Bruck Algorithm Performance Analysis for Multi-GPU All-to-All Communication

Andres Sewell
Utah State University
Logan, Utah, USA
a02024444@usu.edu

Ke Fan
University of Illinois Chicago
Chicago, Illinois, USA

Ahmedur Rahman Shovon
University of Illinois Chicago
Chicago, Illinois, USA

Landon Dyken
University of Illinois Chicago
Chicago, Illinois, USA

Sidharth Kumar
University of Illinois Chicago
Chicago, Illinois, USA

Steve Petruzza
Utah State University
Logan, Utah, USA

ABSTRACT

In high-performance computing, collective communication is critical for facilitating comprehensive data exchange involving all processes within an MPI communicator. Due to their inherently global nature, many collective operations present scalability challenges, particularly the all-to-all data shuffle with its quadratic communication pattern. Using a logarithmic communication pattern, the Bruck algorithm was designed to provide communication efficiency for all-to-all data shuffles involving short-sized messages. The Bruck algorithm has been extensively used to facilitate global data shuffles in a multi-CPU environment and is also part of the MPICH and Open MPI implementations. This work presents the first investigation of using the Bruck algorithm for all-to-all communication in multi-GPU systems using the NVIDIA Collective Communications Library (NCCL). Our experimental study demonstrates that while the Bruck algorithm exhibits superior performance for small-sized messages in a multi-CPU environment, the same advantages are not evident for multi-GPU environments. Furthermore, we describe and compare an optimized Bruck algorithm implementation in NCCL and compare it to NCCL's default all-to-all and MPI-based implementations. Finally, we discuss the challenges and opportunities of implementing new multi-GPU collectives using NCCL's public-facing API.

ACM Reference Format:
Andres Sewell, Ke Fan, Ahmedur Rahman Shovon, Landon Dyken, Sidharth Kumar, and Steve Petruzza. 2024. Bruck Algorithm Performance Analysis for Multi-GPU All-to-All Communication. In *International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2024), January 25–27, 2024, Nagoya, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3635035.3635047

1 INTRODUCTION

Collective functions facilitate data exchange involving *all* processes within an MPI communicator. Historically, collective functions have been used extensively by irregular applications [21, 29, 35] to manage their non-uniform and often sparse workloads. Collectives are generally known to be challenging to scale due to their global nature. Among all collectives, the all-to-all data shuffle is notorious for being the most difficult to scale [14, 27, 33] —primarily because of its *quadratic* communication pattern.

Global data shuffle can be classified into two categories: uniform, where processes exchange the same amount of data among each other, and non-uniform, where exchanged message sizes can vary. Both can be performed using MPI's built-in collectives `MPI_Alltoall` and `MPI_Alltoallv`. These are used by a variety of applications, including parallel training of large-scale neural networks [26], transpose computations in parallel FFT computation [9] and parallel sorting [30].

In this work, we focus on uniform all-to-all where data exchanges are generally performed using two kinds of algorithms: spread-out [13] or Bruck [6, 33]. Spread-out internally performs a *linear* (w.r.t process counts) number of communication steps. It can be visualized as a matrix-like communication pattern, where each process sends data to all other processes in a collective manner. For P processes, each process communicates with the other $P-1$ processes, resulting in a total of $P \times (P-1)$ data exchanges. Based on the *circular shift* and *bit-wise exchange* operations, the Bruck algorithm, on the other hand, performs $\log_2 P$ communication steps. Reducing communication steps (relative to spread-out) comes at the cost of sending more total data. Therefore, spread-out is used for the exchange of large-sized messages, where communication can be saturated by bandwidth, and Bruck is used for short-sized messages, where performance improvements due to reduction in communication rounds compensate for the cost of sending more data. Popular implementations of MPI, MPICH [1] and Open MPI [2, 11], both rely on a decision tree, which helps choose between the two algorithms based on scale and workload.

In the last decade, we have seen a transition towards more heterogeneous HPC environments, where CPUs are coupled with high-performance coprocessors such as GPUs. For example, modern HPC systems such as Aurora [31], Perlmutter [19], and Frontier [4] all rely heavily on GPUs to attain their peak FLOP performance. While MPI can meet the communication needs of GPU-based nodes using features like CUDA-aware MPI, specialized Remote Direct Memory Access (RDMA) communication libraries like the open source NVIDIA Collective Communication Library (NCCL) [24] have been on the rise. NCCL facilitates optimized data communication and synchronization among multiple remote NVIDIA GPUs, making it

This work is licensed under a Creative Commons Attribution International 4.0 License.

