Persistent and Partitioned MPI for Stencil Communication

Gerald Collom
Department of Computer Science
University of New Mexico
Albuquerque, USA
geraldc@unm.edu

Jason Burmark
Lawrence Livermore National Lab
Livermore, USA
burmark1@llnl.gov

Olga Pearce
Lawrence Livermore National Lab
Livermore, USA
pearce8@llnl.gov

Amanda Bienz
Department of Computer Science
University of New Mexico
Albuquerque, USA
bienz@unm.edu

***Abstract*—Many parallel applications rely on iterative stencil operations, whose performance are dominated by communication costs at large scales. Several MPI optimizations, such as persistent and partitioned communication, reduce overheads and improve communication efficiency through amortized setup costs and reduced synchronization of threaded sends. This paper presents the performance of stencil communication in the Comb benchmarking suite when using non-blocking, persistent, and partitioned communication routines. The impact of each optimization is analyzed at various scales. Further, the paper presents an analysis of the impact of process count, thread count, and message size on partitioned communication routines. Measured timings show that persistent MPI communication can provide a speedup of up to 37% over the baseline MPI communication, and partitioned MPI communication can provide a speedup of up to 68%.**

***Index Terms*—MPI, stencil exchanges, communication, persistent communication, partitioned communication**

I. Introduction

Stencil computations dominate a wide range of parallel applications, including scientific simulations based on solving partial differential equations [1] and numerical solvers such as geometric multigrid [17]. They are established as one of the most prevalent patterns in high performance computing [2]. Stencil operations consist of solving a linear method across a structured mesh. At each iteration, all mesh cells are updated, with the new value at each cell dependent on that of all neighboring cells. In parallel, the mesh is decomposed into subdomains for each participating process, with each process holding boundary cell data from processes with neighboring mesh regions (ghost cells) in order to compute its own boundary cell updates. A parallel stencil operation consists of updating values locally before exchanging updated ghost regions during each iteration. As process counts increase, these boundary exchanges dominate performance costs. As a result, efficient boundary exchanges are crucial for performant and scalable stencil codes.

There are several common implementations for boundary exchanges, including hand-rolled point-to-point communication and neighborhood collectives with Cartesian process grids. Regardless of approach, potential optimizations for boundary exchanges have been introduced with each major release of the MPI standard. For example, persistent communication introduced in MPI 3 has shown to improve communication costs [12], [7], [14] by incurring setup and overhead costs once while allowing cheaper iterations of communication afterward. Furthermore, MPI may choose to optimize persistent exchanges through additional overheads in the initialization, such as tag matching. MPI 4 provides an additional optimization opportunity through partitioned communication, allowing threads to concurrently work on a persistent message. This paper presents a novel study of the trade-offs between standard non-blocking, persistent, and partitioned communication throughout iterative boundary exchanges.

Comb, a stencil benchmarking tool from Lawrence Livermore National Laboratory (LLNL) analyzes the performance trade-offs between various stenciled boundary exchanges. This work adds both persistent and partitioned MPI optimizations into this widely-used benchmark to analyze their impact. The optimizations are made publicly available within Comb for application developers to use when analyzing comparable communication strategies.

The contributions of this paper include the following.

1) Optimize stencil exchanges with persistent communication
2) Further optimize stencil exchanges with partitioned communication
3) Demonstrate up to 68% speedup over existing MPI methods in Comb when scaled across thousands of processes.
4) A performance study of persistent and partitioned communication for stencil exchanges across varying process, CPU core and thread counts.
5) Expand the stencil communication benchmark Comb to allow application programmers to test persistent and

partitioned communication strategies.

The following sections are organized as follows. Section II provides background on stencil codes and the persistent and partitioned communication optimizations. Related works are discussed in Section III. In Section IV, the implementation of the two communication optimizations are detailed. Results of performance analysis of persistent and partitioned MPI in the stencil communication benchmark Comb are provided in Section V. Finally, conclusions and future directions are presented in Section VI.

