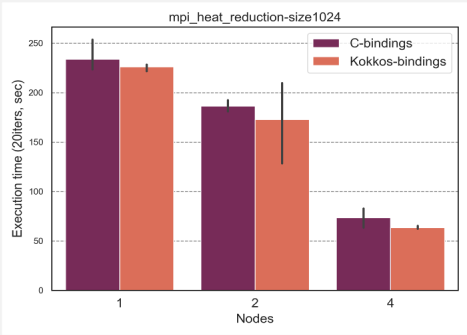
Background

- The Kokkos programming library provides in-memory advanced data structures, concurrency, and algorithms to support advanced C++ parallel programming while MPI provides a widely used message passing model for inter-node communication [1,2,4].
- The primary feature of Kokkos discussed in this poster is the View, a datatype similar to a tensor [1]. A key feature of Kokkos is that Views can be assigned to specific memory and execution spaces [1, 2].
- The purpose of this poster is to integrate Kokkos within the ExaMPI implementation via a MPI Extension to obtain development benefits over having the two running separately in the same program. This is both for MPI itself and for applications that use MPI+Kokkos. First-class Kokkos objects can be passed directly to the new functions.
- The primary advantage will be the time saved during development rather than at runtime, as the underlying model is the same.
- This work has implications separate from Kokkos, such as how true MPI-based C++ bindings differ from classic C bindings.
- Previous work combining both Kokkos and traditional versions of MPI has yielded interesting results. For example, Khuvis et al. [2] have a shown a speedup of GEMM code and the Graph500 benchmark using their version of MPI+Kokkos.

Results



256 element heat reduction test on 1, 2 and 4 nodes for MPI+Kokkos traditional versus new bindings.



1024 element heat reduction test on 1, 2 and 4 nodes for MPI+Kokkos traditional versus new bindings.

The above tests uses the heat reduction test found in Kokko's own tutorials. One with the traditional C-based MPI bindings to handle Views, and another with our C++-enhanced Kokkos-bindings.The primary goal was not a significant change in performance, but roughly equal performance for both bindings. However, our bindings had generally lower execution times, especially for increased size with 1024 elements.

These tests were run on the Blake testbed at Sandia National Laboratories. Blake contains 40 24-core Intel Xeon Platinum nodes with the Intel Omnipath Interconnect. 500 runs were averaged on Views with from 64 to 32768 elements

Methods

```
// old method
    int *recv_buf = (int*) malloc(n * sizeof(int));
    MPI_Recv(recv_buf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    Kokkos::View<int*> recv_check(recv_buf, n);
// new method
    Kokkos::View<int*> A("New Method View", n);
    MPI_Kokkos_Recv<Kokkos::View<int*>, int>(A, n, MPI_INT, 1, 0, MPI_COMM_WORLD);
// newer method
    Kokkos::View<int*> A("New Method View", n);
    MPI::Recv<Kokkos::View<int*>, int>(A, 1, 0, MPI_COMM_WORLD);
```

A comparison of existing and improved methods for sending the contents of a Kokkos View using MPI

- In the new method, all that is required to receive a View is to create an existing View of the correct size and copy to it.
- In the old method, the View's underlying pointer must be accessed by the programmer, opening up the possibility for memory leaks
- MPI_Kokkos_Send sends the underlying array as a Payload. Its counterpart, MPI_Kokkos_Recv receives this Payload, then wraps it back into a View object (either a View pointer or copying to full array with a performance penalty).
- This model is used for all the other MPI bindings as well. Note that this method is only compatible with contiguous arrays currently.
- Most of the bindings follow a traditional MPI binding except MPI_Kokkos_Get_Dims which returns every extent (size of each individual dimension) of the View as a single integer vector.
- The newer method no longer requires the MPI Datatype nor dimensions from the user.

