# Designing and Optimizing GPU-aware Nonblocking MPI Neighborhood Collective Communication for PETSc*

Kawthar Shafie Khorassani, Chen-Chun Chen, Hari Subramoni and Dhabaleswar K. Panda
Department of Computer Science and Engineering
The Ohio State University, Columbus, Ohio
E-mail: {shafiekhorassani.1, chen.10252}@osu.edu, {subramon, panda}@cse.ohio-state.edu

***Abstract*—MPI Neighborhood collectives are used for non-traditional collective operations involving uneven distribution of communication amongst processes such as sparse communication patterns. They provide flexibility to define the communication pattern involved when a neighborhood relationship can be defined. PETSc, the Portable, Extensible Toolkit for Scientific Computation, used extensively with scientific applications to provide scalable solutions through routines modeled by partial differential equations, utilizes neighborhood communication patterns to define various structures and routines.**

**We propose GPU-aware MPI Neighborhood collective operations with support for AMD and NVIDIA GPU backends and propose optimized designs to provide scalable performance for various communication routines. We evaluate our designs using PETSc structures for scattering from a parallel vector to a parallel vector, scattering from a sequential vector to a parallel vector, and scattering from a parallel vector to a sequential vector using a star forest graph representation implemented with nonblocking MPI neighborhood alltoallv collective operations. We evaluate our neighborhood designs on 64 NVIDIA GPUs on the Lassen system with Infiniband networking, demonstrating 30.90% improvement against a GPU implementation utilizing CPU-staging techniques, and 8.25% improvement against GPU-aware point-to-point implementations of the communication pattern. We also evaluate on 64 AMD GPUs on the Spock system with slingshot networking and present 39.52% improvement against the CPU-staging implementation of a neighborhood GPU vector type in PETSc, and 33.25% improvement against GPU-aware point-to-point implementation of the routine.**

***Index Terms*—Neighborhood Collectives, GPU, MPI**

## I. Introduction

Traditional collective operations in the Message Passing Interface (MPI) typically involve a root process synchronizing with multiple processes or all processes within a communicator, synchronizing with each other. The communication pattern is defined and involves all the processes. This leads to non-traditional operations involving many processes but not adhering to a unified communication pattern being implemented using point-to-point operations. This is an inefficient solution that can lead to sub-optimal performance and unloads the responsibility of defining the communication to the user or application developer rather than utilizing a collective call pre-implemented and optimized for any sparse communication pattern workload. This problem has been referred to in past work as sparse communication and addressed in the MPI 3.0 standard with the introduction of Neighborhood collective operations. Neighborhood collectives allow for sparse communication patterns involving unevenly balanced communication loads and patterns. Every process in the communicator can have a different number of neighbors it is sending to and receiving from with varying data loads.

Neighborhood Collective operations within MPI enable communication over process topologies [1]. MPI process topologies are an additional component that can be provided for a communicator within MPI, representing a virtual topology related to the communication. In a scenario where linear ranking is not suitable (i.e. when the pattern of communication is a two-dimensional or three-dimensional grid and cannot be represented by a collective operation in the traditional sense of 0 to n-1 ranks, linearly defined), MPI process topologies allows for defining the topological process of the communication and naming the processes. This also enable a means of mapping the processes to the hardware. The virtual topology being defined for a neighborhood communication pattern provides optimization opportunities that can utilize this knowledge of the virtual topology to improve the communication and better map the processes to the physical hardware.

Top supercomputers [2] are equipped with GPUs and high-speed interconnects that allow for optimal communication paths yielding minimal latency and high throughput amongst these GPUs. The adoption of many HPC and Deep Learning workloads to operate on GPU-based systems motivates a need for enhanced communication operations to address the different requirements of these applications. In particular, sparse communication patterns utilized in these applications are often bound by the performance of traditional collective operations or by point-to-point implementations of a non-traditional communication pattern. In order to address scalability issues with traditional collectives being reliant on all processes in the communicator, neighborhood collectives can optimize the workload to involve specific relevant processes in the topology graph and efficiently distribute the work.

