MPI_Alltoallv Optimizations on GPU-Enabled Architectures

Evelyn Namugwanya[1], Amanda Bienz[2], Derek Schafer[3], and Anthony Skjellum[4]

[1] Tennesse Tech University, Cookevile, USA
[2] University of New Mexico, Albuquerque, USA
[3] University of New Mexico, Albuquerque, USA
[4] Tennesse Tech University, Cookevile, USA
enamugwan@tntech.edu, bienz@unm.edu, dschafer1@unm.edu, askjellum@tntech.edu

Abstract. This paper presents optimized strategies for the `MPI_Alltoallv` collective communication operation, crucial for high-performance applications such as FFT solvers. Due to the dominance of GPU-enabled systems, we evaluated both CUDA-aware and Copy-to-CPU methods, using the MPI Advance framework to enhance scalability and efficiency. CUDA-aware methods allow MPI to optimize the path of communication, such as through direct GPU-to-GPU transfers with GPUDirect communication, whereas Copy-to-CPU algorithms incorporate persistent memory buffers to reduce allocation overhead, excelling with larger data sizes. The study's structured analysis compares these algorithms on the Lassen and Tioga supercomputers under varying message sizes and process counts. Results reveal distinct performance benefits: on Lassen, manually copying to the CPU and initializing all messages with non-blocking sends and receives achieves up to a 3× speedup over CUDA-aware Spectrum MPI with large message sizes, outperforming CUDA-aware approaches in high process-count scenarios. However, on Tioga, performing a batched nonblocking exchange directly between GPUs demonstrates superior scalability up to 5× faster at large message sizes and higher process counts, maintaining lower execution times than Cray-MPICH Alltoallv. The paper also integrates these optimizations into real world applications such as HeFFTe, a state-of-the-art FFT solver, and tests with Beatnik, a mini-app that simulates 3D Raleigh-Taylor instabilities, achieving a 2-3× speedup in communication time due to optimized Alltoallv performance on Lassen. Overall, the study suggests a poly algorithm approach dynamically selecting between CUDA-aware and Copy-to-CPU methods based on data size and system configuration to maximize efficiency across diverse HPC environments.

Keywords: Collective communication, GPU-awareness, Scalability, CUDA-aware, GPUDirect, MPI_Alltoallv, High-Performance Computing (HPC), FFT solvers, GPU-enabled systems, Copy-to-CPU methods, Lassen supercomputer, Tioga supercomputer, Non-blocking communication, Spectrum MPI, Cray-MPICH, Persistent memory buffers, Batched

nonblocking exchange, HeFFTe, Beatnik, 3D Raleigh-Taylor instabilities, Communication path optimization, Data size and process count, Execution time, Algorithmic optimization, Dynamic selection, System configuration, HPC environments

1 Introduction

High-performance computing (HPC) applications, especially those requiring multi-dimensional Fast Fourier Transforms (FFTs), demand efficient collective communication operations. Among these, the MPI_Alltoallv operation is crucial for with non-uniform data exchanges, such as in FFTs like HeFFTe, a state-of-the-art FFT solver, where it is often a performance bottleneck. Optimizing `MPI_Alltoallv` for GPU-enabled architectures is particularly challenging, given the high variability in data sizes and the architectural complexities of heterogeneous systems.

In current CUDA-aware `MPI_Alltoallv` operations, data often moves from GPU memory to the CPU's memory, across the network, and then back to GPU memory on a different node. Some methods, however, leverage GPUDirect, in which data is sent directly between GPUs and the network, bypassing the CPU. This allows faster communication by avoiding extra memory transfers. However, inter-GPU communication can incur significant startup overhead on some machines, reducing the effectiveness of GPUDirect communication, particularly for large message counts.

In this paper, we address this challenge through novel `MPI_Alltoallv` implementations. All algorithms are implemented within the open-source library MPI Advance [5], and collectively exchange data by calling lower-level communication protocols, such as `MPI_Isend` and `MPI_Irecv`, which are provided by the system MPI implementation. This paper evaluates the performance of two key methods for inter-GPU collectives: 1) CUDA-Aware algorithms, which initiate all underlying communication with GPU data, allowing for the data to be sent via GPUDirect messages if the underlying MPI implementation chooses that path; and 2) Copy-to-CPU algorithms, which manually copies all data to CPU memory before exchanging all inter-process communication with lower-latency inter-CPU messages.

The novel contributions of this paper include:

- A systematic analysis of the performance trade-offs between GPUDirect and Copy-to-CPU approaches, in combination with a variety of underlying all-to-allv algorithms.
- Performance measurements spanning multiple heterogeneous supercomputers with various system MPI implementations.
- A case study of the presented all-to-allv algorithms within the FFT solver HeFFTe, showing improvements over using the system MPI implementation.
- Develop portable optimizations for all-to-allv exchanges, which can be used with any system MPI implementation, to reduce performance variability across systems.

Through structured testing across varying message sizes and process counts, we reveal insights into the performance characteristics of each approach. Our findings show that while Copy-to-CPU algorithms demonstrate superior performance for large messages and high process counts on Lassen, CUDA-aware methods, exhibit excellent scalability on Tioga.

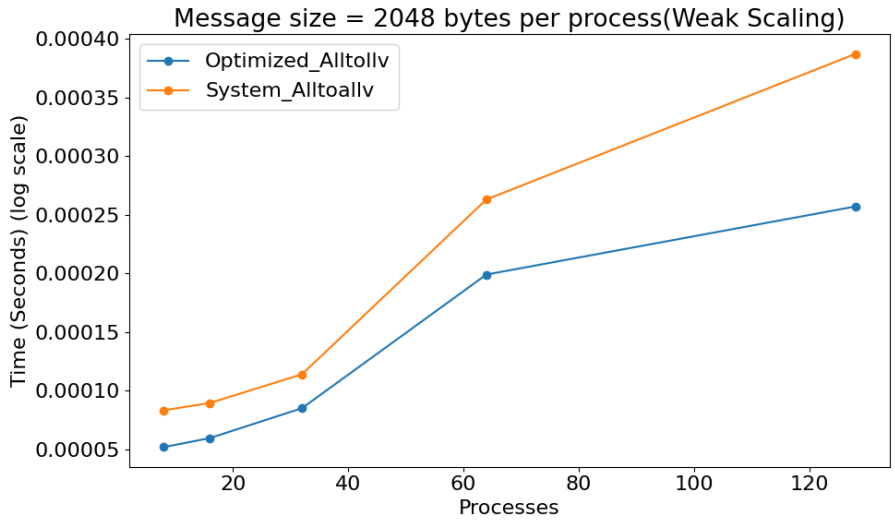

Fig. 1. 8 through 128 processes, one GPU per process. Cost of scaling `System_Alltoallv` vs. our proposed optimized Alltoallv approach on lassen.

Fig. 1 compares the performance of the CUDA-Aware `MPI_Alltoallv` provided by the system MPI (**System Alltoallv**) to the best performing exchange as implemented within MPI Advance (**Optimized Alltoallv**). At 128 processes, the **Optimized Alltoallv** is nearly 1.5 times faster than the implemention provided by the system MPI. These optimizations have been successfully integrated into real-world applications, including HeFFTe, a state-of-the-art FFT solver, and tested with Beatnik, a mini-app that simulates 3D Raleigh-Taylor instabilities, yielding a **2-3X** reduction in communication time on Lassen.

The remainder of this paper is organized as follows: In Section 2, we provide background, which offers an overview of some existing knowledge about optimizing MPI collective communication algorithms, along with previous research and related work that establish the context and set the foundation for the study. Section 3 details the methodology used here. Our results for Structured analysis are presented in Section 4, and our results for real world applications are presented in Section 5, while Section 6 concludes our study.

2 Background

Multidimensional FFT solvers are specifically bottlenecked by a global transpose during which all processes are active in a `MPI_Alltoallv` data exchange in typical implementations. Exploring Alltoallv algorithms in existing MPI's is an important background to this work (Algorithm 1).