Conclusions

This project integrated MPI and Kokkos by implementing several MPI bindings to use Kokkos Views as their primary buffers, keeping the C++ nature of the Kokkos View for users.
After implementing the bindings for this project, we found that the new bindings' performed similarly to the old bindings's.

- Wider range of MPI functions such as All-To-All, Scatter, and Gather.
- Change from MPI_Kokkos_X to overloading existing MPI functions.
- More device-specific support (i.e., MPI_Send<View, class, Device>).
- Testbed for new functions, such as byte-mapping-based transports, rather than traditional datatype-based transports.
- Reconciling the differences between MPI and Kokkos methods of dealing with non-contiguous data, and add non-contiguous View support.
- A creation of new backends to increase speed for specific types of Views (i.e., Views on GPUs, non-contiguous, etc.).

References

[1] H. Carter Edwards and Christian R. Trott. 2013. Kokkos: Enabling Performance Portability Across Manycore Architectures. In 2013 Extreme Scaling Workshop (xsw 2013). 18–24. https://doi.org/10.1109/XSW.2013.7

[2] Samuel Khuvis, Karen Tomko, Jahanzeb Hashmi, and Dhabaleswar K. Panda. 2020. Exploring Hybrid MPI+Kokkos Tasks Programming Model. In 2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM). 66–73. https://doi.org/10.1109/PAWATM51920.2020.00011

[3] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. 2020. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. In High Performance Computing, Juan Luis Crespo-Mariño and Esteban Meneses-Rojas (Eds.). Springer International Publishing, Cham, 153–169.

[4] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. IEEE Transactions on Parallel and Distributed Systems 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

Acknowledgements and Funding

Thanks to Riley Shipley and Derek Schafer for reviewing this work, along with the ExaMPI team and the Kokkos team for technical support. Additional thanks to Dr. Joseph Dumas II and Dr. Michael Ward.

Funding in part is acknowledged from these NSF Grants 191897, 21501020, and 2201497, as well as 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.

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

Center for Understandable Performant Exascale Communication Systems

Collective-Optimized FFTs

Evelyn Namugwanya[1], Amanda Bienz[2], Derek Schafer[2], Anthony Skjellum[1] | [1]TTU/UTC, [2]UNM

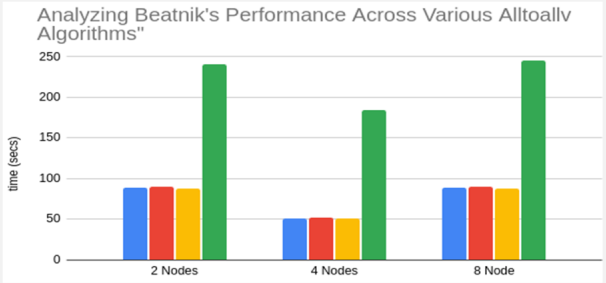
Introduction

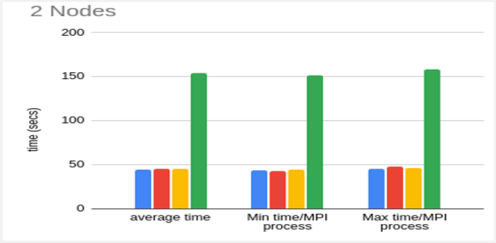
- HeFFTe is an FFT library designed for Exascale, dominated by MPI_Alltoallv communication.
- Key goal: make MPI_Alltoallv faster and in turn make FFTs such as HeFFTe and Beatnik faster.
- Beatnik is a benchmark for global communication based on Pandya and Shkoller's 3D fluid interface "Z-Model" in the Cabana/Cajita mesh framework.
- Beatnik is bottlenecked by HeFFTe; it's a good driver app.
- MPI Advance is a collection of MPI extension libraries showcasing new APIs or optimizations of MPI APIs.
- MPI Advance includes faster MPI_Alltoallv variants.