### A. Motivation

While previous work has addressed optimizations for neighborhood collectives, in practice within MPI libraries, many neighborhood collective operations are still reliant on a tradi-

*This research is supported in part by NSF grants #1818253, #1854828, #1931537, #2007991, #2018627, #2112606, and XRAC grant #NCR-130002.

tional point-to-point implementation. In particular, to the best of our knowledge, none of the state-of-the-art MPI libraries have support for GPU-aware MPI Neighborhood collectives. With an advent of communication requirements for deep learning and high performance computing application workloads in the exascale era where communication performance is critical, particularly at scale, it is important to have support for neighborhood communication using GPUs. In this work, we propose GPU-aware MPI Neighborhood collective communication, we develop optimized designs for neighborhood collectives that take into account the communication topology, and delve into the details of our proposed work using various structures and routines in the PETSc (PETSc, the Portable, Extensible Toolkit for Scientific Computation) [3] toolkit.

In order to motivate the need for enhanced Neighborhood Collective communication, we take a look at the PETSc toolkit [4] as a use case. We evaluate nonblocking neighborhood alltoallv (`MPI_Ineighbor_alltoallv`) communication using the PETSc library and proposed optimized designs to enhance the routines used in PETSc with neighborhood communcation. PETSc offers an option to set up a star forest object using one of three communication options: one-sided, point-to-point, and neighborhood. In the neighborhood version, in order to overlap communication and computation, PETSc uses the MPI_Ineighbor_Alltoallv operation. Due to lack of support for GPU-aware MPI neighborhood communication in GPU-aware MPI libraries, when this neighborhood call is made, PETSc stages the data from device to host and makes the call from the CPU before passing the data back to the GPU after completing the communication pattern. This practice of copying data from the GPU to the CPU before processing the communication, then copying back to the GPU is commonly known as and referred to in this work as *staging*. We utilize this staging approach, and the point-to-point approach in PETSc as the baselines to compare optimized approaches against.

PETSc supports two types of vectors including: sequential and parallel [5]. We look at the following three routines in PETSc, listed in order of complexity of communication patterns: 1.) scatter from a parallel vector to a parallel vector, 2.) scatter from a sequential vector to a parallel vector, and 3.) scatter from a parallel vector to a sequential vector. The vectors represented in PETSc (sequential and parallel) can be compared to the difference between local and distributed vectors. A parallel vector is a vector that is partitioned into chunks across processors. A scatter in PETSc refers to a collective operation that is used to process a parallel vector. In order to scatter from a sequential vector to a parallel vector, every process in the communication contributes to the parallel vector from its sequential vector. In the reverse case, where the operation is scattering from a parallel vector to a sequential vector, every process receives values from different locations into its own sequential vector.

**Limitations of Current State-of-the-art Approaches:** State-of-the-art GPU-aware MPI and communication libraries including NCCL for NVIDIA GPUs [6], RCCL for AMD GPUs [7], OpenMPI [8], SpectrumMPI [9], CrayMPICH [10], and MVAPICH2 [11] have support for GPU-aware communication amongst processes. However, these GPU-aware Communication and MPI libraries do not have support for GPU-aware Neighborhood communication. In many of these communication libraries, neighborhood communication is merely supported through a point-to-point implementation using sends and receives between the neighborhoods in an iterative fashion. While NCCL has support for neighborhood exchange, this is through NCCL point-to-point operations and relies on the user to implement the communication pattern. This is also the case for any GPU-aware MPI Library. The neighborhood exchange can be defined through GPU-aware point-to-point operations at the application layer. With the rapid changes in the ecosystems and run-times, many applications fall behind in providing the capabilities of utilizing technologies and advancements as they become available. While neighborhood collectives were presented in the MPI 3.0 standard [12], they are yet to become as widely used in applications as they can be, considering the many cases where sparse communication patterns are utilized but not optimized. This responsibility should fall to the MPI developer to create the optimal MPI run-time for neighborhood communication, with consideration for correctness, efficiency, and network topologies. In this work, we emphasize the need for GPU-aware neighborhood collective communication by demonstrating the benefits attained through utilizing a neighborhood collective operation in contrast to merely defining the neighborhood pattern through a point-to-point approach.

### *B. Key Insights and Contributions*

MPI and communication libraries do not have support for GPU-aware neighborhood communication. In this work, we propose GPU-aware and optimized designs for neighborhood communication. The key contributions of this work are:

1) GPU-aware `MPI_Ineighbor_Alltoallv` and extension of designs to support all Neighborhood Alltoall Communication patterns available including: MPI_{Neighbor, Ineighbor}_{alltoall, alltoallv, and alltoallw}
2) Optimized designs for MPI_Ineighbor_Alltoallv utilizing optimal scheduling for communication schemes with high node-incast neighborhood communication
3) Evaluation of three communication routines from PETSc using MPI_Ineighbor_Alltoallv on 64 NVIDIA GPUs on a system with Infiniband Network Interconnection
4) Evaluation of three communication routines from PETSc using MPI_Ineighbor_Alltoallv on 64 AMD GPUs on a system with Slingshot Network Interconnection
5) Delve into the challenges and limitations of state of the art approaches for neighborhood communication at the MPI layer and it's usage at the application layer. This work includes suggestions that need to be addressed by application developers in order to include support for communication that is up-to-date with the latest features provided by the MPI standard and to have the

underlying functionality that would enable portability of applications to GPUs.