HPCAsia 2024, January 25–27, 2024, Nagoya, Japan
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0889-3/24/01.
https://doi.org/10.1145/3635035.3635047

an attractive choice for researchers, data scientists, and engineers seeking to accelerate their applications by leveraging the immense parallel processing capabilities of GPUs. Libraries like NCCL enable high-performance inter-GPU communication by reducing the overhead incurred by unnecessary CPU/GPU data transfers. This is especially true as message sizes increase, which is the use case where GPU-to-GPU RDMA communication performs best. Being open source, AMD and Microsoft have each implemented their own NCCL-based multi-GPU communication libraries called RCCL [3] and MSCCL [18], respectively.

While the Bruck algorithm is known to yield better performance for small-sized messages in a multi-CPU environment [10, 20, 33], no study has been performed to understand its impact in a multi-GPU environment. In this paper, we investigate using the Bruck algorithm for multi-GPU all-to-all communication to understand if the algorithm benefits RDMA multi-GPU collective communication. We report an experimental study that compares different MPI and NCCL implementations of all-to-all communication primitives. Our analysis ultimately finds that the Bruck algorithm implemented using the NCCL API does not offer the same performance improvements in multi-GPU settings shown in multi-CPU settings with MPI. Finally, we delve into details to explain why the Bruck algorithm is not suited for multi-GPU environments. The contributions of this work are the following:

(1) Development of an open-source implementation of the Bruck algorithm using the NCCL framework with reproducible performance tests using the *nccl-tests* benchmark suite [1].
(2) Performed a comparative study of the two main algorithms used in multi-CPU collectives: spread-out and Bruck for small message sizes.
(3) Described a NCCL-based Bruck implementation and performed scaling studies to compare this against the default NCCL implementation and MPI multi-CPU implementations.
(4) Discussed the challenges and benefits of using the public-facing NCCL APIs to develop optimized communication algorithms.

This study holds significant importance for the HPC community as it sheds light on the efficacy of the Bruck algorithm in the context of a multi-GPU environment, a domain where its performance had not been comprehensively evaluated before. The negative result, indicating that the Bruck algorithm does not offer the anticipated performance improvements for short-sized messages in multi-GPU scenarios, is valuable for the community and provides a deeper understanding of the complexities associated with collective communication in multi-GPU scenarios, guiding future research toward more efficient solutions.

2 BACKGROUND

In this section, we summarize important applications that require and use all-to-all communication and describe the basic and optimized versions of the Bruck algorithm. While the algorithm has been adopted by state-of-the-art MPI implementations for multi-CPU communication, there is no previous study that assesses its performance in multi-GPU settings.

[1] https://github.com/ComputingElevatedLab/nccl-bruck

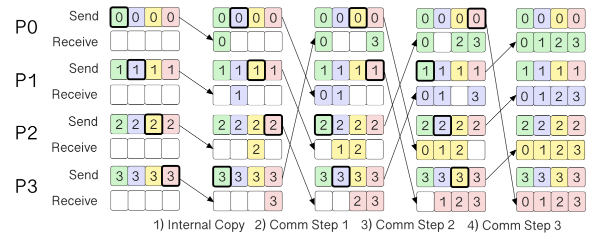

Figure 1: A demonstration of the spread-out algorithm. This algorithm performs a linear number of communication steps wherein processes send one data block directly to a target destination process per communication round, following a round-robin sequence. The send-block for each communication round is indicated by a thick cell border.

2.1 All-to-all Communication

In parallel computing, there exist several fundamental collective communication patterns. An all-to-all operation refers to every process sending data to every other process and receiving data from every other process. There are many use cases for all-to-all communication, with some simple examples including parallel FFT [32], computing matrix transposes, and accelerating parallel relational algebra at scale [15, 16].

In uniform all-to-all, the amounts of data being sent and received by each process are fixed, whereas in non-uniform all-to-all, the amounts of data sent and received may be variable. Some possible approaches for achieving this pattern are the point-to-point, spread-out, and Bruck algorithms. In point-to-point communication, each process sends and receives an entire message directly to every other process in $P-1$ communication steps, where P is the number of processes. Destination processes are chosen in a round-robin fashion to avoid a bottleneck from multiple processes attempting to send data to the same destination at once. While simple to implement, this approach can lead to network contention on the receiver side as the number of processes increases. The spread-out algorithm is the standard linear-time implementation of all-to-all data shuffle adopted by production MPI libraries and is used for both uniform and non-uniform data exchanges. Unlike point-to-point, processes only send one data block directly to a target destination process per communication round. This algorithm also takes $P-1$ communication steps. A diagram of the spread-out algorithm can be seen in Figure 1.