II. Background

Stencil codes split a structured mesh across a Cartesian process grid, requiring processes to iteratively perform local computation followed by boundary exchanges. Each process uses initial boundary data for each neighboring process to update all local mesh cells. After local updates are complete, local boundary cell values are packed into a contiguous buffer and communicated to neighboring processes. Boundary cell values from neighboring process are also received and unpacked from a contiguous buffer into the local mesh. After all updates of ghost cells are complete, successive iterations can occur. This exchange of data, also known as a halo exchange, is further detailed in Section II-A.

A. Halo Exchanges

The pattern of boundary data to be exchanged at each iteration is dependent on the structure of the problem at hand. For example, a 2D stencil requires exchanging edges and corners with up to 8 neighbors, as exemplified in Figure 1. In 3D, this is often extended to a 27-point stencil, in which faces, edges, and corners are communicated to 26 neighboring processes.

While some data is contiguous, such as boundaries on the top and bottom of each process in a row-wise data layout, other exchanges require communication of non-contiguous data, such as the left and right edges. Multiple methods for exchanging non-contiguous data have been explored, including using MPI Datatypes [10], [19]. However, on CPU-only supercomputers, packing costs are typically outweighed by communication at scale. As a result, this work is focused on optimizing the exchange of data, and simply packs each message into a contiguous buffer outside MPI before communication. OpenMP threads are used to perform packing, allowing the benefit of utilizing all CPU cores.

A standard boundary exchange is detailed in Algorithm 1. All messages are first packed into a contiguous buffer using OpenMP threads, before being exchanged with neighboring processes. Finally, data is unpacked into the local domain, to be used during successive steps of computation.

B. Persistent MPI and Partitioned MPI

Persistent communication allows communication patterns to be defined and initialized once and repeatedly executed to avoid repeated overhead costs. This optimization is well suited

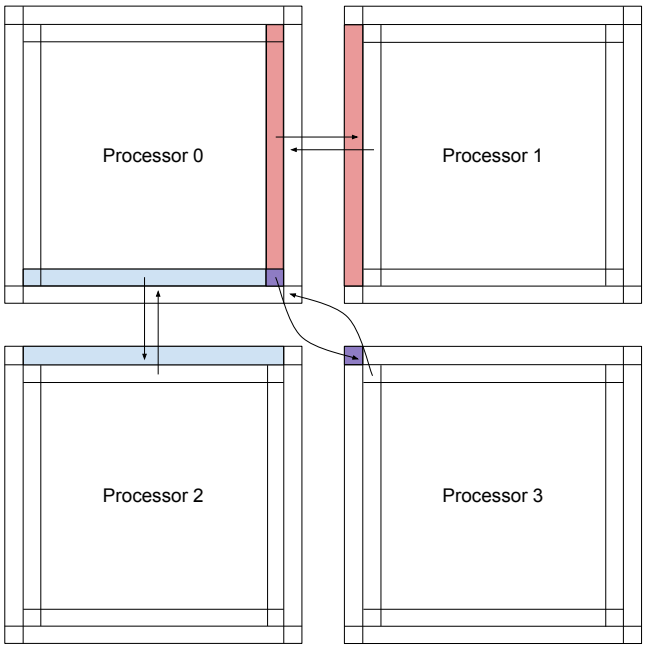

Fig. 1: Shown are example exchanges for a single process in a 2D regular halo exchange. Process 0 sends values for every cell along its right edge (red and purple) to Process 1 to fill its left edge of ghost cells (red). All cells along the bottom edge (blue and purple) are sent to Process 2 to fill its top edge of ghost cells (blue). Finally, the purple corner is sent to Process 3 to fill its top left ghost cell. In return, Processes 1, 2 and 3 send their boundary cell values (white) to Process 0 to update its ghost cells.