Algorithm 1 Pairwise Exchange

Input: p {process id}
n {number of processes}
`args` {arguments passed to MPI_Alltoallv}

for $i \leftarrow 0$ **to** n **do**
  $p_{\texttt{send}} = p + i \mod n$ $p_{\texttt{recv}} = p + n - i \mod n$ Send message to $p_{\texttt{send}}$ and receive message from $p_{\texttt{recv}}$

`MPI_Alltoallv` has two common implementations: pairwise exchange and non-blocking communication. The pairwise-exchange algorithm sends to a single process and receives from one process at each step of the algorithm. For example, at step i, the process p sends to process $p - i$ and receives from process $p + i$. Assuming that all processes are ready to communicate, this algorithm minimizes overheads, such as network contention and queue search costs, since only a single message is communicated from each process in a given round. However, parallel applications, such as FFTs, often have load imbalances and unsynchronized processes, with some processes working on the `MPI_Alltoallv` while others are still computing a previous step. For instance, assume process p is stepping through the `MPI_Alltoallv` algorithm, while process $p - i$ remains in a previous step of the FFT. Process p initializes a send to process $p - i$ and waits idly, even though process $p - i - 1$ may be ready for communication. This synchronization overhead greatly reduces the performance of the pairwise exchange Alltoallv.

Algorithm 2 Non-blocking

Input: p {process id}
n {number of processes}
`args` {arguments passed to MPI_Alltoallv}

for $i \leftarrow 0$ **to** n **do**
  $p_{\texttt{send}} = p + i \mod n$ $p_{\texttt{recv}} = p + n - i \mod n$ Initialize non-blocking send to $p_{\texttt{send}}$ Initialize non-blocking receive from $p_{\texttt{recv}}$
Wait for all sends and receives to complete

The non-blocking implementation of the `MPI_Alltoallv`, consists of initiating all sends and receives with non-blocking communication, such as `MPI_Isend` and `MPI_Irecv`. Then, each process waits for all communication to complete. This

implementation avoids certain synchronization overheads, since all communication is initiated. Therefore, if process $p - i$ remains in a previous step of the FFT, process p is able to exchange data with every other process while $p - i$ completes previous computations. However, the non-blocking algorithm incurs large overheads associated with the vast amount of communication. All messages are potentially sent through the network simultaneously, increasing the likelihood of network contention in which a packet of data needs to traverse a link that is already in use by a separate packet. Depending on the routing algorithm, the packet may sit idly until the link is free of contention, adding large delays to message routing. In addition, large message counts incur significant queue search overhead. Each process posts `MPI_Irecv` calls for every message it expects to receive. Once a message arrives, all posted receives are searched to find a match. As the number of messages grows, this queue search cost incurs large performance overheads.

Related Work. One common theme in collective optimizations is minimizing the message sizes and counts communicated between sets of nodes, because intra-node communication often greatly outperforms inter-node. Hierarchical collectives achieve this minimum by gathering all data to one, or a small number of, leader processes per node before performing the collective between only the leader processes. The final results are broadcast locally within each node [12, 13, 20]. Multi-lane collectives further optimize large collective operations, with each process per node communicating an equal but minimal amount of data [21]. Locality-aware collectives minimize the number of inter-node messages, with each process per node communicating with a separate subset of nodes [2, 3]. Topology-aware optimizations also rely on architecture-awareness, minimizing communication for a given interconnect [14, 15, 17].

While the Bruck algorithm was created for `MPI_Alltoall` algorithms with small data sizes; it has recently been extended for variable data sizes within the `MPI_Alltoallv` [9]. However, the authors are unaware of general architecture-agnostic optimizations for large `MPI_Alltoallv` algorithms.

CUDA-Aware MPI facilitates the communication of data between GPU memories using the MPI API without requiring the user to invoke GPU calls themselves. GPUDirect [18] enables the direct transfer of data between GPUs, bypassing the need to copy to the CPU. These enhancements collectively enhance the performance of inter-GPU data movement by eliminating unnecessary copying.

Bienz et al. . [4] explored the diverse factors influencing the cost of data movement in parallel systems, such as machine architecture, job partition, and neighboring tasks. With modern heterogeneous architectures introducing increased variability in data movement due to multiple viable paths for inter-GPU communication, their paper introduces performance models for these paths in inter-node communication. Their models also examine the trade-off between GPUDirect communication and copying to CPUs. The authors take their models to present a novel optimization strategy for inter-node communication, which is then used to demonstrate performance improvements for MPI collective operations.

Chen-Chun et al. [8] refined the Alltoall and Alltoallv communication algorithms for GPU systems, significantly enhancing the handling of the communication bottleneck present in high-performance computing (HPC) and deep learning applications. Their work capitalizes on Inter-Process Communication (IPC) mechanisms alongside contemporary GPU functionalities, introducing hybrid designs aimed at optimizing both intra-node and inter-node communication efficiencies. These designs incorporate kernel-based and memcpy-based IPC strategies tailored for varying message sizes. However, the scope of message sizes evaluated in their study is comparatively limited relative to the larger message sizes we consider. Consequently, the extensive variability in message sizes in accordance to Table 2 and the diverse optimization strategies might limit the direct comparability of their findings to those presented in our work.

It is worth noting that Open MPI enhances the Alltoallv algorithm by adjusting its optimization according to the size of the communicator (observed by reading source code) [10]. On the other hand, we do not know the version or optimizations that Cray Mpich and Spectrum MPI's Alltoallv are doing since they are closed sources.

3 Methodology

This paper presents a variety of unique Alltoallv operations for heterogeneous architectures, combining a variety of our baseline algorithms with both CUDA-Aware and Copy-to-CPU data movement approaches, as shown in Table 1. The baseline algorithms explored in this paper, in additional to the pairwise exchange and non-blocking approaches, include multi-pair blocking exchange, multi-pair nonblocking exchange, and multi-pair test exchange. As noted previously, there are trade offs to the pairwise exchange and non-blocking Alltoallv algorithms. The pairwise exchange algorithm limits the amount of data sent at any given time, but incurs large synchronization costs in applications with load imbalances. On the other hand, the non-blocking algorithm minimizes synchronization overheads by initializing all communication at one time, but can incur significant costs associated with network contention and queue search overheads. We address this challenge with a multi-pair blocking exchange, which communicates to only a subset of other processes at once, the number of which is denoted by `stride`. This reduces overloading the network which may cause packet delays and network contention.

While this allows more flexibility than the standard pairwise exchange, it still results in synchronization costs. If one of the many `stride` processes is not ready, the communicating process waits idly. The multi-pair non-blocking exchange further improves upon existing approaches by waiting for any one of the `stride` processes to finish and then communicating to the next, allowing the algorithm to continue even if some processes have not yet started their portion of the `MPI_Alltoallv`. Further, we have devised CUDA-Aware versions of all MPI Advance's discussed baseline Alltoallv algorithms. These CUDA-Aware adaptations use the underlying base algorithms but harness the advantages of

Table 1. A comparison of Alltoallv communication strategies. A new strategy is one that is implemented using MPI advance.

Alltoallv Strategy	New?	Transfer	Baseline Algorithm
CUDA-Aware Waitany Stride	Yes	From/To GPU	Multi-pair Nonblocking
CUDA-Aware Waitall Stride	Yes	From/To GPU	Multi-pair Blocking
CUDA-Aware Pairwise	Yes	From/To GPU	Pairwise
Copy-to-CPU Nonblocking	Yes	Copy to/from CPU	Multi-pair Nonblocking
Copy-to-CPU Pairwise	Yes	Copy to/from CPU	Pairwise
CUDA-Aware MPI	No	MPI Specific	MPI Implementation Specifc
Copy-to-CPU MPI	Hybrid	Copy to/from CPU	Copy-to-CPU then MPI Implementation

GPU capabilities, such as Direct GPU, Copy-to-CPU modes of data transfers, to enhance the efficiency of data exchanges.

For our experiments, a stride of five was used in our results because by keeping the stride at five, the algorithm sends only a subset of messages at any given time, helping to prevent network congestion and packet delays that are more likely with larger strides. In addition, this smaller stride helps reduce the probability of network contention, where multiple messages compete for the same network resources simultaneously.

In contrast, larger strides such as 10 or 15 might increase contention and queue search costs, as they involve a higher volume of concurrent communications, leading to more waiting time. Thus, a stride of five strikes a balance by limiting synchronization overhead while avoiding excessive network load.

3.1 Alltoallv Algorithms

The three variations of multi-pair exchanges presented in this paper are detailed in the remainder of this section.