**To the best of our knowledge, no other work has been done to propose GPU-aware MPI Neighborhood communication support and optimization.**

### C. Challenges

In this work, we address the need for applications to be up to date with the MPI standard through looking at the case of neighborhood collectives. While the MPI standard is continuously changing, and the trends in modern supercomputers continue to lead innovations and research in different directions, we find that at the application layer, the rate at which changes are made to support these newer innovations both in architecture and in the MPI standard typically falls behind. This leads to an inefficient use of systems where optimizations can be made at the application layer to utilize many of these technologies and advancements. This is particularly evident in the MPI neighborhood collective communication case.

While MPI neighborhood collective communication existed as a concept and in the standard for years, we find that many communication patterns in applications that utilize sparse communication or unbalanced communication loads can benefit from utilizing this communication. However, the gap in merging the communication layer, architecture layer, and application layer knowledge leaves many of these areas underutilized where they can be exploited to benefit each of the other layers. In this work, we address the importance of connecting these layers through MPI neighborhood communication as an example. The application layer can benefit from appointing a neighborhood communication pattern rather than utilizing several point-to-point calls, which will then in turn better utilize the MPI layer. At the MPI layer, we address the importance of utilizing the underlying interconnects and architectural advancements in order to better exploit the features available for communication optimization. By connecting these three layers, we see enhanced performance for patterns that have been extensively used but not optimized.

In Figure 1, we demonstrate this hierarchy between the application, communication, and accelerator layers. At the accelerator layer, note that each of the different GPU vendor types are independent of each other due to their underlying software requirements. While this is addressed at the MPI layer for communication support on each of these vendor types (i.e. CUDA for NVIDIA GPUs, ROCm for AMD GPUs, SYCL for Intel GPUs), the application layer must also have overall support for these vendors in other aspects of their code besides just the communication layer. This adds an additional component to this equation to ensure that every piece of the puzzle has the appropriate functionality and support to enable usage of newer innovations in each of these layers. For example, the time between introducing support for MPI neighborhood communication calls, and adding support for each of the different GPU vendor types, then looking into how to change the application layer to divert from utilizing a less efficient MPI call to a more specific one. Through this work, we hope to motivate the use of neighborhood collective operations for many neighborhood patterns that are currently implemented through standard point-to-point operations.

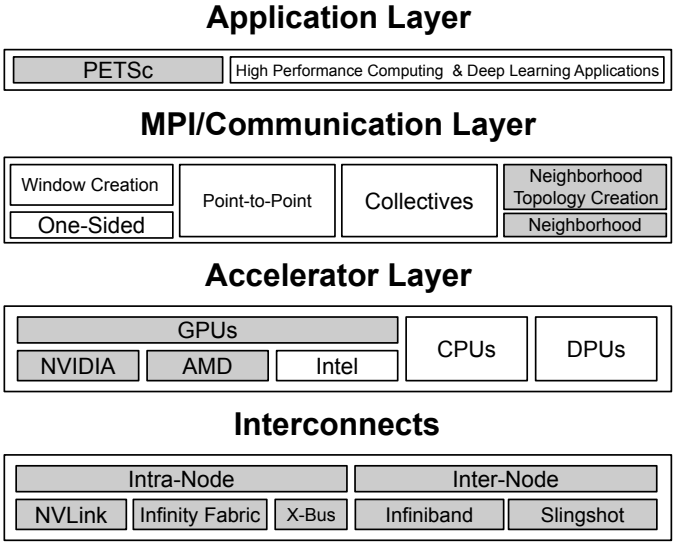


Fig. 1. Connection between the various layers and how they interact with each other once an MPI call is made at the application layer to how the accelerators and interconnects are utilized. *–The highlighted sections emphasize the scope of this work.*

## II. Background

### A. MPI Neighborhood Collectives

Neighborhood Collective Communication was introduced in the MPI 3.0 standard. Neighborhood collectives provide a means of communication over non-fixed or graph-like topologies that cannot be done with standard MPI Collectives. The communication is determined statically in a neighborhood operation. A neighborhood collective call contains information specific to each rank about the incoming processes in the neighborhood, the outgoing data and processes, the varying data counts associated with each send and receive and the general graph topology associated with the communication pattern.