Algorithm 1: `Exchange`

Input: `msgs_info, requests`
{message information (process pairs, data sources), requests storage array}

parallel region {OpenMP Region}
  // Copy data from mesh to contiguous message buffer
  `pack(msgs_info)`

// Begin communication
for $i \leftarrow 0$ **to** n_{send} **do**
  MPI_Isend(msgs_info[i], requests[i])
for $i \leftarrow 0$ **to** n_{recv} **do**
  MPI_Irecv(msgs_info[$i + n_{send}$],
   requests[$i + n_{send}$])

// Wait on messages
`MPI_Waitall(requests)`

parallel region
  // Copy values from message buffer into mesh
  `unpack(msgs_info, msgs_info)`

for applications like stencil codes with iterative communication. A persistent communication pattern consists of calling a method such as `MPI_Send_init` once to initialize communication, providing all arguments that are typically provided to `MPI_Isend`. However, rather than beginning communication, `MPI_Send_init` returns a handle for a persistent request. Each iteration of communication is then performed by passing the persistent request to persistent exchange routines, such as `MPI_Start` and `MPI_Wait`. As a result, initialization costs are only incurred once and then amortized over all subsequent exchanges.

Partitioned communication builds upon persistent communication, partitioning each message across multiple threads and allowing for portions of a message to be independently sent and received without synchronizing on the entirety of the message. This optimization allows threads to work concurrently and for communication of parts of a large message to begin as soon as they are ready, without waiting for the entire message data to be ready. This early communication can reduce network contention by utilizing the network early rather than sending all data at once. Further, partitioned communication allows for early work, in which successive computation (or unpacking) can begin as soon as any portion of a message arrives.

Partitioned communication can work well for large messages, but does incur some overheads. The flag `MPI_THREAD_MULTIPLE` is required as multiple threads all perform communication at the same time, which can cause slowdowns that vary greatly among versions of MPI [23]. Additionally, for a given message, all partitions must be equal in size and there must be an equal number of partitions on the sending and receiving sides. While padding can be added to the last message for those with sizes that do not evenly divide partition count, this padding can be complex.

Similar to persistent communication, partitioned sends first require an initialization call, `MPI_Psend_init`, requiring standard communication arguments along with the number and size of partitions for subsequent iterations. Each iteration, data is exchanged through a single instance of `MPI_Pstart` per message, followed by invoking `MPI_Pready` for each partition to mark the portion of data is ready to be sent. Finally, `MPI_Wait` is called to await the arrival of all partitions. If attempting to perform early work, `MPI_Parrived` can be used to test the arrival of an individual partition, which, if arrived, is ready for use in computation.

III. Related Works

Persistent MPI communication was introduced in the MPI 1.1 standard. Since then, work has been done to implement and optimize persistent communication [12], [14]. As an older optimization, adoption is not uncommon in practice, although far from ubiquitous. Persistent communication has also been used within the context of neighborhood collectives. [16]

Several works have tackled the design of partitioned communication within MPI [9], [5], [3], [22], [15]. Additional works have explored methods to accurately model and benchmark partitioned communication, and analyzed the potential benefits of this optimization [5], [11], [21], [8]. Partitioned MPI communication has also been evaluated in comparison to other approaches to MPI+Threads [23]. The combination of partitioned and collective communication is another direction that has previously been explored [13].

Separately, prior work looked into the optimization of stencil code communication without the use of persistent or partitioned MPI [18], [20]. Specifically, optimization techniques such as node-awareness have been successfully applied to stencil communication [18]. MPI Datatypes have also seen use in stencil computations [10], [19]. The stencil communication benchmark Comb can also test on heterogeneous architectures and prior work has explored optimizations in this case [6], although the work in this paper focuses on CPU only systems.

IV. Implementations

Standard stencil codes use non-blocking sends and receives at each iteration. Both persistent and partitioned communication optimizations are suitable for this structured, iterative exchange. Because the stencil computation and communication is iterative, the use of persistent communication provides a natural optimization to incur an overhead cost once at initialization and utilize it in every following iteration of communication. In the case where multiple threads are used to efficiently pack data into contiguous buffers, partitioned communication is a natural fit, as these same threads can be used to parallelize communication with partitions. Furthermore, exchanges of entire faces of a 3D domain often result in very large messages, which can be optimized when split across many available CPU cores and communicated asynchronously.