- **Multi-pair Blocking Exchange**
  The multi-pair blocking exchange algorithm waits for an entire batch to complete before exchanging the next batch. This method uses `MPI_Waitall` to wait on all messages in a given batch to complete. This approach is an implementation of Algorithm 2, but uses the `stride` value mentioned above.
- **Multi-pair Non-blocking Exchange**
  This strategy waits for any message in a batch to complete before sending the next message. This algorithm uses `MPI_Waitany` to wait for a completion within a given batch. Rather than sending an entire subsequent batch, it will send one more message as soon as a single message completes. This allows for further reduction in synchronization over the Waitall approach, as if any of the processes within a single batch is lagging, subsequent messages can still be initiated.

– **Multi-pair Test Exchange**
  This strategy is similar to the previous strategy, but it replaces the call to `MPI_Waitany` with a call to `MPI_Testany`.

3.2 CUDA-Aware Alltoallv Extensions

In our methodology, we employ CUDA-Aware algorithms that call the MPI Advance Alltoallv algorithms discussed in Section 3.1. This approach enables data in GPU memory to be passed directly to the MPI routines, utilizing GPUdirect where available for direct data transfers between GPUs and the network, rather than moving data to and from CPU memory across nodes. All sends and receives throughout the underlying method are then initiated with GPU data, allowing for the data to be send via GPUDirect if the underlying MPI implementation chooses that path. Note, these algorithms are referred to as CUDA-Aware rather than GPUDirect as it is unclear if underlying algorithms are always using GPUDirect as they are not open-source, but tests indicate GPUDirect is used at majority of the time.

Algorithm 3 CUDA-Aware Alltoallv Setup

Input: f // alltoallv_ftn - any cpu underlying alltoallv function

// Arguments passed to CUDA_aware_alltoallv Setup:
f, sendbuf, sendcounts, sdispls, // Send buffer details
sendtype, recvbuf, recvcounts, // Receive buffer details
rdispls, recvtype, comm // Communication context
Output: // Status of the operation f
Optional: Perform any necessary preprocessing steps

// Function execution and return status
status = f(sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, `comm->global_comm`)

In this paper, Algorithm 3 serves as the foundational setup for all CUDA-Aware Alltoallv algorithms. The CPU-baseline multi-pair blocking exchange and multi-pair nonblocking exchange operate as underlying algorithms for MPI Advance's CUDA-Aware Alltoallv algorithms. As mentioned previously, the CUDA-Aware Alltoallv algorithms seek to maximize data transfer efficiency with GPU-Direct while, providing fallback mechanisms for systems without GPU-Direct. MPI chooses either to use GPU Direct or GPU-to-CPU.

– **CUDA-Aware Waitall Stride**
  The CUDA-Aware Waitall Stride instantiates the CUDA-Aware Alltoallv setup function in Algorithm 3, passing the multi-pair blocking exchange method as f.
– **CUDA-Aware Waitany Stride**

Algorithm 4 Copy to CPU Alltoallv Setup

Input: sendbuf, sendcounts, sdispls, sendtype, recvbuf, recvcounts, rdispls, recvtype, comm, cpu_recvbuf, cpu_sendbuf

Output: Error status after data transfers and MPI operation

Determine the number of processes in comm→global_comm Calculate size in bytes of sendtype and recvtype Initialize sendcount and recvcount to 0 **for** *each process in num_procs* **do**

∟ Accumulate total number of elements to send and receive

Calculate total bytes to send (total_bytes_s) and receive (total_bytes_r) Copy data from GPU to CPU memory (gpuMemcpyDeviceToHost) Execute alltoallv communication operation 'f' among CPUs Copy data back from CPU to GPU memory (gpuMemcpyHostToDevice) **return** *Error status (ierr)*

This differs from the CUDA-Aware Waitall Stride method by invoking multipair nonblocking as the underlying Alltoallv function in the setup Algorithm 3. Due to performance similarities in the underlying algorithm, a CUDA-Aware Test Stride method is not investigated in this paper, resulting in this being the sole underlying algorithm allowing processes to move on if any process within a batch is unavailable.

– **CUDA-Aware Pairwise**
  This method passes the CPU-baseline Alltoallv pairwise method to the CUDA-Aware setup Algorithm 3, resulting in all exchanges within a standard pairwise exchange algorithm being called directly from GPU memory.

3.3 Copy-to-CPU Alltoallv and Extensions

Copy-to-CPU extensions facilitate manually copying data between a GPU and a CPU before using a given Alltoallv baseline algorithm. By default, there is a large overhead to all Copy-to-CPU algorithms due to significant costs associated with the `cudaMallocHost` calls required to allocated data on the CPU. Therefore, all algorithms presented allocate these data once outside of MPI Advance implementations, showing the importance of persistent communication. This approach enables efficient reuse of pre-allocated buffers, reducing the overhead associated with repeated memory.

Algorithm 4, is a setup function or foundation algorithm for all Copy-to-CPU Alltoallv algorithms. Operation f is the underling collective operation, function, and this can be any Alltoallv algorithm from any MPI platform previously discussed. The underlying Alltoallv algorithm f invokes all required sends and receives with CPU memory. As a result, GPUDirect should never be used within these methods.

– **Copy-to-CPU Pairwise**, this algorithm invokes the set-up of Algorithm 4, and passes the baseline pairwise exchange algorithm in for f of Algorithm 3.
– **Copy-to-CPU Nonblocking**, the Copy-to-CPU Nonblocking algorithm invokes the setup from Algorithm 4, and passes the baseline multipair nonblocking algorithm in for f of Algorithm 3.

– **Copy-to-CPU MPI_Alltoallv**, this strategy integrates the Copy-to-CPU Alltoallv algorithm from MPI Advance with the underlying Alltoallv algorithm of the provided MPI framework. This approach not only leverages the memory optimization strategies inherent to MPI Advance's Copy-to-CPU Alltoallv algorithm but also has the potential to enhance the efficiency of any MPI implementation of Alltoallv.

4 Results for Structured Analysis

This section presents the performance impacts of communication optimizations of Alltoallv (GPU-Aware Alltoallv algorithms and optimized Copy-to-CPU strategies) within MPI Advance based on our structured analysis on Lassen and Tioga supercomputers.

4.1 Benchmark Overview

To support our structured investigation detailed in Sections 4.2- Section 4.5, a specialized benchmark was developed to evaluate and compare various implementations of the `MPI_Alltoallv` communication operation. This benchmark focuses on heterogeneous computing environments, leveraging GPU-accelerated and CPU-based strategies. Its primary objective is to provide a framework for analyzing the performance, scalability, and correctness of multiple `Alltoallv` variants under varying message sizes, process counts, and system configurations.

Experimental Design. The benchmark operates under both *strong scaling* conditions, maintaining a fixed problem size while increasing the number of processes, and *weak scaling* conditions, maintaining a constant per-process problem size while increasing the number of processes with the increasing problem size. This evaluates scalability across system configurations. Experiments were performed on both Lassen and Tioga supercomputers. The benchmark:

– Uses randomized data initialization.
– Performs correctness checks by comparing results to the baseline (standard MPI_Alltoallv).
– Executes multiple iterations (1000) to ensure robustness.
– Records performance metrics, including maximum execution time, message size transferred to highlight bottlenecks.

4.2 Structured Analysis on Lassen Supercomputer

– First we do a performance analysis of CUDA-aware Spectrum vs. all our proposed Alltoallv implementations across different process count and Message size on Lassen system. We do a similar performance analysis for Cray-MPICH vs. all our proposed Alltoallv implementations on Tioga system.

- The baseline for the comparisons made in Fig. 2 and Fig. 3 is standard MPI implementation commonly used in high-performance computing, is particularly CUDA-aware SpectrumMPI on Lassen system. These traditional implementations serve as a reference point to highlight the improvements brought by the proposed optimizations.
- Fig. 2 compares the performance of the optimized algorithms CUDA-aware SpectrumMPI against all the proposed Alltoallv implementations, particularly across varying message sizes. The baseline shows how standard implementations handle increasing message sizes, which helps demonstrate the advantages of the novel algorithms, particularly in scenarios with larger messages where our optimizations significantly reduce overhead and improve efficiency.
- Fig. 3 compares scalability across different process counts, using CUDA-aware SpectrumMPI as the baseline. The structured analysis reveals that while CUDA-aware SpectrumMPI performs well with smaller process counts, it struggles as the system scales up. In contrast, the proposed Copy-to-CPU Nonblocking algorithm continues to perform efficiently at larger process counts, indicating better scalability.
- We cannot conclude why Spectrum and Cray-MPICH perform the way they do since they are both closed sources. However, the performance of our algorithms is attributed to their nature as explained in the methodology.