Libraries like Unified Collective Communication (UCC) [8] support distributed heterogeneous communication by selecting the best implementation available (e.g., using NCCL or MPI) for a specific use case based on various runtime heuristics. Our experimental study will shed some light on how some of those heuristics related to all-to-all collectives could be defined based on message size and scale.

2.2 Bruck Algorithm

The Bruck algorithm for all-to-all communication within message-passing systems was first published in 1997 [7]. Bruck differentiates itself from alternative all-to-all communication algorithms by minimizing the total number of internal communication steps involved

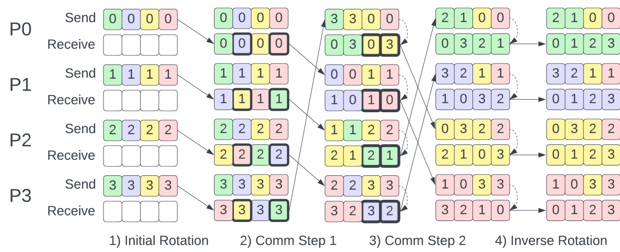

Figure 2: A demonstration of the basic Bruck algorithm. P0-3 are each processes with their own send and receive buffers. The arrows represent information being passed (via send and receive operations), and thick cell borders indicate the send-blocks for that communication round.

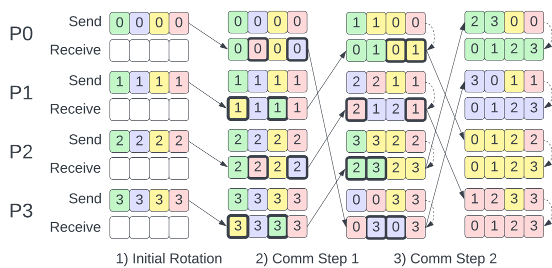

Figure 3: A demonstration of how the modified Bruck algorithm omits the inverse rotation step but achieves the same final result. Avoiding the final rotation is possible due to slight tweaks in how data is copied during previous steps.

in the all-to-all transaction. It reduces them from *O(P)* to *O(log P)* communication steps, where *P* represents the number of processes or compute units. This is possible by transmitting a larger aggregate data size while distributing it over a reduced number of iterations. This strategy offers significant advantages when dealing with data messages of relatively small sizes (*16b* to *2K*) [5]. By leveraging the increased bandwidth available by handling smaller messages, the algorithm utilizes available computing resources more efficiently. This enables the Bruck algorithm to process small data messages efficiently, improving overall performance and reducing execution time in communication-bound scenarios. Figure 2 demonstrates how Bruck performs *log(4)=2* communication steps for 4 processes, as opposed to the 3 spread-out communication steps shown in Figure 1.

As a testament to its effectiveness, the Bruck algorithm has been widely adopted in state-of-the-art MPI implementations, including MPICH [33] and Open MPI [12], specifically to implement the uniform all-to-all collective communication operation (*MPI_Alltoall*). The basic Bruck algorithm (see Figure 2) has three steps, including an initial local rotation, *log(P)* global communications, and a final local inverse rotation [7]. The modified inverse Bruck algorithm enhances the basic Bruck algorithm by removing the final local inverse rotation step [34]. This removal is possible through subtle adjustments in data copying within earlier phases, which preclude the need for the final local inverse rotation. Figure 3 illustrates the difference between the modified inverse Bruck and the basic variant. The zero-copy variant further improves the algorithm by eliminating the need for explicit data copying in uniform all-to-all communication, enabling in-place data access during communication operations. Specifically, when performing a uniform all-to-all operation in *MPI_Alltoall*, the zero-copy Bruck algorithm can achieve a significant performance improvement [34].

3 ALL-TO-ALL COLLECTIVE IMPLEMENTATIONS

Recent work based on the modified Bruck [10] has been shown to outperform the linear-step spread-out implementation in a multi-CPU setting. For this reason, we hypothesized that the Bruck algorithm could also be promising for achieving faster and more efficient all-to-all communication in multi-GPU scenarios. We were particularly interested to see how it performs at scale, given that modern HPC clusters are now being built with thousands of total GPUs. We begin this section by presenting the existing implementation of all-to-all data exchange within NCCL and then describe our implementation of the Bruck algorithm within NCCL.