### B. GPU-aware MPI

GPU-aware MPI [13] libraries provide a means to communicate directly between GPU buffers passed to the MPI call without any involvement from the host, eliminating the overhead of copying data back and forth to the CPU in order to make the MPI call from the GPU. Currently, all MPI libraries with GPU support do not provide functionality for Neighborhood based communication directly from GPUs. In fact, even a simple naive approach that encompasses staging from a GPU buffer to the CPU is not supported in current libraries. In order to utilize Neighborhood collective communication from GPUs, this functionality must be added to current MPI libraries and further optimizations can be made to make better use of the architecture, interconnects, and accelerators available. Various transports and protocols related

to GPU-aware communication that have been introduced over the years to optimize performance of GPU-aware MPI at varying messages need to be considered in implementing a GPU-aware approach for different systems and underlying accelerators and interconnects. These include various methods including GPUDirect [14], Inter-Process Communication [15], and GDRCopy [16]. In addition, to the specific protocols that need to be considered, the support for the different protocols over different accelerators such as AMD GPUs [17], and for different interconnects such as Slingshot [18] in an ecosystem heavily optimized for NVIDIA GPUs and Infiniband interconnection is necessary. To develop optimized GPU-aware MPI communication, these underlying approaches and functionalities need to be considered and utilized within the MPI library to trigger the GPU-aware implementation of the operation beyond a simple staging approach.

## III. Design

In this section, we delve into the details of the optimized designs we propose for GPU-aware Nonblocking Neighborhood Alltoallv communication. These optimizations are also extended to other forms of neighborhood alltoall communication including MPI_{Neighbor, Ineighbor}_{alltoall, alltoallv, alltoallw}.

### A. State-of-the-Art Neighborhood Collectives

First, we take a look at the state-of-the-art MPI libraries implementations of neighborhood based collective communication. While sparse/neighborhood communication has been studied in the literature, in practice, we find that MPI libraries still rely on an iterative send and recv approach similar to implementing a communication pattern with point-to-point operations. In Algorithm 1, we detail the current implementation of nonblocking neighborhood collective communication. At the application layer, after the communication graph is created with a call to one MPI_Cart_Create or MPI_Dist_graph_create, a call is made to MPI_Ineighbor_Alltoallv. In Algorithm 1, we see that at each rank, information is extracted related to the `indegree` (number of processes sending to the source rank), and the `outdegree` (number of processes receiving from the source rank). This information is then used to create two loops to first schedule the send operations iteratively in the order of the `src` array and second to schedule the receive operations iteratively in the order that they are passed into the algorithm in the `dsts` array. This kind of a design in practice does not account for contention or take advantage of the extensive information provided by a neighborhood call related to the communication graph and processes involved in each neighborhood, and merely implements the neighborhood communication pattern in the same way a point-to-point implementation of the same communication pattern would operate.

We use this baseline to motivate optimizations for neighborhood communication where processes are scheduling communication such that the network is congested by multiple processes in the communicator sending to one destination process. This is further elaborated on in the examples presented and evaluated in Section IV. The specific examples such as Figure 4 show all processes are sending to the same process in each step, rather than scheduling the communication such that this congestion is unloaded at each step to better utilize the network, and the underlying interconnects providing high speed latency between processes on the same node. We propose optimizations to Neighborhood MPI communication to enhance the lower-level utilization of the hardware and interconnects.

**Algorithm 1:** MPI_Ineighbor_Alltoallv State-of-the-Art

1: $sendbuf \leftarrow$ send buffer starting address
2: $sendcounts \leftarrow$ # of elements to send to every neighbor
3: $sdispls \leftarrow$ displacement i corresponds to data for neighbor i
4: $recvbuf \leftarrow$ receive buffer starting address
5: $rcounts \leftarrow$ # of elements received by every neighbor
6: $rdispls \leftarrow$ displacement j corresponds to location to store data incoming from neighbor j
7: $indegree \leftarrow$ # of processes source is receiving from
8: $outdegree \leftarrow$ # of processes source is sending to
9: $MPI\_Dist\_graph\_neighbors\_count$ (indegree, outdegree, weighted)
10: **for** $i = 0 \rightarrow outdegree$ **do**
11: $sbuf = sendbuf + sdispls[i]$
12: $Schedule\_Send$(sbuf, sendcounts[i], dsts[i])
13: **end for**
14: **for** $j = 0 \rightarrow indegree$ **do**
15: $rbuf = recvbuf + rdispls[j]$
16: $Schedule\_Recv$(rbuf, recvcounts[j], src[j])
17: **end for**