4.3 Performance Analysis: Copy-to-CPU Nonblocking vs. CUDA-aware Spectrum on Lassen Supercomputer Across Varrying Message Sizes

Copy-to-CPU Nonblocking emerges as the top-performing algorithm, especially for larger message sizes. It outperforms the baseline **CUDA-aware Spectrum** and other alternatives, showcasing superior scalability and efficiency in Fig. 2.

Key Speedup Observations on Fig. 2 (Performance Analysis at 32 Nodes, 128 Processes (Constant) Across Different Message Sizes).

1. **Small Message Size (2,048)**:
   At smaller message sizes, **Copy-to-CPU Nonblocking** closely matches **CUDA-aware Spectrum**, with a near-identical execution time and a speedup of approximately **1.01x**. This minor advantage demonstrates that **Copy-to-CPU Nonblocking** is just as efficient at handling small-scale communication.
2. **Medium Message Size (4,096 bytes)**:
   For medium-sized messages, **Copy-to-CPU Nonblocking** experiences a temporary a slow down in performance relative to **CUDA-aware Spectrum**, achieving about **0.16x** the speed of the baseline. While there is a slowdown here, this is quickly recovered as the message size increases further.

3. **Large Message Size (524,288 bytes)**:
   As the message size increases to 524,288 bytes, **Copy-to-CPU Nonblocking** begins to demonstrate its strength, outperforming **CUDA-aware Spectrum** with impressive scalability and a speed up of **1.6**×, showing significant potential for handling larger data sizes. All other proposed Alltoallv algorithms also perform better than the baseline at this point.
4. **Very Large Message Size (8,388,608 bytes)**:
   At the largest message size of 8MB, **Copy-to-CPU Nonblocking** clearly dominates, with the fastest execution time among all algorithms. It performs better than **CUDA-aware Spectrum** by a significant margin, delivering **3x** faster execution times. This proves that **Copy-to-CPU Nonblocking** is particularly well-suited for large-scale data transfers, making it the best-performing algorithm in this test.
5. **Other Proposed Alltoallv Algorithms**:
   Although other proposed **Copy-to-CPU and CUDA-aware** algorithms, perform better than standard CUDA-aware Spectrum at some occasions, **Copy-to-CPU Nonblocking** has the most significant performance.
6. **Copy-to-CPU Spectrum Alltoallv**:
   Copy-to-CPU Spectrum Alltoallv, the optimized version of CUDA-aware Spectrum clearly performs better than standard CUDA-aware Spectrum at a small message size.

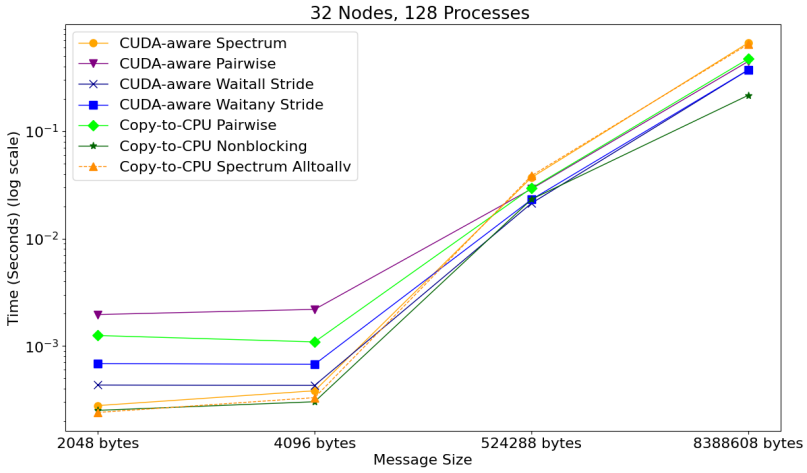

Fig. 2. The performance analysis of `CUDA-aware Spectrum` vs all our proposed `Alltoallv algorithms` across different message sizes on Lassen.

4.4 A Performance Comparison of CUDA-aware Spectrum with our Proposed Copy-to-CPU and CUDA-aware Implementations Across Varying Process Counts on Lassen Supercomputer

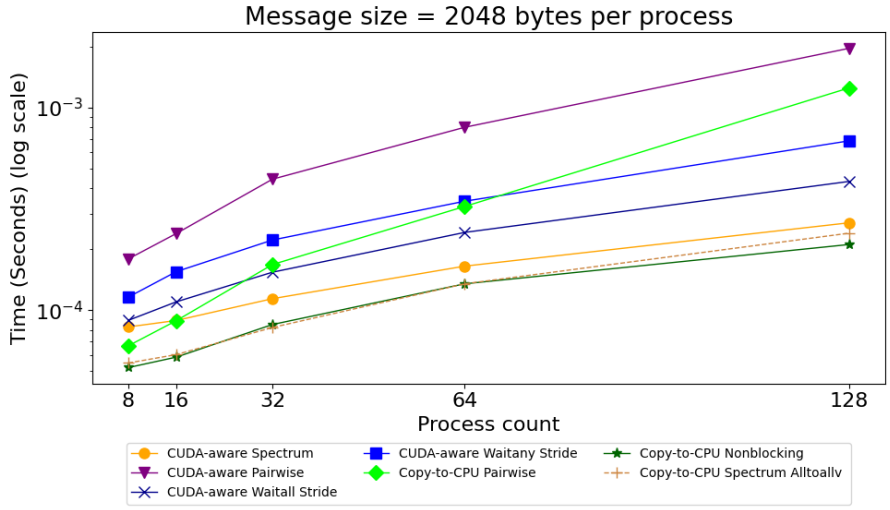

Fig. 3. The performance analysis of `CUDA-aware Spectrum` vs our proposed `Copy-to-CPU and our CUDa-aware implementations` across different process count on Lassen.

Key Observations on Fig. 3 (Scalability Comparison of our Algorithms Vs. Standard MPI Implementation Across Different Process Counts for a Message Size of 2,048 bytes per process).

1. In this performance analysis of multiple communication algorithms across various process counts, **Copy-to-CPU Nonblocking** emerges as the most efficient algorithm, exhibiting outstanding scalability and maintaining low execution times even as the number of processes increases. It consistently outperforms other algorithms, including **CUDA-aware Spectrum**, which serves as the baseline.
2. **Copy-to-CPU Nonblocking** proves to be the best choice for high-process environments, particularly at larger scales, where communication overhead can become a bottleneck.
3. **Copy-to-CPU Spectrum Alltoallv (optimized version of Spectrum Alltoallv)** is the second best performing algorithm. It outperforms **CUDA-aware Spectrum** and other algorithms.

4. **Copy-to-CPU Pairwise** also shows strong performance, remaining competitive with **Copy-to-CPU Nonblocking**, although it begins to show a more noticeable increase in execution time as the process count reaches 128, positioning it slightly behind the leader algorithm.
5. **Copy-to-CPU Spectrum Alltoallv** demonstrates stable, consistent performance across all process counts, but it fails to surpass the top algorithms like **Copy-to-CPU Nonblocking**.
6. **CUDA-aware Spectrum**, while not the fastest, offers good scalability and steady performance, making it a reliable baseline for comparison.
7. Overall, **Copy-to-CPU Nonblocking** is the most scalable and efficient option, particularly for environments with large numbers of processes, making it the ideal choice for high-performance computing tasks.

4.5 Structured Analysis Results on Tioga Supercomputer

This section analyzes the Scalabity of Cray-mpich and our proposed Alltoallv algorithms on Tioga.

- Similarly to the first part of this analysis, in the second phase of our analysis, we conducted another structured study on the Tioga system. Fig. 4 and Fig. 5, show the comparative performance of these optimized algorithms.
- The baseline for the comparisons made in Fig. 4 and Fig. 5 is Cray-MPICH Alltoallv.