A. Persistent MPI

Algorithm 2: `Persistent Init`

Input: `msgs_info, requests`

{message information (process pairs, data sources), requests storage array}

```
// Initialize messages
for i ← 0 to n_send do
    MPI_Send_init(msgs_info[i],
     requests[i])
for i ← 0 to n_recv do
    MPI_Recv_init(msgs_info[i + n_send],
     requests[i + n_send])
```

Persistent stencil communication is split into three separate methods: initialization, iterative exchange, and destruction. First, each process initializes the persistent exchange, as described in Algorithm 2. This consists of initializing each persistent send and receive. Then, during each iteration of the persistent boundary exchange described in Algorithm 3, all processes pack data in an equivalent fashion to the standard approach in Algorithm 1. However, all non-blocking

Algorithm 3: `Persistent Exchange`

Input: `msgs_info, requests`
{message information (process pairs, data sources), requests storage array}

parallel region {OpenMP Region}
- // Copy data from mesh to contiguous message buffer
- `pack(msgs_info)`

// Begin communication
`MPI_Startall(requests)`

// Wait on messages
`MPI_Waitall(requests)`

parallel region
- // Copy values from message buffer into mesh
- `unpack(msgs_info)`

Algorithm 4: `Persistent Destroy`

Input: `requests` {requests storage array}

for $i \leftarrow 0$ **to** $n_{recv} + n_{send}$ **do**
- `MPI_Request_free(requests[`i`])`

communication is now replaced with `MPI_Startall` and `MPI_Waitall`. Finally, when all iterations are complete, the persistent request handles must be destroyed, as shown in Algorithm 4.

B. Partitioned MPI

Algorithm 5: `Partitioned Init`

Input: `n_parts, msgs_info, requests`
{number of partitions, message information (process pairs, data sources), requests storage array}

// Initialize messages
for $i \leftarrow 0$ **to** n_{send} **do**
- `MPI_Psend_init(n_parts, msgs_info[`i`], requests[`i`])`

for $i \leftarrow 0$ **to** n_{recv} **do**
- `MPI_Precv_init(n_parts, msgs_info[`$i + n_{send}$`], requests[`$i + n_{send}$`])`

Partitioned communication further optimizes stencil operations beyond the persistent optimizations, allowing for threads to each communicate a portion of the data. This optimization consists of first initializing the partitioned exchange, as described in Algorithm 5, during which partitioned MPI initialization calls are instantiated for each message. The arguments provided are the same in Algorithm 2, except an additional argument for the number of partitions of the message. Then, every iteration of the partitioned stencil code performs a boundary exchange among all threads, as shown in Algorithm 6. In this method, the packing thread calls

Algorithm 6: `Partitioned Exchange`

Input: `msgs_info, requests`
{message information (process pairs, data sources), requests storage array}

// Begin communication
`MPI_Startall(requests)`

parallel region {OpenMP Region}
- // Copy data from mesh to contiguous message buffer
- `pack(msgs_info)`
- // Mark partition of current thread as ready
- `MPI_Pready(partition)`

// Wait on messages
`MPI_Waitall(requests)`

parallel region
- // Copy values from message buffer into mesh
- `unpack(msgs_info)`

Algorithm 7: `Partitioned Destroy`

Input: `requests` {requests storage array}

for $i \leftarrow 0$ **to** $n_{recv} + n_{send}$ **do**
- `MPI_Prequest_free(requests[`i`])`

`MPI_Pready` immediately after packing its portion of the data, marking the specified message partition as ready to send. For simplicity, all process then wait for all messages to complete before unpacking. However, this approach could be further optimized through early work, in which each thread could unpack partitions of a message on arrival. After all iterations of halo exchanges have completed, the partitioned request handle is destroyed, as shown in Algorithm 7.

V. Results

All optimizations described in Section IV have been added throughout the Comb halo exchange benchmarking suite from LLNL. The partitioned communication implementation we used was the MPIX Partitioned Communication Library (MPIPCL)[1] and included it directly within Comb. The presented timings are acquired from Comb simulations with three mesh variables, a single cell wide exchange boundary and a periodic problem mesh in all directions so that no special case edge regions occur in the decomposition. Additionally, the following Comb options were disabled: `per_message_pack_fusing` and `message_group_pack_fusing`. All presented timings account for asynchrony of processes and timer inaccuracies through multiple methods. Before any timing is initialized, a barrier occurs to synchronize all processes. Further, timer precision is accounted for, and all measurements are timed over 1000 exchanges, with the average per-exchange cost extracted. Finally, the impact of nearby jobs is lessened as

[1]https://github.com/mpi-advance/MPIPCL

each test is performed three separate times on the system and the average time of the three runs is used.