Methodology

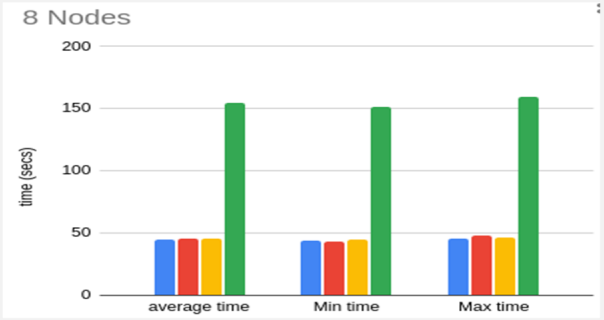
- We used Caliper to instrument HeFFTe to get the timing for GPU Alltoallv routines. We used MPI_Wtime to analyze the difference in Beatnik's performance with various Alltoallv algoritms. We modified the HeFFTe library and replaced the OpenMPI Alltoallv with MPI Advance's Alltoallv. Our goal is to see which setup is fastest in various situations and vs. baseline performance.
- Previously, we tested different setups of collective communication: Non-blocking Alltoallv, Alltoallv pairwise, Multi-pair blocking exchange, and Multi-pair test exchange.
- Currently testing the following GPU-aware Alltoallv algorithms:
  - **GPU pairwise alltoallv:** pairwise exchange.
  - **GPU multi-pair blocking exchange:** combines Non-blocking Alltoallv and Pairwise Alltoallv, uses Waitall.
  - **GPU multi-pair nonblocking exchange:** Uses Waitany.
  - **Alltoallv:** the OpenMPI Alltoallv

Results





Timing GPU Alltoallv Routines within the Application



Timing GPU Alltoallv Routines within the Application

Key:

- gpu_multi-pair exchange
- gpu_multi-pair non-blocking exchange
- gpu_pairwise
- gpu_Alltoallv

Conclusions

1. Low bars mean less time taken and high bars mean more time taken (in seconds)
2. The three GPU Alltoallv algorithms seem to follow a comparable pattern.
3. The GPU algorithms outperform the standard version applied in HeFFTe, Beatnik.
4. In general, the Beatnik FFT application operates at a threefold greater speed.
5. When we make adjustments to Alltoallv and enhance HeFFTe's performance since Beatnik invokes HeFFTe, we notice a substantial increase in Beatnik's speed, about three times faster.
6. The next step will be to utilize persistent collectives with GPU optimizations

References

1. https://github.com/mpi-advance
2. https://github.com/CUP-ECS/beatnik/
3. https://icl.utk.edu/files/publications/2022/icl-utk-1558-2022.pdf
4. https://hpc.llnl.gov/software/development-environment-software/tau-tuning-and-analysis-utilities
5. https://software.llnl.gov/Caliper/

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF NEW MEXICO

Evaluating the Viability of LogGP for Modeling MPI Performance with Non-contiguous Datatypes on Modern Architectures

Nicholas Bacon and Patrick Bridges | Department of Computer Science

Introduction

Modern architectures and communication systems software include complex hardware, communication abstractions, and optimizations that make their performance difficult to measure, model, and understand. The communication abstractions such as MPI's derived datatypes are a core component of modern high-performance computing (HPC) communication systems. These abstractions are designed to ease programmability and allow the communication system to efficiently send, receive, and compute on (e.g., reduce) complex data structures. Unfortunately, even highly-optimized versions of these abstractions have wildly-varying performances when using realistic application data structures on modern GPU-based systems. In our initial tests, even highly-optimized datatype engines such as MPICH/Yaksa and TEMPI often performed significantly (5%-50%) worse than simple application data packing kernels when working on realistic application data layouts. Importantly, we have not found any case where datatypes outperformed simple application packing kernels when doing GPU to GPU communication.