### B. GPU-aware Implementation of Neighborhood Collectives

While GPU-aware MPI libraries have extensive support and optimized algorithms for GPU-aware MPI collective operations, this has not been extended to neighborhood collectives. State-of-the-art MPI libraries do not have support for GPU-aware Neighborhood collective communication. The NCCL library has support for neighbor exchange, but the responsibility is passed to the user to implement this using NCCL send and receive operations. This level of point-to-point implementations of neighborhood communication patterns can be supported by all GPU-aware MPI libraries, but does not account for the knowledge gap between application development and underlying MPI usability. By adding support for GPU-aware Neighborhood collectives, all factors such as deadlock, correctness, optimizations, and efficient performance are handled at the MPI-level and not exposed to the user. They merely utilize the collective operation and MPI graph creation to handle their communication pattern. In this work, we extend GPU-awareness for Neighborhood collectives such that our implementation of neighborhood collectives can have a GPU buffer passed directly into the MPI

**Algorithm 2:** MPI_Ineighbor_Alltoallv Proposed–Nearest Neighbor Design

**Input :**
$sendbuf \leftarrow$ send buffer starting address
$sendcounts \leftarrow$ # of elements to send to every neighbor
$sdispls \leftarrow$ displacement i corresponds to data for neighbor i
$recvbuf \leftarrow$ receive buffer starting address
$rcounts \leftarrow$ # of elements received by every neighbor
$rdispls \leftarrow$ displacement j corresponds to location to store data incoming from neighbor j
**Variables Defined :**
$n\_nhbr \leftarrow$ nearest neighbor
$soffset \leftarrow$ Offset into send Buffer
$indegree \leftarrow$ # of processes source is receiving from
$outdegree \leftarrow$ # of processes source is sending to
**Function :**
// Extract info related to every rank in/out degree
$MPI\_Dist\_graph\_neighbors\_count$ (indegree, outdegree, weighted)
**if** $dsts[outdegree-1] <= rank$ **then**
  **for** $k = (outdegree-1)$ **to** $0$ **do**
    $sbuf = sendbuf + sdispls[k]$
    $Schedule\_Send$(sbuf, sendcounts[k], dsts[k])
  **end for**
**else if** $dsts[0] >= rank$ **then**
  **for** $l = 0$ **to** $outdegree$ **do**
    $sbuf = sendbuf + sdispls[l]$
    $Schedule\_Send$(sbuf, sendcounts[l], dsts[l])
  **end for**
**else**
  // Refer to Algorithm 6
  $Get\_Nearest\_Neighbor$(rank, dsts, outdegree)
  $x \leftarrow 0$
  $y \leftarrow n\_nhbr$
  $z \leftarrow n\_nhbr - 1$
  **while** $x < outdegree$ **do**
    **if** $y < outdegree$ **then**
      $sbuf = sendbuf + sdispls[y]$
      $Schedule\_Send$(sbuf, sendcounts[y], dsts[y])
      $y++$;
      $x++$;
    **end if**
    **if** $z \geq 0$ **then**
      $sbuf = sendbuf + sdispls[z]$
      $Schedule\_Send$(sbuf, sendcounts[z], dsts[z])
      $z--$;
      $x++$;
    **end if**
  **end while**
**end if**
// Schedule receives for every rank
**for** $j = 0$ **to** $indegree$ **do**
  $rbuf = recvbuf + rdispls[j]$
  $Schedule\_Recv$(rbuf, recvcounts[j], src[j])
**end for**