Standard, persistent, and partitioned exchanges are analyzed on the Quartz supercomputer, an Intel SMP architecture at LLNL, using the default system MPI Mvapich2 version 2.3.7. While Quartz nodes each contain 36 cores, all performance measurements in this section use 32 processes per node to keep all process counts powers of 2. Further, all tested thread counts are also powers of 2, and all stencil sizes were chosen such that halo exchange sizes are also powers of 2. As a result, partitioned communication is performed in all cases without the need for padding, but it should be noted that to use partitioned communication as currently defined by the MPI standard, padding would need to be added to account for uneven partition sizes. Finally, our tests used 2 OpenMP threads per core to take advantage of hyperthreading.

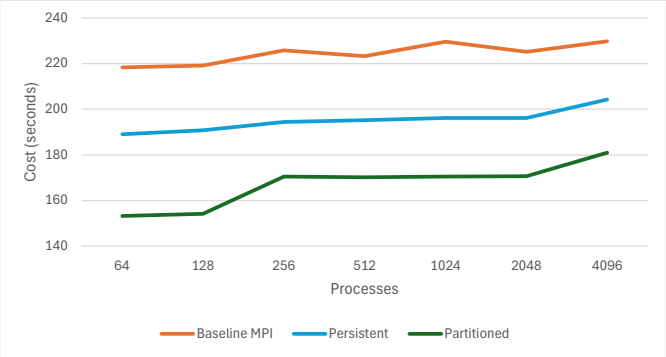

Fig. 2: Weak scaling of the tested communication strategies for message sizes of $524,288$ doubles.

In Figure 2, we present a weak scaling study from 64 to 4096 processes, with messages sizes of $524,288$ double-precision floats (doubles) at each scale. All cases in this study are run with 32 ranks per node, 32 active cores per node, and 2 threads per core. At the largest scale of 4096 processes, persistent communication obtains a 12.5% speedup over Comb's baseline MPI implementation, while partitioned communication further improves performance by another 14.5% to a total of 27% over the baseline. Note, performances for all three methods trend upwards at higher scales and appear to slightly approach each other, suggesting additional communication or contention costs occur at larger scales.

In Figure 3, we present a strong scaling study of the communication costs of the different policies from 128 to 4096 processes, using equivalent per-node process, core, and thread counts to the weak scaling study. This test simulates a 2048^3 cell problem, which results in message sizes of $262,144$ doubles at the smallest scale, decreasing as the number of processes grows. Persistent communication achieves speedup of 37% over the baseline at 2048 processes, but performs equivalently to standard approaches at both the smallest and largest process counts. Partitioned communication outperforms the other methods, achieving speedups over the baseline of

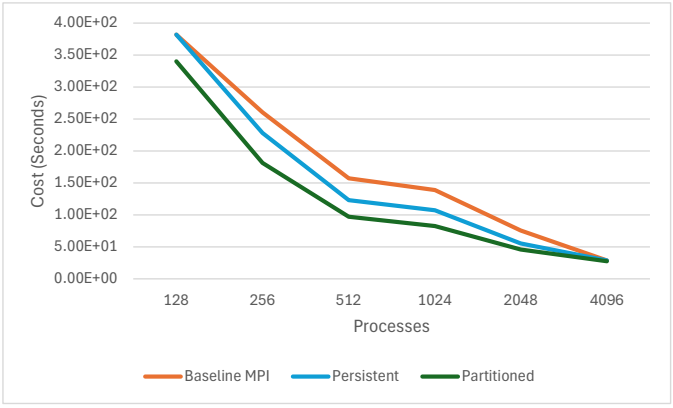

Fig. 3: Strong scaling study of the communication strategies for a 2048^3 cell mesh. Message sizes start at $262,144$ doubles.