We modified versions of the existing Netgauge communication performance measurement tool and LogGOPS performance model to accurately characterize the communication behavior of modern hardware, MPI abstractions, and implementations. This includes analyzing their ability to model both GPU-aware communication in different MPI implementations and quantifying the performance characteristics of different approaches to non-contiguous data communication on modern GPU systems. We apply these techniques to quantify the performance of different implementations and optimization approaches to non-contiguous data communication on a variety of systems, demonstrating that modern communication system design approaches can result in widely varying and difficult-to-predict performance variation, even within the same hardware/communication software combination.

Results

How accurate are LogGOP and LogGP for contiguous buffers?

LogGP models tend to overestimate ping-pong times for flat buffers. Despite this, the LogGPS and LogGOP models demonstrate a remarkable ability to accurately capture the overall trends and shape in measured communication performance.

How accurate are LogGOP and LogGP for non-contiguous buffers?

Model accuracy is better for non-contiguous buffers, including the vector datatypes tested. We believe this is because LogGP models more accurately capture data packing and unpacking times than GPU communication latencies. capture the overall trends and shape in measured communication performance.

Overheads per byte generally higher with non-contiguous buffers

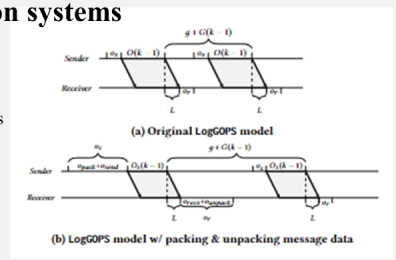
The overheads of the send and receive of non-contiguous data is generally much higher than that of the flat and contiguous data types. The model captures datatype packing and unpacking overheads better than communication cost. For some MPIs, particularly Spectrum and Mvapich2, packing and unpacking overheads can dominate the total cost of data transmission.

Acknowledgements

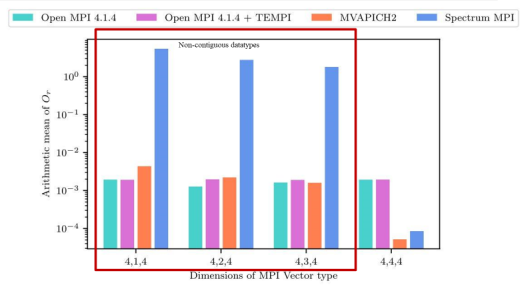
- National Science Foundation OAC-2103510
- U.S. Dept. of Energy Award DE-NA0003966
- Kurt Ferreira and Scott Levy at Sandia National Laboratories

Mapping LogGOP to GPU communication systems

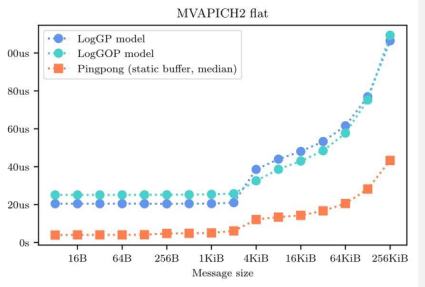
Figure 1a shows a simple example of sending two back-to-back k- byte messages between a Sender and Receiver. Figure 1b shows a simple example of sending two back-to-back k-byte messages using our simple extension of the LogGOPS model. The principal difference between this model and the original LogGOPS model is that, unlike the original model, we explicitly account for the costs associated with moving data between host and device memory and assembling non-contiguous data into Evaluating the Viability of LogGP for Modeling MPI Performance with contiguous message buffers.