4.6 The Performance Analysis of Cray-MPICH vs our Proposed Alltoallv Algorithms Across Different Process Counts on Tioga Supercomputer

Key Observations:

1. **Cray-MPICH Alltoallv** demonstrates stable performance at smaller process counts but exhibits a steady increase in execution time as the process count rises beyond 30, making it less suitable for larger process counts. In contrast, **CUDA-aware Waitall Stride (Blue Line)** is upto 5x faster than Cray-MPICH Alltoallv.
2. **CUDA-aware Waitall Stride** emerges as the most efficient algorithm, showing an initial increase in execution time but achieving a significant reduction between 30 and 40 processes. By maintaining the lowest execution time at 64 processes, **CUDA-aware Waitall Stride** proves to be the most scalable and efficient among the algorithms, outperforming **Cray-MPICH Alltoallv** at larger scales.
3. **Copy-to-CPU Nonblocking** also shows strong performance, with a steady increase in execution time that remains competitive with the baseline, although it doesn't achieve the same scalability as **CUDA-aware Waitall Stride** beyond 30 processes.

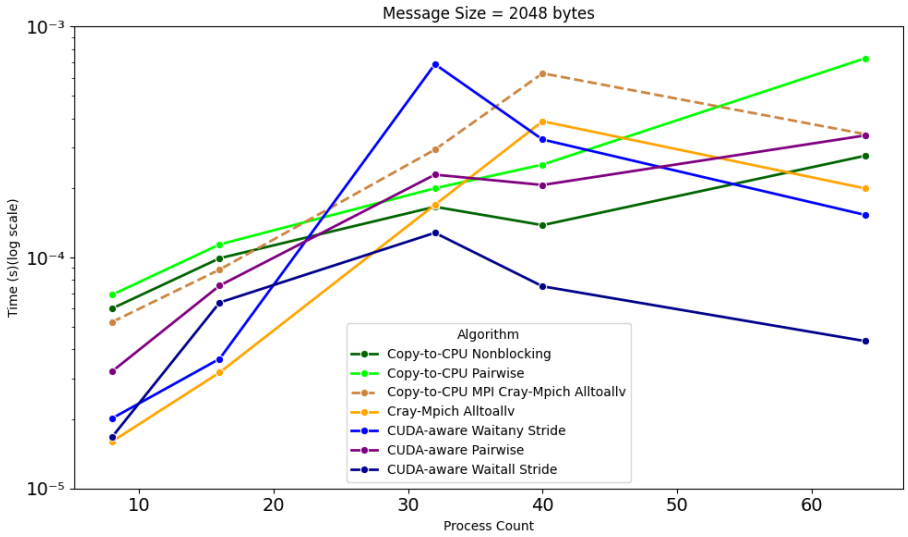

Fig. 4. The performance analysis of `Cray-MPICH` vs our proposed `Alltoallv` algorithms across different number of processes on Tioga Cluster.

4. Falling into the middle of performance, **Copy-to-CPU Pairwise** exhibits a gradual increase in execution time that is slightly higher than **Cray-MPICH Alltoallv** but remains competitive overall, though its scaling is less efficient than the leading algorithms.
5. **CUDA-aware Pairwise** display less scalability, especially at larger process counts.
6. **CUDA-aware Waitany Stride** presents a fluctuating trend, with an initial steep rise in execution time, a recovery around 40 processes and it emerges the second best algorithm at 64 processes.
7. Lastly, **Copy-to-CPU MPI Cray-MPICH Alltoallv (optimized version of Cray-MPICH Alltoallv)** remains competitive with **Copy-to-CPU Nonblocking** at smaller process counts, though it scales less efficiently than the baseline and other top performing algorithms beyond 40 processes.
8. In summary, **CUDA-aware Waitall Stride** leads in scalability and efficiency, significantly outperforming **Cray-MPICH Alltoallv** at higher process counts, followed closely by **Copy-to-CPU Nonblocking**, **CUDA-aware Waitany especially at 64 processes**. **Cray-MPICH Alltoallv** remains the baseline performer but its outperformed in scalability when compared to **CUDA-aware Waitall Stride**.

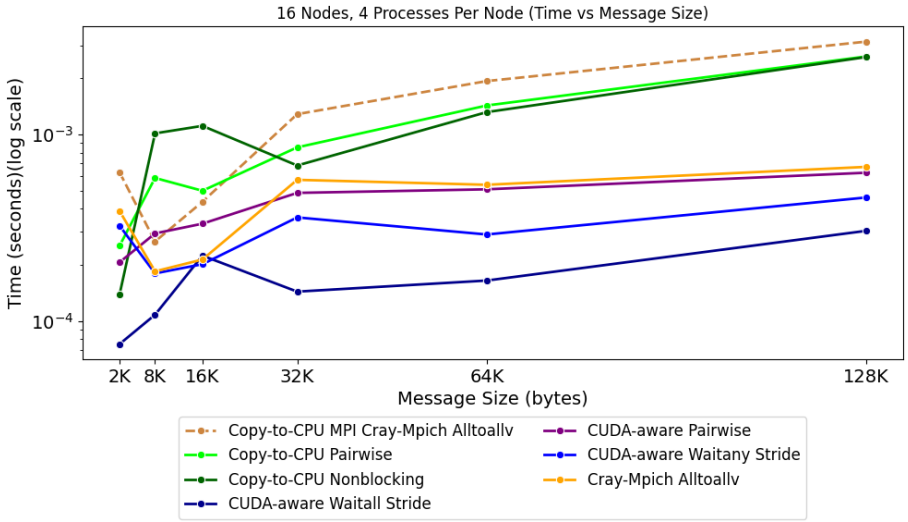

Fig. 5. The performance analysis of `Cray-Mpich` vs. our proposed `Copy-to-CPU` and `CUDA-aware` algorithms across different message sizes on Tioga Cluster.

4.7 Performance Analysis of all Alltoallv Algorithms for Varrying, Increasing Message Sizes on Tioga Supercomputer

Fig. 5 presents an in-depth performance comparison of various communication algorithms as message sizes increase, using **Cray-Mpich Alltoallv** (orange) as the baseline comparison algorithm. **CUDA-aware Waitall Stride** is 2 × faster than **Cray-Mpich Alltoallv** (orange). **CUDA-aware Waitall Stride** and **CUDA-aware Waitany Stride** emerge as the most efficient algorithms as compared to **Cray-Mpich Alltoallv** which is the baseline comparison. Both demonstrate exceptional scalability, maintaining low and stable execution times across all message sizes. Their behavior as message size increases makes them ideal for applications requiring high scalability. Compared to the baseline **Cray-Mpich Alltoallv**, which maintains a consistent but moderate execution time across all message sizes, our CUDA-aware algorithms significantly outperform other algorithms, especially as message sizes grow. **Cray-Mpich Alltoallv** serves as a stable but less optimal choice when low execution time is critical.

Copy-to-CPU Nonblocking, **Copy-to-CPU Pairwise**, and **Copy-to-CPU Waitany** display moderate scalability. While they remain competitive at smaller message sizes (2K–8K bytes), their execution times increase notably as message sizes exceed 16K bytes, falling behind the CUDA-aware algorithms. Our Copy-to-CPU algorithms provide reasonable performance, but do not match the scalability of our CUDA-aware options on the Tioga system.

CUDA-aware Pairwise performs closely better than **Cray-MPICH Alltoallv** but with slightly higher execution times, remaining less efficient than **CUDA-aware Waitall Stride** and **CUDA-aware Waitany Stride**.

The structured tests presented in Sections 4.1 through 4.7 evaluate GPU-aware and Copy-to-CPU `MPI_Alltoallv` algorithms across various configurations on Lassen and Tioga systems, highlighting performance, scalability based on message sizes and process configurations.

1. **Lassen: GPU-Aware vs. Copy-to-CPU** - Copy-to-CPU Nonblocking algorithms consistently outperform CUDA-aware SpectrumMPI for small, medium, and larger message sizes on Lassen, achieving up to a 3× speedup. This is due to optimized memory management and minimized repeated data transfer overhead for large messages.
2. **Tioga: Performance Scaling** - CUDA-aware Waitall Stride on Tioga demonstrates the best scalability for process count above 30 processes. For high process counts, CUDA-aware Waitall Stride maintains lower execution times and outperforms Cray-MPICH Alltoallv, which shows limitations beyond 30 processes.
3. **Impact of Message Size** - As message sizes increase, CUDA-aware Waitany Stride and Waitall Stride algorithms maintain low execution times on Tioga, proving to be efficient for applications with large messages. Copy-to-CPU approaches outperform GPU-aware algorithms on Lassen.