12%, 68%, and 4.4% at 128, 1024, and 4096 processes respectively. Note, partitioned communication provides less of an advantage for smaller message sizes. This is a contributing factor to the diminishing speedup from the use of partitioned communication at higher process scales for the same problem size.

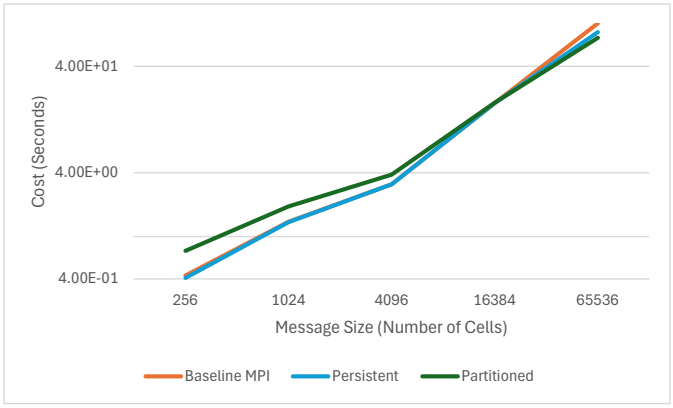

Fig. 4: A comparison of the communication strategies on 4096 processes as message size is increased from 768 to 196608 doubles.

Figure 4 displays the cost of halo exchanges with each of the three policies on 4096 processes, with varying halo exchange sizes. The range of message sizes varies from 256 to 65536 mesh cells consisting of 3 doubles each. At the smallest message scale tested, persistent communication performed similarly to the baseline, while partitioned communication performs significantly worse, with the baseline performing 73% faster. As message sizes increase, however, persistent and partitioned communication performance eventually overtake for a speedup over the baseline of 21% for persistent and 37% for partitioned at the largest tested message size.

Figure 5 shows the cost of halo exchanges across 64 nodes, with varying numbers of MPI processes per node. All cases consist of 32 active cores per node and 64 OpenMP threads per node. As the number of MPI ranks per node increases,

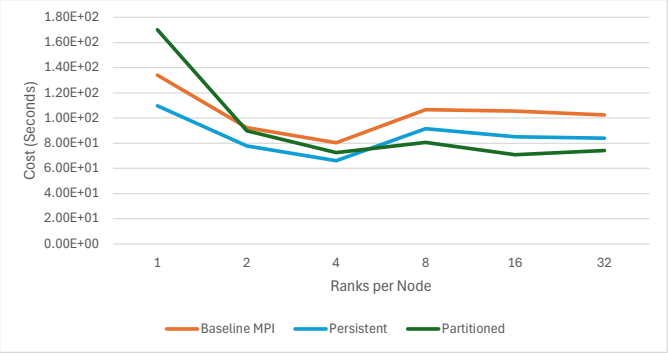

Fig. 5: A study of the effect of the number of MPI ranks used per node on the performance of the communication strategies for a $2048 \times 4096 \times 4096$ mesh.

OpenMP threads per rank decrease accordingly. As a result, when using fewer MPI processes per node, thread launch overheads increase. Furthermore, in the standard and persistent cases, fewer cores are utilized during MPI communication. All test contain a constant problem size of $2048 \times 4096 \times 4096$ cells.

This study shows that persistent communication outperforms the baseline for every tested number of ranks per node, with a speedup of around 20% in each case. Additionally, at one rank per node, partitioned communication performs significantly worse than other methods, including the baseline. At 2 ranks per node, however, partitioned communication performance slightly overtakes the baseline and at 8 ranks per node, it overtakes persistent communication performance as well. As the ranks per node increase, the threads per rank decrease. Therefore, partitioned communication only achieves speedups when each rank has a limited number of threads, indicating that there is a limit to the number of partitions into which a message should be split. Note for the single rank per node case, the threads were likely split across sockets, potentially amplifying overheads.