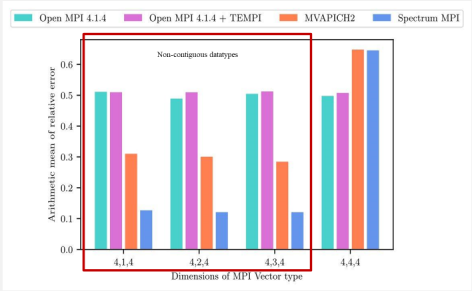
To capture the impact of these costs, we model the per-message overheads (os and or) and per-byte overhead (Os and Or) to include: (i) the time required for sending messages to (o_{send}), and receiving messages from (o_{recv}), the network; and (ii) the costs associated with preparing non-contiguous data for transmission (o_{pack}) and the costs associated with processing non-contiguous data after reception (o_{unpack})



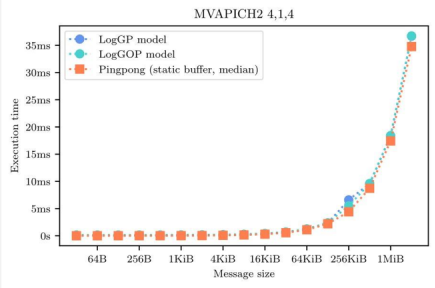
LogGOP overhead of the receiving times with non-contiguous (red box) and contiguous datatype all buffer sizes



LogGP and LogGOP modeled performance versus ping pong latency with a flat buffer



LogGOP accuracy with non-contiguous (red box) and contiguous datatype ping pong latency averaged across all buffer sizes



LogGP and LogGOP modeled performance versus ping pong latency with a non-contiguous datatype

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF NEW MEXICO

Beatnik: A Prototype High Performance Parallel Interface Benchmark

Jason Stewart and Patrick Bridges | Computer Science

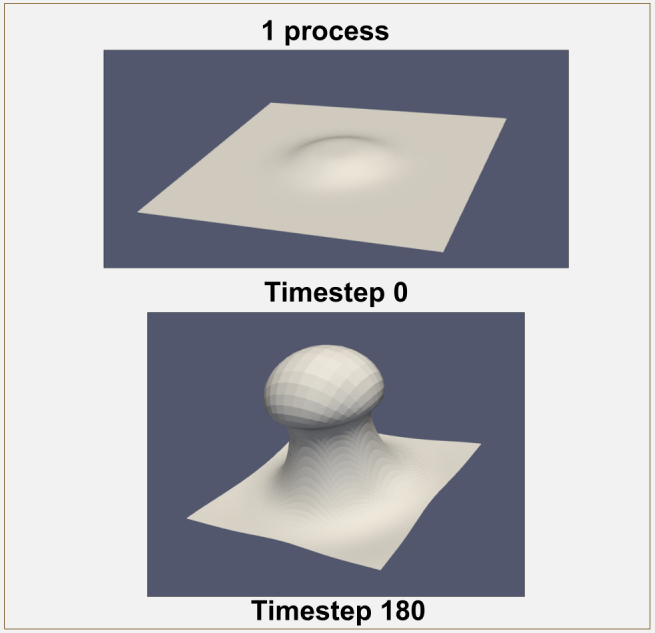
Purpose

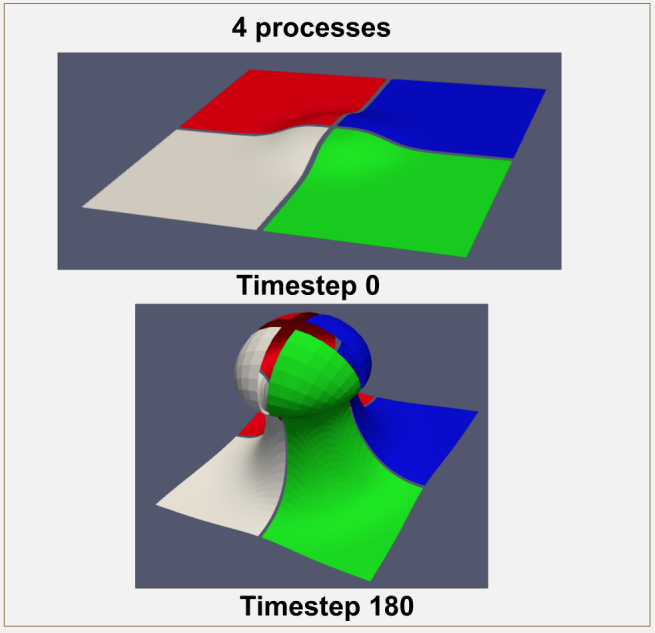
- Kokkos solver for the 2D interface between two fluids undergoing severe interface roll-up
- Effective global communication benchmark, step toward coupled code benchmark
- The mathematics for this model originate from a paper by Gavin Pandya and Steve Shkoller [1].