3.1 Default NCCL All-to-all Implementation

As of the NCCL 2.18.1 documentation [23], there does not exist an explicitly named implementation of all-to-all data shuffle within NCCL. Rather, all-to-all communication is achieved by defining a for-loop of NCCL send and receive operations wrapped within a *ncclGroup*. This is conceptually equivalent to the spread-out algorithm, as it takes a linear number of communication rounds. For the sake of clarity, even though NCCL does not provide a named implementation for all-to-all data shuffle, we will refer to this as the default NCCL all-to-all for the remainder of this paper, expressing it in Algorithm 1. As can be seen in the algorithm, there is a linear-step loop (w.r.t the number of processes) in line number 5, where for each iteration, a process sends and receives data from some other process. A key point to note is the usage of `ncclGroupStart` and `ncclGroupEnd`, used to wrap the loop of communication rounds.

Algorithm 1 Default NCCL all-to-all implementation

1: $P \leftarrow$ total number of processes.
2: $sendbuf \leftarrow$ buffer for data to be sent.
3: $recvbuf \leftarrow$ buffer to store received data.
4: **ncclGroupStart()**
5: **for** $i \in [0, P]$ **do**
6:     send data in $sendbuf[i]$ to i;
7:     receive data from i into $recvbuf[i]$;
8: **end for**
9: **ncclGroupEnd()**

The implementation of this collective in NCCL is interesting due to every send and receive operation being wrapped into one *ncclGroup*, a concept that MPI has no direct equivalent for. *ncclGroups* are defined by their start and end functions, which queue any intermediate NCCL operations to be executed after the group ends. This approach enables the NCCL runtime to capture the full communication scenario and apply optimizations. The NCCL documentation

states that groups are used for 'managing multiple GPUs from one thread (to avoid deadlocks), aggregating communication operations to improve performance, or merging multiple send/receive point-to-point operations' [22].

A *ncclGroup* execution is treated as a single communication, avoiding the GPU kernel launch overhead that would be associated with executing each communication operation individually. Despite the default NCCL all-to-all appearing to use the spread-out algorithm, internal runtime optimizations may potentially merge the calls to improve performance.

3.2 NCCL Bruck Implementation

In Algorithm 2, we present the Bruck algorithm as implemented using NCCL. There are two key points to note in this algorithm:

(1) The total number of iterations performed here is $\log P$.
(2) The `ncclGroupStart` and `ncclGroupEnd` wrap each send and receive operation individually (see line number 17 and 19), as opposed to encompassing the entire for loop in Algorithm 1.

The usage of *ncclGroups* can be explained by further examining the Bruck algorithm. As opposed to the spread-out (or point-to-point) implementation of Algorithm 1, Bruck is a store-and-forward algorithm that takes $\log(P)$ communication steps. This means that both send (S) and receive (R) data buffers are used for sending, receiving, and storing data during intermediate communication rounds. Unlike spread-out, buffers S and R are both involved in the communication step, as some received data blocks will have to be present for a later communication step. This store-and-forward nature of the algorithm imposes constraints on the ordering of the communication rounds. Unlike the linear-step implementations, Bruck must maintain an explicit communication ordering, where iteration $i+1$ *must* occur after iteration i in physical time. The algorithm can also be seen in Figure 3, which shows that the different communication phases must be executed in a sequential order.

As discussed earlier, once a *ncclGroup* enqueues a set of send and receive operations, the NCCL runtime will be responsible for scheduling those operations, and strict ordering cannot be enforced. Therefore, wrapping all of the send and receive operations produced by the Bruck algorithm into a single *ncclGroup* will lead to incorrect results. For this reason, we could only create a *ncclGroup* for each pair of *send* and *recv* operations (see Algorithm 2, line number 17 and 20). In the evaluation section, we discuss how this requirement affects the performance of the Bruck algorithm for multi-GPU all-to-all collective communication using NCCL.