VI. Conclusions and Future Directions

This paper demonstrated how persistent and partitioned MPI communication optimizations can be used to improve halo exchanges. Persistent communication optimizes iterative stencil exchanges, providing significant speedups over standard non-blocking communication. Partitioned communication provides additional speedups when fewer threads are used per process, as long as data exchanges are large.

Additional partitioned communication optimizations can be further explored in the future, namely performing early work, unpacking data immediately upon arrival through the use of the `MPI_Parrived` routine. Furthermore, the use of partitioned communication can be explored on heterogeneous architectures. Depending on future availability of stream and kernel triggered MPI routines, it is possible partitioned communication could be used to optimize GPUDirect communication. Partitioned communication could also be explored in the context of copy-to-CPU communication routines on heterogeneous architectures, as modeling results have shown use cases GPUDirect is outperformed when copying to the CPU and using all available CPU cores [4]. Partitioned communication has the potential to further improve copy-to-CPU methods by partitioning large messages across all available CPU cores.

This paper presents an initial analysis of the impact of partitioned communication on stencil exchanges, with results guiding use cases for the optimization. However, each application programmer is currently required to implement these optimizations by hand to achieve the performance benefits shown in this paper. The work presented throughout this paper could, in the future, be made accessible through the use of persistent neighborhood collectives, allowing the underlying system MPI to choose the optimal approach for a given iterative halo exchange.

Acknowledgment

This work was performed with partial support from the National Science Foundation under Grant No. CCF-2151022 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 and the U.S. Department of Energy's National Nuclear Security Administration.

References

[1] S. Abhyankar, J. Brown, E. M. Constantinescu, D. Ghosh, B. F. Smith, and H. Zhang, *Petsc/ts: A modern scalable ode/dae solver library*, 2018, https://arxiv.org/abs/1806.01437, https://arxiv.org/abs/1806.01437.

[2] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, *A view of the parallel computing landscape*, Commun. ACM, 52 (2009), p. 56–67, https://doi.org/10.1145/1562764.1562783, https://doi.org/10.1145/1562764.1562783.

[3] P. Bangalore, A. Worley, D. Schafer, R. Grant, A. Skjellum, and S. Ghafoor, *A portable implementation of partitioned point-to-point communication primitives.*, tech. report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2020.

[4] A. Bienz, L. N. Olson, W. D. Gropp, and S. Lockhart, *Modeling data movement performance on heterogeneous architectures*, 2021, https://arxiv.org/abs/2010.10378.

[5] M. G. Dosanjh, A. Worley, D. Schafer, P. Soundararajan, S. Ghafoor, A. Skjellum, P. V. Bangalore, and R. E. Grant, *Implementation and evaluation of MPI 4.0 partitioned communication libraries*, Parallel Computing, 108 (2021), p. 102827, https://doi.org/https://doi.org/10.1016/j.parco.2021.102827, https://www.sciencedirect.com/science/article/pii/S0167819121000752.

[6] B. Elis, O. Pearce, D. Boehme, J. Burmark, and M. Schulz, *Non-blocking gpu-cpu notifications to enable more gpu-cpu parallelism*, in Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, 2024, pp. 1–11.

[7] P. Georg, D. Richtmann, and T. Wettig, *pmr: A high-performance communication library*, arXiv preprint arXiv:1701.08521, (2017).

[8] T. GILLIS, K. RAFFENETTI, H. ZHOU, Y. GUO, AND R. THAKUR, *Quantifying the performance benefits of partitioned communication in mpi*, in Proceedings of the 52nd International Conference on Parallel Processing, 2023, pp. 285–294.

[9] R. E. GRANT, M. G. F. DOSANJH, M. J. LEVENHAGEN, R. BRIGHTWELL, AND A. SKJELLUM, *Finepoints: Partitioned multithreaded MPI communication*, in High Performance Computing, M. Weiland, G. Juckeland, C. Trinitis, and P. Sadayappan, eds., Cham, 2019, Springer International Publishing, pp. 330–350.

[10] J. M. HASHMI, C.-H. CHU, S. CHAKRABORTY, M. BAYATPOUR, H. SUBRAMONI, AND D. K. PANDA, *Falcon-x: Zero-copy mpi derived datatype processing on modern cpu and gpu architectures*, Journal of Parallel and Distributed Computing, 144 (2020), pp. 1–13.