Approach

- Parallelize brute force high-order model computation of Birkhoff-Rott velocity using ring pass algorithm
- Asymptotic run time is $O(n^4)$, but reduced by a constant factor of the number of processes
- Working on cutoff-based approach to reduce complexity of far field force calculation

High-order produces identical results serially and in parallel





Future Work

- Finish cut-off based high-order model
- Integrate Beatnik with fluid solvers
- Benchmark communication performance on new systems

Conclusions

- Integrating Beatnik with other fluid solvers is only reasonable if they have comparable computational complexities.
- Ideally the high-order *z*-model should run in n^3 time for this integration to be feasible

Acknowledgements

This work was funded by the PSAAP program. We would like to thank the UNM Center for Advanced Research Computing, and the Department of Energy for providing the research computing resources used in this work.

References

1.Gavin Pandya and Steve Shkoller. “3D Interface Models for Rayleigh-Taylor Problems”. 2023. DOI: https://doi.org/10.48550/arXiv.2201.04538

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF NEW MEXICO

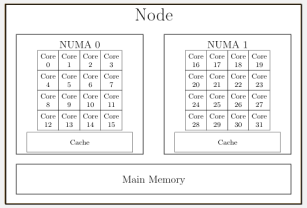
Towards Partitioned Locality Aware Neighborhood Collectives

Gerald Collom and Amanda Bienz | Dept. of Computer Science

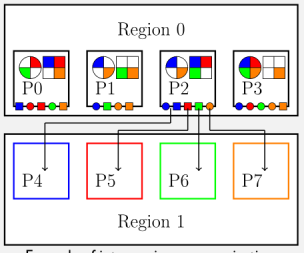
Locality Aware Neighborhood Collectives

Background

- Irregular communication bottleneck
- Typical approach: point-to-point, integrated optimizations
  - Harder, less portable optimizations
- Persistent neighborhood collectives
  - Full info of communication
  - Portable optimizations behind MPI
- Optimization/adoption stagnation
- **Locality regions**:
  - Socket, node, group, distance in system topology
- **Locality Awareness**
  - Utilize locality to increase cheaper communication and reduce more expensive communication



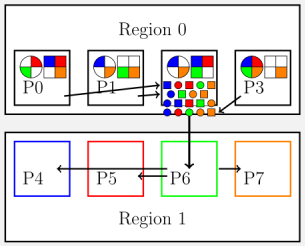
Example NUMA node



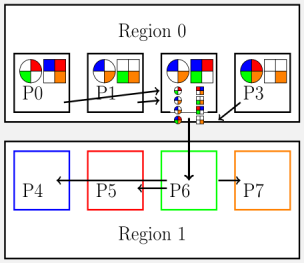
Example of inter-region communication

Approach

- **Hypre**: tested application
  - Algebraic Multigrid (AMG)
  - SpMV, SpMM
  - Implementations added through MPI Advance
- **3-step Aggregating Method**:
  1. Locally aggregate all messages destined to another region
  2. Single inter-region message sent between region pairs
  3. Locally send to destination process
- **Deduplicating Method**:
  - Reduce size of inter-region messages
  - Uses extended interface with unique identifiers