Algorithm 2 NCCL Bruck algorithm

1: $P \longleftarrow$ total number of processes.
2: **for** $i \in [0, P]$ **do**
3: $R[i] = S[(p+i)\%P]$ // S and R are send and receive buffers, and p is rank id of each process;
4: **end for**
5: allocate temporary buffer T with $SC \times (P+1)/2$ elements; // SC is number of elements per data-block.
6: **for** $k = 1;\ k < P;\ k <<= 1$ **do**
7: allocate send indexes array SB with $(P+1)/2$ integers;
8: number of send data-blocks $NB \longleftarrow 0$;
9: **for** $i \in [k, P]$ **do**
10: **if** $i\ \&\ k$ **then**
11: $NB[NB] \longleftarrow i$;
12: copy $R[i]$ into $T[NB]$;
13: $NB \longleftarrow NB + 1$;
14: **end if**
15: $sendproc \longleftarrow (p+k)\ \%\ P$;
16: $recvproc \longleftarrow (p-k+P)\ \%\ P$;
17: **ncclGroupStart()**
18: send data in T to *sendproc*;
19: receive data from *recvproc* into S;
20: **ncclGroupEnd()**
21: **for** $i \in [0, SB]$ **do**
22: copy $T[i]$ into $R[SB[i]]$;
23: **end for**
24: **end for**
25: **for** $i \in [0, P]$ **do**
26: $R[i] = R[(p-i+P)\ \%\ P]$ // final rotation;
27: **end for**
28: **end for**

4 EVALUATION

In this section, we report experimental studies to assess the performance of uniform all-to-all collectives for small-sized messages using the Bruck algorithm in both multi-CPU and multi-GPU settings. Furthermore, we compare implementations of the Bruck algorithm using both NCCL (multi-GPU) and MPI (multi-CPU) to understand if and when this algorithm would be effective for multi-GPU collectives. We performed our experimentation on the Polaris supercomputer [17] operated by the Argonne Leadership Computing Facility at Argonne National Laboratory. Polaris consists of 560 nodes, each containing a single 2.8 GHz AMD EPYC Milan 7543P 32-core CPU, 512 GB of DDR4 RAM, and 4 NVIDIA A100 40GB GPUs connected via NVLink. Our software stack included CUDA 11.8, which was provided via NVHPC 23.1 and with NCCL 2.16.4.

4.1 MPI-based Multi-CPU All-to-all

We performed a weak scaling study of a uniform all-to-all communication pattern in a multi-CPU setting, comparing two algorithms: spread-out and Bruck. For this study, the total message size every MPI process sends varies proportionally with the number of processes involved in the collective operation. For example, looking at the first plot in Figure 4, the total amount of data sent by every process varied from 64×16 bytes at 64 processes to 512×16 bytes at 512 processes. The results reported in Figure 4 show that the Bruck algorithm performs best for small message sizes, especially at larger scales. This is due to its more efficient communication pattern, which groups smaller messages into fewer large data exchanges. However, the advantages diminish as messages become larger. This experimental study serves as the motivation for our experimentation in multi-GPU settings, where we wanted to validate if the Bruck algorithm would continue to perform well for small-sized messages.

Figure 4: Weak scaling study comparing MPI all-to-all methods, our basis for investigating Bruck performance in NCCL. The Bruck implementation performs significantly better for small-sized messages and at larger scales than spread-out. This advantage, however, becomes much smaller for large-sized messages.

4.2 NCCL-based Multi-GPU All-to-all

For our multi-GPU implementation of the Bruck algorithm, we used NVIDIA's NCCL library. NVIDIA provides an open source tool for benchmarking NCCL collectives called *nccl-tests* [25]. It provides many useful features, such as a configurable number of warm-up and benchmark iterations, varying message sizes, result verification, and so on. For these reasons, we used *nccl-tests* to perform our experiments. The *nccl-tests* benchmark suite first prepares the GPU buffers and then passes them to a test function. The test function is a generic interface that links to a range of different implementations. This ensures that all of the tests receive the same data to start with. The process of adding new algorithms to the benchmark consists of creating a separate test function for each new algorithm. Since everything is implemented as a test function, the timer can start and stop in the same place across all algorithms. All existing tests conclude once the GPU buffers contain the final result. We performed the following four sets of experiments:

(1) `Default NCCL All-to-All` - this is the default implementation of all-to-all currently provided by NCCL. This linear-step implementation is directly based on Algorithm 1.
(2) `NCCL Modified Bruck` - this is our implementation of modified Bruck using the public-facing NCCL APIs. It performs a logarithmic number of communication rounds and is directly based on Algorithm 2.
(3) `MPI Spread-out` - an implementation of spread-out that relies upon the multi-CPU data exchange protocol.
(4) `MPI Modified Bruck` - an implementation of modified Bruck that relies upon the multi-CPU data exchange protocol.