These results show the necessity for a polyalgorithmic strategy in optimizing `MPI_Alltoallv` operations. It shows performance differences between GPU-aware and Copy-to-CPU methods based on specific workload size, and system architecture. This is a great foundation for applications in high-performance computing environments. This can be a motivation for real world experiments.

We used a smaller range of message sizes for Tioga results because Tioga uses a Cray interconnect designed for scalability, but large messages from multiple processes can still strain the network. Cray interconnects are optimized for small, medium messages rather than very large, simultaneous transmissions [11]. This explains why we chose to use Lassen for the real World Applications since our configurations test very large messages.

5 Real World Applications

Having done a structured analysis of all algorithms, we further made a performance analysis on real world applications as detailed in the rest of this section. Fast Fourier Transforms (FFTs) are widely used to solve a variety of problems in scientific and engineering computations [7]. HeFFTe [1], a state-of-the-art parallel FFT solver, efficiently computes discrete Fourier transforms (via FFT algorithms) on emerging scalable architectures. As process counts increase, HeFFTe is bottle-necked by a data transpose, data reshuffles in which all processes exchange data of varying sizes with all other processes. Typically, this exchange is implemented with an `MPI_Alltoallv`, the cost of which of increases with

increased process count and message sizes. Each emerging parallel computers bring added computational power through increased core counts or accelerator performance. Therefore, as FFT solvers such as HeFFTe are scaled across increasingly large parallel systems, the Alltoallv bottleneck increases in dominance. The `MPI_Alltoallv` operation consists of non-symmetric data exchanges between each set of processes in a given communicator. As with all collective operations, the cost of Alltoall-type operations increases with process count. With current MPI implementations as observed from our structured study Fig. 4, the cost of Alltoallv increases with process count so the algorithms become a dominant cost at large scales for certain applications and algorithms (like FFTs); thus, performance optimization is an important issue.

5.1 Problem Setup

Beatnik[1] is a benchmark for global communication based on Pandya and Shkoller's 3D fluid interface Z-Model [16] in the Cabana/Cajita mesh framework [19]. The Beatnik library is **bottlenecked by the FFT solves within HeFFTe**, which involve a lot of data reshuffles that in turn rely on `MPI_Alltoallv` for communication between different processes.

The performance impacts of a variety of Alltoallv operations on real application like FFT solvers such as HeFFTe, are presented in this section. We have CPU baseline Alltoallv algorithms tested on the CPUs of AMD-Epyc processor cluster. However, this discussion is focusing mainly on CUDA-Aware Alltoallv algorithms analyzed on a Lassen, an IBM POWER9 and NVIDIA GPU V100 system at Lawrence Livermore National Laboratory. We assess the efficiency of various algorithms by varying node counts, process configurations per node, and message sizes while measuring the maximum time taken by any process to execute the `MPI_Alltoallv`. To precisely track the duration of each `MPI_Alltoallv` operation within HeFFTe, we employed profiling tools such as Caliper [6] for instrumentation.

We performed experiments using various iterations of MPI Advance's algorithms, namely, CUDA-Aware, Copy-to-CPU oriented Alltoallv algorithms, alongside Open MPI's Alltoallv, CUDA-Aware Spectrum and CUDA-Aware MVAPICH2 Alltoallv algorithms. These tests were conducted on systems ranging from 2 to 128 nodes, with the number of processes ranging from 8 to 512. The size of the problems addressed was increased proportionally to the square root of the total process count, indicating a weak scaling approach. The simulation setup with Beatnik used a problem size with a grid size of 4,096, which was scaled up by varying the number of processes. We increased the grid size linearly with the number of processes, using the square root factor of the number of processes. For instance, when we used 64 processes, the mesh dimensions were 32,768 by 32,768, which is equivalent to $4,096 \times 8$ by $4,096 \times 8$ bytes. When we used 32 processes, the square root was about 5.66, resulting in a mesh dimen-

[1] https://github.com/CUP-ECS/beatnik/

Table 2. Resource allocation across different nodes, detailing grid and message sizes. Grid dimensions are weakly scaled with the square root of processes ($\sqrt{P}$) times a constant grid size (G=4,096). Message size reflects total data for all processes, determined by the squared grid dimension ($N^2 \times 8$ bytes).

Nodes	GPUs	Processes (P)	$N = \sqrt{P} \times G, G = 4096$
1	4	4	$2.0 \times G$
2	8	8	$2.8 \times G$
4	16	16	$4.0 \times G$
8	32	32	$5.7 \times G$
16	64	64	$8.0 \times G$
32	128	128	$11.3 \times G$
64	256	256	$16.0 \times G$
128	512	512	$22.6 \times G$

sion of 23,170 by 23,170. The exact problem sizes are outlined in Table 2; any experiments that deviate from these values are noted.

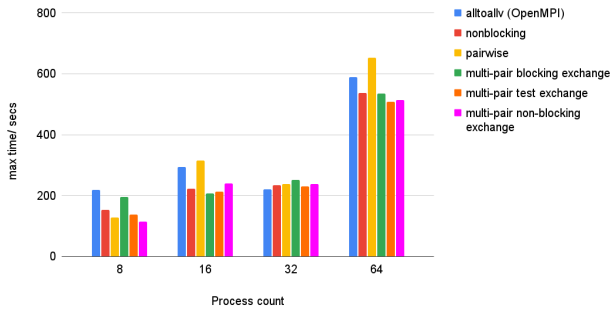

Fig. 6. A comparison of Alltoalv times using algorithms from Open MPI and MPI Advance. Problem sizes are in accordance with Table 2. All runs were on one node.

Analysis of Baseline Alltoallv Algorithms. First we analyzed MPI Advance's baseline versions of Alltoallv as compared to Open MPI's Alltoallv on CPUs of AMD-Epyc processor cluster.

In Fig. 6, our observations indicate that MPI Advance's baseline algorithms, such as multi-pair nonblocking exchange and multi-pair test exchange, perform well for both small and large numbers of processes. This can be seen for 8, 16, and 64 processes, as compared to other versions of baseline MPI Advance Alltoallv algorithms and Open MPI's Alltoallv which is selecting basic linear algorithm for 8, 16, 32 processes and pairwise for 64 processes. We also observed that MPI Advance's baseline versions, such as Alltoallv pairwise, perform better with

smaller grid and problem sizes, while MPI Advance's multi-pair non-blocking exchange and multi-pair test exchange perform well with large or small grid sizes. The performance difference between multi-pair non-blocking exchange and multi-pair test exchange is slight. These early observations helped drive our strategy for other problem setup and experimentation.

MPI Advance GPU-Aware Alltoallv Algorithms. We observed that MPI Advance's CUDA-Aware (GPUDirect) algorithms perform better than Copy-to-CPU algorithms for only medium sized messages; this is also observed by Bienz et al. [4]. Copy-to-CPU algorithms outperform all other algorithms on larger message size and number of processes with 2×-3× speedup or more as you are yet to see in the presented results. This performance is attributed to two main reasons in this paper:

(1) Copy-to-CPU algorithms outperform GPUDirect algorithms for larger message sizes and numbers of processes because, as the number of messages increases, the latency associated with GPUDirect communication's higher start-up costs significantly impacts performance. In contrast, copying data to the CPU, despite its initial overhead, becomes more efficient for managing multiple messages due to its lower per-message latency [4].

(2) The Copy-to-CPU algorithms presented in this paper eliminate the need for repeated CPU memory allocations, streamlining the process and significantly enhancing their speed unlike the traditional versions. This efficiency improvement is due to reduced overhead from avoiding these memory operations. This is a novel optimization implementation of Alltoallv discussed by this paper.

We offer a variety of setup options for our computing tasks, with configurations that can include anywhere from 2 to 64 individual computing nodes. For each of these nodes, we assign 8 to 256 processes to handle the computational work. The system is equipped to run 4 processes simultaneously; each of these processes has its own dedicated GPU, resulting in 4 GPUs being allocated per node. Some of our configurations were carried out using weak scaling. As we increased the number of processes, the problem size is also scaled proportionally to the square root of the number of processes, as shown in Table 2.