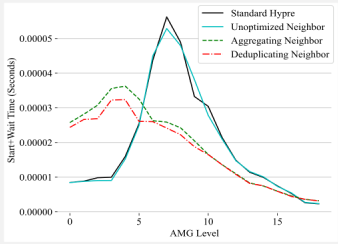


Aggregating method example

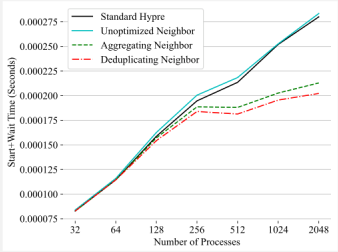


Deduplicating method example

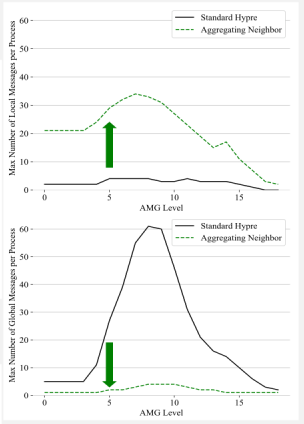
Results



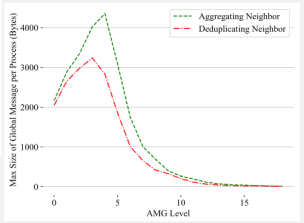
Communication cost per AMG level



Strong scaling communication cost



Local/inter-region message tradeoff

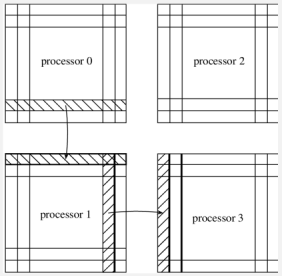


Deduplication message size reduction

Partitioned MPI

Background

- Regular Halo Exchange
- Packing
  - thread divergence
  - synchronization before sending
- Partitioned MPI:
  - Partition large messages
  - Ready: send parts as soon as they are ready
  - Early communication
  - Early work



Halo exchange. image source: https://www.researchgate.net/figure/Two-stages-of-a-halo-exchange-in-two-dimensions-Note-the-corner-is-transmitted-from_fig3_7660293

Methods (Work in Progress)

- Comb:
  - Regular halo exchange
  - Benchmark: test various methods and parameters of communication plus execution
- Added persistent communication method
- Adding partitioned communication method
  - Used MPIPCL, included directly in Comb
  - Tested without threading
  - Implementing early communication with threading

Future Work: Partitioned Locality Aware Neighborhood Collectives

- SpMM in Hypre during setup
  - Large number of large messages (entire matrix rows)
- Locality-aware aggregation has shown to improve SpMM
- Partitioned communication has shown to improve large message communication
  - Multiple threads pack and communicate parts of the message asynchronously
- Natural combination:
  - Remove duplicate communicated values
  - Distribute communication work across threads
- Benchmarks for systems with GPUs have shown copying data to CPUs and using all cores to communicate outperforms GPUDirect or copying to a single CPU for irregular communication