We note that for the latter two implementations, we performed a data offload (i.e., *memcpy*) between GPU and CPU before performing the MPI collective. This approach is useful to understand the trade-offs of using direct multi-GPU collective operations vs. offloading the data to CPUs to execute multi-CPU (MPI) collectives instead.

The experimental study reported in Figure 5 compared the MPI all-to-all implementations and the default NCCL all-to-all performance to our modified Bruck implementation with message sizes ranging from $16b$ to $2KB$. The experiments were run on 16, 32, 64, and 128 nodes resulting in data collected with 64, 128, 256, and 512 GPUs. All experiments were performed using one MPI process per GPU, measuring average execution times across MPI ranks, using send and receive buffers of type *ncclChar*, non-blocking communication, and setting the NCCL_PLUGIN_P2P environment variable to UCX [28] (performance without the UCX plugin reported the same overall trends). As a note, although it is possible to use one MPI process per node to manage four GPUs each, a best practice is to let each MPI process be responsible for managing exactly one GPU. Default configuration values were left unchanged unless a different value resulted in a performance improvement, as in the case of setting a non-default NCCL_PLUGIN_P2P value. For all experiments, the number of warm-up iterations was set at 100, and the number of benchmark iterations was set at 500. Increasing the number of samples measured helped reduce the influence of outliers.

In Figure 5, we report the experimental results of our weak scaling study comparing both multi-CPU and multi-GPU performance for uniform all-to-all collectives. For multi-CPU settings, we also include the time to load and unload data between CPU and GPU. This is done to understand when it would be convenient to rely on direct multi-GPU collective vs. multi-CPU MPI collectives. We make two key observations: Direct GPU-to-GPU communication is slower at scale when compared to offloading the same data to the CPU and performing the same MPI all-to-all collective across CPUs. This trend is validated by the red trendline in all figures, which corresponds to multi-CPU-based modified Bruck– it consistently outperforms all other approaches at a larger scale. (ii) Surprisingly, the default NCCL-based all-to-all method demonstrates better performance than our Bruck implementation in NCCL. As highlighted in the next section, such performance loss can be attributed to the overhead associated with GPU kernel launches that take place during the execution of each separate `ncclGroup`.

5 DISCUSSION

As stated previously, the Bruck algorithm consists of various phases: an initial data rotation, a communication phase, and a final data rotation. Each of these phases is dependent on the result of the previous phase, and this is true for each of the communication steps as well. Each Bruck communication step is a *sendrecv* operation

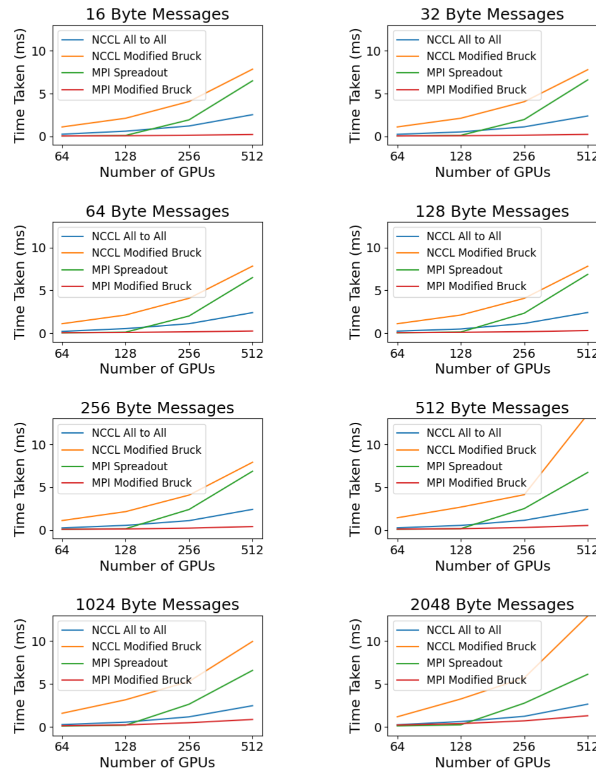

Figure 5: Weak scaling study comparing multi-GPU and multi-CPU all-to-all methods. In multi-GPU settings, the default NCCL all-to-all implementation always outperforms the Bruck implementation. Furthermore, at a larger scale, offloading data to the CPU and using an MPI multi-CPU implementation yields better performance for those message sizes.