This approach ensures that as we add more computing power by increasing the number of processes and nodes, the size of the problem each process handles grows at a rate that maintains the overall computational load per process. This ensures that every process has a fair and manageable amount of work, preventing scenarios where some processes are overloaded while others have too little to do. This strategy helps in achieving efficient scaling of computational tasks.

Overall Speedup in the Beatnik Application. Fig. 7 shows the performance of the Beatnik application across a few configurations. We observed a 2-3× speedup as compared to the base line algorithm (CUDA-aware spectrum) in Beatnik's overall performance after using MPI Advance's Alltoallv in HeFFTe. This speedup can be attributed to the nature of the algorithms, as explained in Section 3.

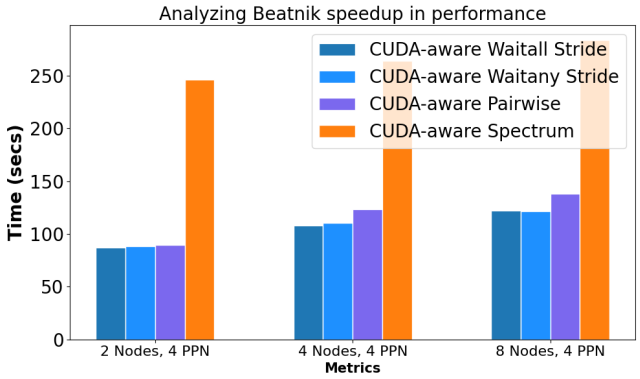

Fig. 7. Maximum time per MPI process taken by Beatnik simulations with various versions of Alltoallv. Problem sizes are in accordance with Table 2.

Comparing MPI Advance's CUDA-Aware and Copy-to-CPU Algorithms. In Fig. 8, when we added two new algorithms, Copy-to-CPU Nonblocking and Copy-to-CPU Pairwise, we observed that the Copy-to-CPU Nonblocking algorithm outperforms all other CUDA-Aware algorithms. Therefore, this paper presents Copy-to-CPU Nonblocking algorithm as the best performing algorithm for transferring various large message sizes.

The **significance of the other algorithms diminished, and therefore, they were not included in further analysis**. In addition, Copy-to-CPU Nonblocking algorithm outperformed CUDA-Aware Spectrum Alltoallv at large problem sizes, as shown in the bottom row of graphs in Fig. 9. We noted a **2**×—**3**× speed up in the maximum time per MPI process as compared to the SpectrumMPI's CUDA-Aware Alltoallv for the weak scaling configurations.

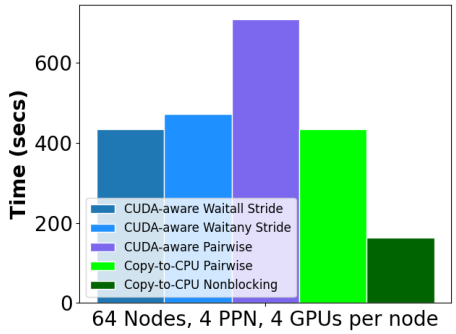

Fig. 8. Maximum time per MPI process taken by MPI Advance's copy to CPU algorithms vs. all CUDA-Aware algorithms on 64 nodes with 256 processes. Problem size is in accordance with Table 2.

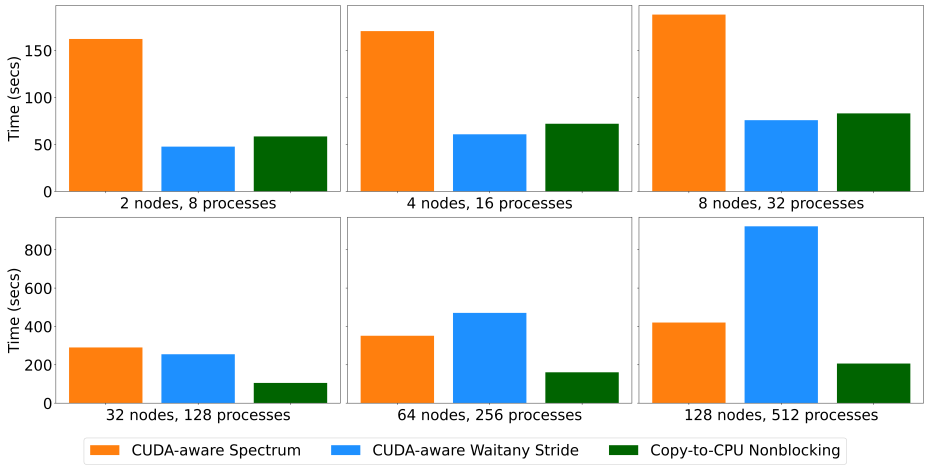

Fig. 9. Performance of Alltoallv algorithms for weak-scaling scenarios, all GPU-oriented. For all graphs, there were four processes and four GPUs per node. Problem sizes are in accordance with Table 2.

We provide graphs depicting the scalability of various algorithms, based on the maximum time taken by any process to execute the MPI Alltoallv command across configurations ranging from two nodes, with four processes for each node to 128 nodes with 512 processes, as shown in Fig. 9. Our observations reveal that MPI Advance's CUDA-Aware algorithms excel with medium-sized messages, as evident in the **top row** of graphs in Fig. 9. For large messages, MPI Advance's Copy-to-CPU Nonblocking algorithm is the most effective, with supporting data shown in **bottom row** of Fig. 9.

The performance demonstrated from the graphs emphasizes that there is no one-size-fits-all algorithm; instead, the relative performance of each algorithm depends on the input parameters. Despite this, all of the MPI Advance algorithms presented outperform CUDA-Aware Spectrum at all time.

Strong Scaling Results. In Fig. 10, we present strong-scaling results with a fixed problem size, a grid size of $12,288 \times 12,288$; we scaled out from 1 node to 64 nodes with four processes per node and four GPUs per Node.

- In Fig. 10, at a smaller number of processes, all of MPI Advance's algorithms performed better than CUDA-Aware SpectrumMPI. As the problem scales, the message sizes will clearly decrease while the message count increases.
- While all of MPI Advance's Copy-to-CPU Alltoallv algorithms surpass the performance of CUDA-Aware SpectrumMPI's Alltoallv when used with a smaller number of processes (corresponding to a larger message size per process), the situation changes at the extremes of strong scaling. In such cases, SpectrumMPI's CUDA-Aware Alltoallv begins to outperform all of MPI Advance's CUDA-Aware algorithms.

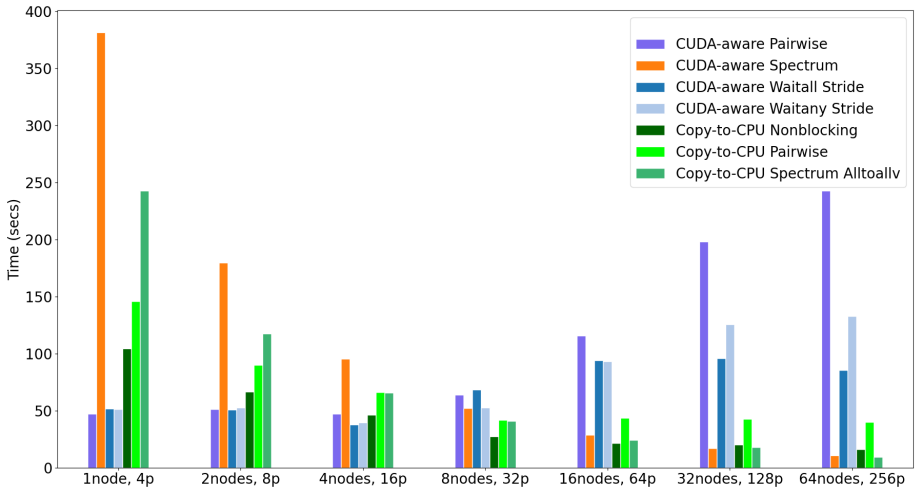

Fig. 10. Maximum times are shown for Alltoallv per MPI process taken by one node, four processes through 64 nodes, 256 processes (strong scaling the problem grid of 12, 288 × 12, 288 for one node, four processes). Use cases are GPU-centric.