**Algorithm 3:** Get Nearest Neighbor Implementation

**Input :**
$rank \leftarrow$ Rank of source process
$dsts \leftarrow$ stores ranks of outgoing neighbors of source process
$outdegree \leftarrow$ # of outgoing neighbors of the source process
**Output :**
$n\_nhbr \leftarrow NearestNeighbor$
**Function :**
$Get\_Nearest\_Neighbor(rank, dsts, outdegree)$ :
$f \leftarrow outdegree$
**while** $d < f$ **do**
  $soffset \leftarrow (d+f)/2$
  **if** $dsts[soffset] == rank$ **then**
    $n\_nhbr \leftarrow soffset$;
    $break$;
  **else if** $dsts[soffset] > rank$ **then**
    **if** $(soffset > 0$ **and** $rank > dsts[soffset-1])$ **then**
      $n\_nhbr \leftarrow (rank - dsts[soffset-1] \geq dsts[soffset] - rank)\ ?\ soffset : (soffset-1)$;
      $break$;
    **end if**
    $f \leftarrow soffset$
  **else**
    **if** $(soffset < (outdegree-1)$ **and** $rank < dsts[soffset+1])$ **then**
      $n\_nhbr \leftarrow (rank - dsts[soffset] \geq dsts[soffset+1] - rank)\ ?\ (soffset+1) : soffset$;
      $break$;
    **end if**
    $d \leftarrow soffset + 1$
  **end if**
  $n\_nhbr \leftarrow soffset$
**end while**

call allowing for utilization of different GPU-based protocols. Through support for GPU-awareness, underlying protocols such as GDRCopy [16], GPUDirect RDMA technology [14], and Inter-Process Communication (IPC) [15] can be utilized between the GPUs during communication to utilize the hardware efficiently. This is also extended over underlying CUDA and ROCm support to run on both NVIDIA and AMD GPUs.

### *C. Non-Blocking Neighborhood Alltoallv Designs*

The challenges in optimizing neighborhood communication come from the arbitrariness of the communicator size and number of processes involved in each rank's communication scheme. For traditional collectives, creating an optimized algorithm that utilizes topology information to generate a hierarchy or tree-like structure amongst processes will involve

a fixed offset or a fixed tree pattern that can be applied in all contexts of the collective call. For neighborhood communication, the design considerations can be specific to the communication pattern at hand. In particular, many neighborhood communication patterns can have quite an unbalanced communication scheme amongst processes, defined by the degree of processes sending to (`indegree`) and the degree of processing being sent from (`outdegree`) each rank. This information is extracted at the beginning of the neighborhood call and utilized for memory allocation considerations and for scheduling the number of sends and receives specific to each rank (Algorithm 2, Line 15).

In particular, with the example demonstrated in Table VII and Figure 4, each rank loops through the `outdegree` value and schedules send operations to each process in its neighborhood and then loops through the `indegree` value to schedule all the receives. As noted in these demonstrations of the communication pattern, an iterative approach like this with no consideration to the topology or graph pattern can lead to very inefficient scheduling. In each step, multiple processes are sending to the same rank, leaving many other communication paths idle. This is particularly relevant in a GPU-aware case where high-speed interconnects between GPUs in a GPU-dense node configuration, such as NVLink, provide high speed and throughput that can be utilized here. If the scheduling is optimized such that the links are being utilized in scenarios where they would otherwise remain idle, the contention going to one process in each step can be reduced significantly. This contention is even more critical in such a communication pattern at a larger scale where many more processes are sending to the same process at the same time.

We propose a nearest-neighbor design for MPI nonblocking neighborhood Alltoallv that utilizes information extracted from `MPI_Graph_neighbors_count` or `MPI_Dist_graph_neighbors_count` depending on whether it is a standard graph topology or distributed graph topology, respectively. In this work, we look at examples that utilize a distributed graph topology and use `MPI_Dist_graph_neighbors_count` to extract the `indegree` and `outdegree` associated with each rank.

In Algorithm 2, we consider GPU-dense systems where intra-node communication involves high-speed interconnects and higher connectivity enabling more efficient communication to first schedule the nearest neighbor and reorder the pattern with which processes send data to each other. If the source process is the highest rank amongst the destinations that the source process will be sending to in the ordered array of destinations (`dsts[]`), then the process will send to the ranks in its communicator starting from the process closest to it until the lowest rank. This is applied in the opposite direction, assuming that the current rank is the lowest rank in the communicator, then the sends are initiated in a standard iterative approach to schedule. In all other cases, where the current rank is neither the highest nor the lowest rank in the communicator, we make a call to the `Get_Nearest_Neighbor` (Algorithm 2 line 28) function to get the nearest neighbor (`n_nhbr`) in the `dsts[]` array amongst the outgoing neighbors of the process. This is implemented in Algorithm 3.