that requires some amount of data to be copied from the receive buffer into a temporary buffer beforehand. This makes Bruck an inherently serial algorithm, and disqualifies our NCCL implementation from using a single *ncclGroup* to aggregate and optimize all of the communication operations at once. In this scenario, we lose the opportunity for the NCCL runtime to perform aggregate communication optimizations, and we also incur the overhead of repeatedly creating and executing separate *ncclGroups*, all while the operation progresses synchronously. In contrast, the default NCCL all-to-all is able to execute its entire communication scenario asynchronously with exactly one *ncclGroup*, incurring only a constant amount of kernel launch overhead.

Furthermore, for very small message sizes, our experimental results suggest that it is faster to copy device (GPU) memory into host (CPU) memory before using MPI to perform data exchanges. Message sizes are an example of a heuristic that can help communication libraries determine at runtime which API and collective implementation will perform best for the given scenario.

We contacted an NVIDIA employee who works on distributed multi-GPU applications to discuss our findings. They explained that NCCL may optimize collectives internally and that the process is not transparent to end-users. The conversation reiterated the importance of using *ncclGroups* as well as the fact that for such small message sizes, using MPI tends to be faster than using NCCL directly. The reason for this is that GPU kernel launches take a relatively long time compared to data transfer and MPI communication. Our primary takeaways from the conversation were that our results appear reasonable and that a new all-to-all implementation would generally have to be implemented within NCCL itself to be competitive with the public-facing API.

6 CONCLUSION

This work presents the first experimental study to assess the performance of the Bruck algorithm for uniform all-to-all communication in multi-GPU settings using NVIDIA's NCCL library. We described how the implementation of the Bruck algorithm in NCCL leverages *ncclGroups*, which is a mechanism that allows for multiple communication primitives (i.e., send and receives) to be aggregated, optimized, and executed asynchronously by the NCCL runtime. We presented an experimental study that also includes multi-CPU (MPI) all-to-all collective operations to understand when it is ideal to rely on multi-GPU RDMA collective communication vs. offloading data to the CPU and performing MPI collectives. Our experiments conclude that the Bruck algorithm for all-to-all communication does not outperform the default NCCL all-to-all implementation. We have demonstrated that this is clearly in contrast to Bruck's multi-CPU performance, which outperforms its point-to-point and spread-out alternatives for the same message sizes.

This discrepancy is explained by the fact that the Bruck algorithm requires multiple phases of data exchanges that need to be executed in a strict sequential order. When implementing strict communication ordering using NCCL, it was required that we use a separate *ncclGroup* for each communication phase, eventually introducing significant overhead due to each *ncclGroup* launching a separate GPU kernel.

The insights from this study are important for understanding how collective operations perform in multi-GPU settings and will help the community set proper heuristics in future implementations to determine the best API and algorithm to use for a given communication workload and scale.

ACKNOWLEDGMENTS

This work was partly funded by NSF RII Track-4 Award 2132013, NSF Collaborative Research Awards 2221811 and 2221812, and NSF PPoSS Planning and Large awards 2217036 and 2316157. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Polaris supercomputer located at the Argonne National Laboratory.