[11] Y. HASSAN TEMUCIN, R. E. GRANT, AND A. AFSAHI, *Micro-benchmarking mpi partitioned point-to-point communication*, in Proceedings of the 51st International Conference on Parallel Processing, 2022, pp. 1–12.

[12] M. HATANAKA, A. HORI, AND Y. ISHIKAWA, *Optimization of mpi persistent communication*, in Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, New York, NY, USA, 2013, Association for Computing Machinery, p. 79–84, https://doi.org/10.1145/2488551.2488566, https://doi.org/10.1145/2488551.2488566.

[13] D. J. HOLMES, A. SKJELLUM, J. JAEGER, R. E. GRANT, P. V. BANGALORE, M. G. DOSANJH, A. BIENZ, AND D. SCHAFER, *Partitioned collective communication*, in 2021 Workshop on Exascale MPI (ExaMPI), IEEE, 2021, pp. 9–17.

[14] T. JAMMER AND C. BISCHOF, *Compiler-enabled optimization of persistent mpi operations*, in 2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI), IEEE, 2022, pp. 1–10.

[15] W. P. MARTS, A. WORLEY, P. SOUNDARAJAN, D. SCHAFER, M. G. F. DOSANJH, R. E. GRANT, P. V. BANGALORE, A. SKJELLUM, AND S. GHAFOOR, *Design of a portable implementation of partitioned point-to-point communication primitives*, Concurrency and Computation: Practice and Experience, 35 (2023), p. e7655, https://doi.org/https://doi.org/10.1002/cpe.7655, https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7655, https://arxiv.org/abs/https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.7655.

[16] B. MORGAN, D. J. HOLMES, A. SKJELLUM, P. BANGALORE, AND S. SRIDHARAN, *Planning for performance: persistent collective operations for mpi*, in Proceedings of the 24th European MPI Users' Group Meeting, 2017, pp. 1–11.

[17] K. NAKAJIMA, *Optimization of serial and parallel communications for parallel geometric multigrid method*, in 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), 2014, pp. 25–32, https://doi.org/10.1109/PADSW.2014.7097787.

[18] C. PEARSON, M. HIDAYETOĞLU, M. ALMASRI, O. ANJUM, I.-H. CHUNG, J. XIONG, AND W.-M. W. HWU, *Node-aware stencil communication for heterogeneous supercomputers*, in 2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2020, pp. 796–805, https://doi.org/10.1109/IPDPSW50202.2020.00136.

[19] C. PEARSON, K. WU, I.-H. CHUNG, J. XIONG, AND W.-M. HWU, *Tempi: An interposed mpi library with a canonical representation of cuda-aware datatypes*, in Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing, 2021, pp. 95–106.

[20] J. PEKKILÄ, M. S. VÄISÄLÄ, M. J. KÄPYLÄ, M. RHEINHARDT, AND O. LAPPI, *Scalable communication for high-order stencil computations using cuda-aware mpi*, Parallel Computing, 111 (2022), p. 102904.

[21] W. SCHONBEIN, S. LEVY, M. G. DOSANJH, W. P. MARTS, E. REID, AND R. E. GRANT, *Modeling and benchmarking the potential benefit of early-bird transmission in fine-grained communication*, in Proceedings of the 52nd International Conference on Parallel Processing, 2023, pp. 306–316.

[22] A. WORLEY, P. PREMA SOUNDARARAJAN, D. SCHAFER, P. BANGALORE, R. GRANT, M. DOSANJH, A. SKJELLUM, AND S. GHAFOOR, *Design of a portable implementation of partitioned point-to-point communication primitives*, in 50th International Conference on Parallel Processing Workshop, ICPP Workshops '21, New York, NY, USA, 2021, Association for Computing Machinery, https://doi.org/10.1145/3458744.3474046, https://doi.org/10.1145/3458744.3474046.

[23] R. ZAMBRE AND A. CHANDRAMOWLISHWARAN, *Lessons learned on mpi+ threads communication*, in SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2022, pp. 1–16.