The nearest neighbor implementation (Algorithm 3) takes as input the rank of the source process (`rank`), the information stored regarding the ranks of the outgoing neighbors of the source process (`dsts`), and the number of outgoing neighbors from the source (`outdegree`) to find the rank of the nearest neighbor to the source rank in the neighborhood. For a traditional collective operation, this could be extracted easily by assuming that ranks in the +/- 1 range of a process are the nearest process. However, with neighborhood collectives, this becomes a lot more complex considering not all processes are in the neighborhood of each other (i.e. they may not be outgoing/incoming neighbors of each other). Since we have the number of `outdegree` and an array storing the ranks of each neighbor, we can do a binary search to find the nearest neighbor (`n_nhbr`). Once this value is determined, in Algorithm 2, we then schedule a send operation to the nearest rank greater than and the nearest rank less than the current process until all processes are met. We also handle the case where this is not an even pattern between processes. This allows for an example demonstrated in Figure 4, where `MPI_Ineighbor_alltoallv` is used to scatter from a parallel vector to a sequential vector to be rescheduled such that the load is more balanced as demonstrated in Figure 5. This approach can also be applied to blocking MPI neighborhood alltoall, alltoallv, and alltoallw and to nonblocking neighborhood alltoall and alltoallw. The different schemes require separate consideration for the datatype being passed depending on whether you are utilizing alltoallw, and the amount of data being passed for alltoallv. A nonblocking algorithm can easily be applied as a blocking algorithm, with a MPI_Waitall operation at the end of the algorithm definition.

## IV. Evaluation

In this section, we evaluate our proposed optimizations and GPU-aware Neighborhood nonblocking Alltoallv communication using the PETSc library. We utilize PETSc, the Portable, Extensible Toolkit for Scientific Computation to demonstrate performance benefits of our designs compared to existing methodologies used in the library.

### *A. Experimental Setup*

In order to demonstrate support on multiple different types of GPUs (NVIDIA and AMD), and various network interconnects (Infiniband and Slingshot), we expand our evaluation to cover a larger scope of system configurations. We utilize the Lassen system, [19], at Lawrence Livermore National Laboratory (LLNL) and the Spock system, [20], at the Oakridge Leadership Computing Facility (OLCF). The system details associated with Lassen are detailed in Table I where we run our experiments on up to 64 GPUs (16 nodes, 4 GPUs per node) with NVIDIA V100 GPUs, NVLink connecting the GPUs within the node, and Infiniband networking across nodes.

The system details associated with Spock are detailed in Table II, where we run our experiments on up to 64

TABLE I
DETAILS OF SYSTEM USED FOR EVALUATION

| System Details - **Lassen at LLNL** | |
|---|---|
| Architecture: | ppc64le |
| Model name: | POWER9 |
| Numa Nodes: | 6 |
| Core(s) per socket: | 22 |
| Socket(s): | 2 |
| Network Interconnect: | Infiniband EDR (25 GB/s) |
| GPU Vendor & Type | NVIDIA V100 |
| GPUs Per Node | 4 |
| Intra-Node GPU Interconnect: | 2-Lane NVLink (50 GB/s) |
| Intra-Node Cross Socket Interconnect: | X-Bus (64 GB/s) |
| CUDA Version | 11.0.2 |

GPUs (16 nodes, 4 GPUs per node) with AMD MI100 GPUs, Infinity fabric connecting GPUs within the nodes, and Slingshot networking across nodes. With the growing system requirements for the top supercomputers, particularly with the #1 Supercomputer, Frontier [21] on the Top500 [2] list being equipped with Slingshot networking and AMD GPUs, it is pertinent to show optimizations and scalability in this ecosystem in addition to the more widely accessible NVIDIA GPUs and infiniband networking on systems in recent years.

TABLE II
DETAILS OF SYSTEM USED FOR EVALUATION

| System Details - **Spock at OLCF** | |
|---|---|
| Architecture: | x86_64 |
| Model name: | AMD EPYC 7662 64-Core Processor |
| Numa Nodes: | 4 |
| Core(s) per socket: | 64 |
| Socket(s): | 1 |
| Network Interconnect: | Slingshot-10 (12.5 + 12.5 GB/s) |
| GPU Vendor & Type | AMD MI100 |
| GPUs Per Node | 4 |
| Intra-Node GPU Interconnect: | Infinity Fabric (46 + 46 GB/s) |
| ROCm Version | 5.0.2 |

### *B. PETSc*

The PETSc toolkit is widely used in HPC applications for algebraic solving including applications such as AWP-ODC, MILC, and OpenFOAM. It supports stencil and ghost-cell communication patterns and includes various scalable solutions defined by structures for applications modeled by partial differential equations [3]. The toolkit has been optimized to have support for GPUs [22] and to run with CUDA, HIP, SYCL, OpenCL, and RAJA. It is an open-source implementation widely used and continuously enhanced to support changing hardware and architecture. With the exascale era demanding more GPU-based performance optimizations, there is a need for scalable solutions on GPUs within PETSc that apply to both NVIDIA GPUs and optimized on AMD GPUs and the Slingshot interconnect for the current exascale supercomputer, Frontier, and the upcoming, Aurora system [23] at the Argonne Leadership Computing Facility (ALCF).