Center for Understandable Performant Exascale Communication Systems

THE UNIVERSITY OF ALABAMA® | College of Engineering

Pulse:
A GPU Halo Exchange Benchmark

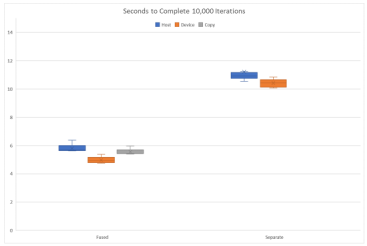
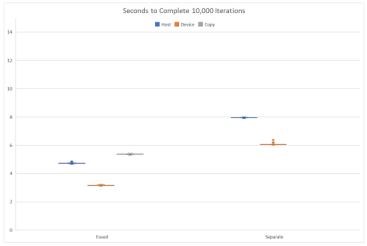
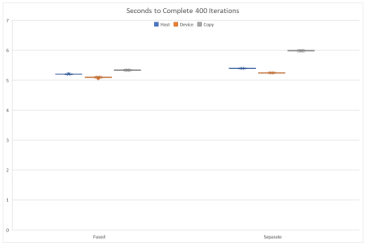
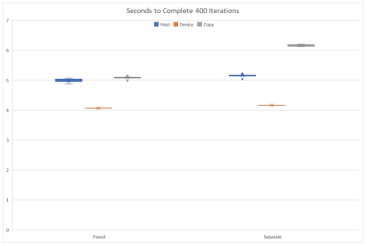
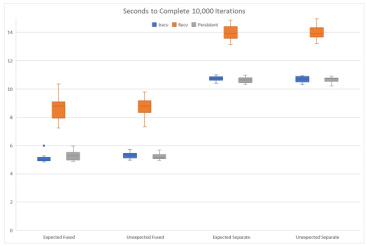
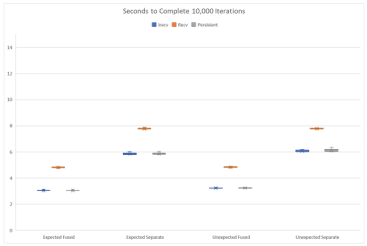
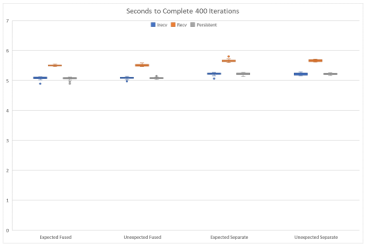
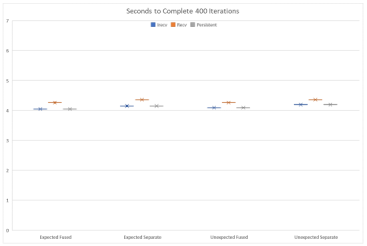
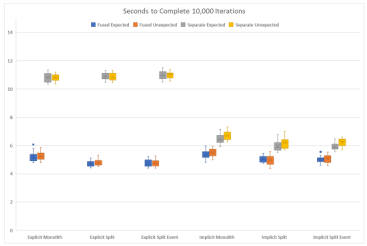
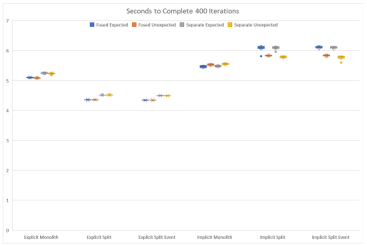
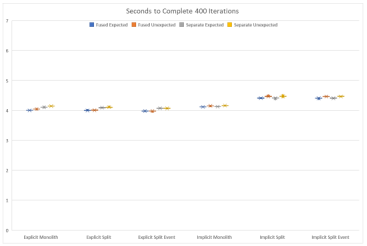
Thomas Hines, Tony Skjellum, Patrick Bridges, and Purushotham Bangalore

Pulse explores various ways to perform a halo exchange for a GPU-based stencil-grid application. It supports both CUDA and HIP as well as communication/computation overlap.

Pulse Modular Structure
Environment – grid setup and compute
Packer – packing and unpacking of halos
Sender – Transferring of halos to other ranks
Executor – Calling the other components in the right order

Pulse Setup Options
Local grid dimensions
Process grid dimensions
Number of variables per cell
Halo depth
Compute kernel length

Halo Exchange Options
Fused vs. Separate packing kernels
Host vs. Device packing buffers vs. copy
Pre-posted receives vs. not
MPI_Irecv vs. MPI_Recv vs. MPI Persistent
Explicit vs. Implicit corner exchange
Monolith vs. Split compute kernels

50x50x50 Local Grid 6x6x6 Process Grid Lassen	50x50x50 Local Grid 4x4x4 Process Grid Tioga	200x200x200 Local Grid 6x6x6 Process Grid Lassen	200x200x200 Local Grid 4x4x4 Process Grid Tioga	3 variables per cell 3 deep halo
				Device packing buffers and a fused packing kernel performs the best.
				A blocking MPI receive performs the worst. MPI Irecv and persistent are the same performance.
				A split compute kernel is only sometimes better than a single compute kernel