REFERENCES

[1] MPICH Home Page. https://www.mpich.org.
[2] OpenMPI Home Page. https://www.open-mpi.org.
[3] AMD. 2023. rccl. https://github.com/ROCmSoftwarePlatform/rccl.
[4] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, Thomas Evans, Matthew Ezell, Nicholas Frontiere, Antigoni Georgiadou, Joe Glenski, Philipp Grete, Steven Hamilton, John Holmen, Axel Huebl, Daniel Jacobson, Wayne Joubert, Kim Mcmahon, Elia Merzari, Stan Moore, Andrew Myers, Stephen Nichols, Sarp Oral, Thomas Papatheodore, Danny Perez, David M. Rogers, Evan Schneider, Jean-Luc Vay, and P. K. Yeung. 2023. Frontier: Exploring Exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, CO, USA) *(SC '23)*. Association for Computing Machinery, New York, NY, USA, Article 52, 16 pages. https://doi.org/10.1145/3581784.3607089
[5] Amanda Bienz, Shreeman Gautam, and Amun Kharel. 2022. A Locality-Aware Bruck Allgather. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA'22)*. Association for Computing Machinery, New York, NY, USA, 18–26. https://doi.org/10.1145/3555819.3555825
[6] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal, and Derrick Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on parallel and distributed systems* 8, 11 (1997), 1143–1156.
[7] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. 1997. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems* 8, 11 (1997), 1143–1156. https://doi.org/10.1109/71.642949
[8] UCF Consortium. 2023. Unified collective communication (ucc). https://ucfconsortium.org/projects/ucc/.
[9] Jun Doi and Yasushi Negishi. 2010. Overlapping methods of all-to-all communication and FFT algorithms for torus-connected massively parallel supercomputers. In *SC'10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–9.
[10] Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 2022. Optimizing the bruck algorithm for non-uniform all-to-all communication. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 172–184.
[11] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.
[12] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*. Budapest, Hungary, 97–104.
[13] Qiao Kang, Robert Ross, Robert Latham, Sunwoo Lee, Ankit Agrawal, Alok Choudhary, and Wei-keng Liao. 2020. Improving all-to-many personalized communication in two-phase i/o. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.
[14] R. Kumar, A. Mamidala, and D. K. Panda. 2008. Scaling alltoall collective on multi-core systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8.
[15] Sidharth Kumar and Thomas Gilray. 2019. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, Vol. 1.
[16] Sidharth Kumar and Thomas Gilray. 2020. Load-balancing parallel relational algebra. In *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35*. Springer, 288–308.
[17] Argonne National Laboratory. 2023. Polaris. https://www.alcf.anl.gov/polaris
[18] Microsoft. 2023. msccl. https://github.com/microsoft/msccl.
[19] NERSC. 2022. Perlmutter. https://www.nersc.gov/systems/perlmutter/
[20] Nick Netterville, Ke Fan, Sidharth Kumar, and Thomas Gilray. 2022. A Visual Guide to MPI All-to-all. In *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*. 20–27. https://doi.org/10.1109/HiPCW57629.2022.00008
[21] Jaechun No, Rajeev Thakur, Dinesh Kaushik, Lori Freitag, and Alok Choudhary. 2001. A scientific data management system for irregular applications. *arXiv preprint cs/0102016* (2001).
[22] NVIDIA. 2019. Group Calls. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/groups.html
[23] NVIDIA. 2019. Point-to-point communication. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/p2p.html#all-to-all
[24] NVIDIA. 2023. nccl. https://github.com/NVIDIA/nccl.
[25] NVIDIA. 2023. nccl-tests. https://github.com/NVIDIA/nccl-tests.
[26] Cèdric Renggli, Saleh Ashkboos, Mehdi Aghagolzadeh, Dan Alistarh, and Torsten Hoefler. 2019. SparCML: High-performance sparse communication for machine learning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.
[27] Rodolphe Sepulchre, Derek A Paley, and Naomi Ehrich Leonard. 2007. Stabilization of planar collective motion: All-to-all communication. *IEEE Transactions on automatic control* 52, 5 (2007), 811–824.
[28] Pavel Shamis, Manjunath Gorentla Venkata, M Graham Lopez, Matthew B Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L Graham, Liran Liss, et al. 2015. UCX: an open source framework for HPC network APIs and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 40–43.
[29] Lorna Smith and Mark Bull. 2001. Development of mixed mode MPI/OpenMP applications. *Scientific Programming* 9, 2-3 (2001), 83–98.
[30] Edgar Solomonik and Laxmikant V. Kalé. 2010. Highly scalable parallel sorting. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)* (2010), 1–12.
[31] Rick Stevens, Jini Ramprakash, Paul Messina, Michael Papka, and Katherine Riley. 2019. *Aurora: Argonne's next-generation exascale supercomputer*. Technical Report. Argonne National Lab.(ANL), Argonne, IL (United States).
[32] D. Takahashi. 2000. High-performance parallel FFT algorithms for the HITACHI SR8000. In *Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, Vol. 1. 192–199 vol.1. https://doi.org/10.1109/HPC.2000.846545
[33] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. 2005. Optimization of collective communication operations in MPICH. *The International Journal of High Performance Computing Applications* 19, 1 (2005), 49–66.
[34] Jesper Larsson Träff, Antoine Rougier, and Sascha Hunold. 2014. Implementing a classic: Zero-copy all-to-all communication with MPI datatypes. In *Proceedings of the 28th ACM international conference on Supercomputing*. 135–144.
[35] Katherine A Yelick. 1993. Programming models for irregular applications. *ACM SIGPLAN Notices* 28, 1 (1993), 28–31.