– This motivated us to develop a new algorithm (Copy-to-CPU `MPI_Alltoallv`), which can embed any underlying Alltoallv from any MPI platform.
Copy-to-CPU `MPI_Alltoallv`'s "algorithmic wrapping" uses MPI Advance library for memory management; in these experiments it calls SpectrumMPI's CUDA-Aware Alltoallv as the baseline algorithm.
In strong scaling scenarios, as depicted in Fig. 10, the Copy-to-CPU Alltoallv, using SpectrumMPI's CUDA-Aware Alltoallv as the baseline algorithm (referred to as Copy-to-CPU Spectrum), outperforms CUDA-Aware Spectrum and other algorithms.

6 Conclusion

We compared the performance of different versions of the `MPI_Alltoallv` operation enabled by the MPI Advance library against Open MPI, SpectrumMPI, and Cray-MPICH. We made structured tests for the algorithms and achieved our goal of making the `MPI_Alltoallv` operations faster, even when applied to real-world applications in FFTs such as HeFFTe. The structured analysis conducted on the LLNL's Lassen and Tioga supercomputers highlight distinct advantages based on message size and system configuration:

1. **Copy-to-CPU Nonblocking on Lassen:** This algorithm shows consistent performance advantages over CUDA-aware SpectrumMPI for large message sizes and high process counts, achieving up to a 3× speedup. This improvement is attributed to optimized memory handling and reduced data transfer overhead for large messages.

2. **CUDA-aware Waitall Stride on Tioga:** On Tioga, CUDA-aware Waitall Stride emerges as the most scalable option with upto 5× speedup, especially for message sizes above 16K bytes and larger process counts. This algorithm outperforms Cray-MPICH Alltoallv at higher process counts by maintaining low execution times.
3. **Message Size analysis:** On Tioga, the structured tests indicate that as message sizes increase, CUDA-aware Waitall Stride, CUDA-aware Waitany Stride maintain low execution times on Tioga. On Lassen, Copy-to-CPU methods are more scalable for larger message sizes as compared to the baseline CUDA-aware SpectrumMPI.

We further tested real world applications; we tested baseline algorithms on the CPUs of AMD-Epyc processor cluster and focusing mainly on CUDA-aware algorithms on Lassen system. The HeFFTe library was modified by replacing the `MPI_Alltoallv` routine with Copy-to-CPU and CUDA-aware optimized algorithms from the MPI Advance library. The Beatnik library was application successfully integrated with the modified HeFFTe. Our configurations included both weak scaling and strong scaling. These results show a significant speedup in the CPU-baseline collective routines, and in some cases we registered *2×—3× speedup* in Alltoallv in HeFFTe when we applied our Copy-to-CPU and CUDA-aware Alltoallv algorithms. This in turn speed up Beatnik, which was used as a driver application for HeFFTe in this study.

Our observations suggest that there is no single algorithm that is best for `MPI_Alltoallv` in all situations (a polyalgorithm is warranted): the performance of algorithms depends on message size, grid size, system architecture and the number of processes. For example On Lassen, the Copy-to-CPU Non-blocking algorithm performs best for large messages, large process count. Note that the Copy-to-CPU strategies in MPI Advance are a combination of MPI Advance's memory management optimizations and the underlying Alltoallv algorithms from a given MPI implementation. MPI Advance's CUDA-Aware Alltoallv algorithms perform best for medium size messages. The hybrid algorithms and Copy-to-CPU Spectrum perform best for small messages. Considering weak scaling configurations, most of MPI Advance's CPU baseline algorithms, Copy-to-CPU and CUDA-aware Alltoallv algorithms performed better than the native MPI implementations' algorithms. In contrast, for strong scaling configurations, at low process counts, MPI Advance's CUDA-aware algorithms were about 2×—3× faster than CUDA-aware Spectrum algorithm and MPI Advance's Copy-to-CPU Spectrum Alltoallv. However, at an extreme degree of strong scaling (small message size), Copy-to-CPU Spectrum Alltoallv performs better than CUDA-aware SpectrumMPI in all our experiments. It should be noted that CUDA-aware Spectrum is a great algorithm for smaller message sizes. Overall, we provided effective ways for optimizing MPI_Alltoallv communication which has the potential to enhance the performance of various parallel computing applications in the field of HPC, including but not limited specifically to those relying on multi-dimensional FFTs demonstrated here via HeFFTe.

Acknowledgment

This work was performed with partial support from the National Science Foundation under Grants Nos. 2405142, 2412182, and 2201497, 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, the U.S. Department of Energy's National Nuclear Security Administration.

References

1. Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. Heffte: Highly efficient fft for exascale. In *Computational Science – ICCS 2020: 20th International Conference, Amsterdam, The Netherlands, June 3–5, 2020, Proceedings, Part I*, page 262–275, Berlin, Heidelberg, 2020. Springer-Verlag.
2. Amanda Bienz, Shreeman Gautam, and Amun Kharel. A locality-aware bruck allgather. In *Proceedings of the 29th European MPI Users' Group Meeting*, EuroMPI/USA'22, page 18–26, New York, NY, USA, 2022. Association for Computing Machinery.
3. Amanda Bienz, Luke N. Olson, and William D. Gropp. Node-aware improvements to allreduce. In *Proceedings of ExaMPI 2019*, pages 19–28, United States, November 2019. IEEE.
4. Amanda Bienz, Luke N. Olson, William D. Gropp, and Shelby Lockhart. Modeling data movement performance on heterogeneous architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2021.
5. Amanda Bienz, Derek Schafer, and Anthony Skjellum. MPI Advance : Open-Source Message Passing Optimizations. *arXiv e-prints*, page arXiv:2309.07337, September 2023.
6. 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. IEEE, 2016.
7. E Oran Brigham. *The fast Fourier transform and its applications*. Prentice-Hall, Inc., 1988.
8. Chen-Chun Chen, Kawthar Shafie Khorassani, Quentin G. Anthony, Aamir Shafi, Hari Subramoni, and Dhabaleswar K. Panda. Highly efficient alltoall and alltoallv communication algorithms for gpu systems. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 24–33, 2022.
9. Ke Fan, Thomas Gilray, Valerio Pascucci, Xuan Huang, Kristopher Micinski, and Sidharth Kumar. 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*, HPDC '22, page 172–184, New York, NY, USA, 2022. Association for Computing Machinery.

10. 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. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.
11. Nathan T. Hjelm, Samuel K. Gutierrez, and Manjunath Gorentla Venkata. On the current state of open mpi on cray systems. In *Proceedings of the Cray User Group*, Los Alamos, NM, 2014. Los Alamos National Laboratory and Oak Ridge National Laboratory. Los Alamos National Laboratory Report: LA-UR-14-22496.
12. Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing multi-leader-based allgather algorithms for multi-core clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–8, 2009.
13. N.T. Karonis, B.R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 377–384, 2000.
14. Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack Dongarra. Hierknem: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 970–982. IEEE, 2012.
15. Seyed H. Mirsadeghi and Ahmad Afsahi. Topology-aware rank reordering for mpi collectives. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1759–1768, 2016.
16. Gavin Pandya and Steve Shkoller. 3d interface models for rayleigh-taylor problems. *arXiv preprint arXiv:2201.04538*, 2022.
17. P. Patarasuk and X. Yuan. Bandwidth efficient all-reduce operation on tree topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007.
18. Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K. Panda. Efficient inter-node mpi communication using gpudirect rdma for infiniband clusters with nvidia gpus. In *2013 42nd International Conference on Parallel Processing*, pages 80–89, 2013.
19. Stuart Slattery, Samuel Reeve, Christoph Junghans, Damien Lebrun-Grandié, Robert Bird, Guangye Chen, Shane Fogerty, Yuxing Qiu, Stephan Schulz, Aaron Scheinberg, Austin Isner, Kwitae Chong, Stan Moore, Timothy Germann, Jim Belak, and Susan Mniszewski. Cabana: A performance portable library for particle-based simulations. *Journal of Open Source Software*, 7:4115, 04 2022.
20. Jesper Larsson Träff. Efficient allgather for regular smp-clusters. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, EuroPVM/MPI'06, page 58–65, Berlin, Heidelberg, 2006. Springer-Verlag.
21. Jesper Larsson Träff and Sascha Hunold. Decomposing mpi collectives for exploiting multi-lane communication. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 270–280, 2020.