Through the PETSc test suite, when testing different data structures and routines, we can pass one of either: basic, window, or neighbor to sf_type defining whether to utilize one-sided, point-to-point or a neighborhood communication mechanism used to create the star forest (SF) object (PetscSF), [24]. PETSc uses star forests to define communication patterns efficiently and to manage the communication of arrays and vectors within an MPI communication realm. It has support for running with `-sf_type neighbor` to initiate a non-blocking neighborhood alltoallv MPI call. Within PETSc, the application will stage the buffer to the CPU (copy from device to host), then call the MPI operation, and then stage the buffer back to the GPU (copy from host to device) when the MPI library does not have support for passing a GPU buffer directly to the MPI call. With our GPU-aware MPI implementation, we are able to run the neighborhood option on GPUs without requiring staging at the application level or at the MPI level, eliminating this added copy overhead, which is evident in the performance comparisons presented in this section.

In our evaluation we compare the neighborhood approach against the basic, point-to-point approach to demonstrate optimized performance when utilizing a specific MPI neighborhood call to define the forest. We noticed several run-time errors utilizing the window option with GPU-aware PETSc and therefore did not include this in the evaluation. We also evaluate the neighborhood sf_type using different `vec_types` including `--vec_type cuda` to run with CUDA-aware MPI on NVIDIA GPUs and `--vec_type hip` to run with ROCm-aware MPI on AMD GPUs.

We demonstrate three routines in PETSc that utilize MPI_Ineighbor_alltoallv below: 1. scattering from a parallel vector to a parallel vector (Figure 2), 2. scattering from a sequential vector to a parallel vector (Figure 3), and 3. scattering from a parallel vector to a sequential vector (Figure 4). These examples are presented in order of complexity of the communication pattern. We evaluate the time spent in the following three events for each of these routines: VecScatterBegin, VecScatterEnd, and SFSetUP. These three routines are impacted by the performance of a MPI_Ineighbor_alltoallv call to do the scatter operation on the vector (VecScatterBegin and VecScatterEnd) and to set up the star forest based on the MPI communication happening.

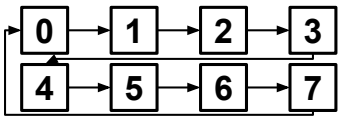


Fig. 2. PETSc —**Scatter from a parallel vector to a parallel vector**— Routine (NOTE: Example shows the communication pattern on 8 Processes)

In the first routine in PETSc that we evaluate, involving scattering data from a parallel vector to a parallel vector using neighborhood communication, Table IV, each of the processes send data to the next rank in a ring fashion. In this table, "Out Degree" refers to the number of processes the source rank is sending to, and the "In Degree" refers to the number of processes the source rank is receiving from. In this communication pattern, Figure 2, the load is balanced across

TABLE III
PETSc —**Scatter from a parallel vector to a parallel vector**— Routine
Performance Evaluation on 64 GPUs (16 nodes and 4 GPUs per Node) on the Lassen System

| Event | Time (seconds) | | | % Improvement | |
|---|---|---|---|---|---|
| | -sf_type basic -vec_type cuda | -sf_type neighbor -vec_type cuda | -sf_type neighbor -vec_type cuda w/ GPU-aware MPI Designs | Compared to -sf_type basic -vec_type cuda | Compared to -sf_type neighbor -vec_type cuda |
| **VecScatterBegin:** | 8.23E-03 | 9.79E-03 | 6.70E-03 | 18.55% | 31.58% |
| **VecScatterEnd:** | 1.85E-02 | 2.13E-02 | 1.84E-02 | 0.37% | 13.65% |
| **SFSetUp:** | 9.56E-04 | 2.49E-03 | 4.29E-04 | 55.10% | 82.74% |
| **SUM of Events:** | 2.76E-02 | 3.36E-02 | 2.55E-02 | **7.67%** | **24.00%** |

processes where each process is sending to and receiving from one other process.

TABLE IV
PETSc —**Scatter from a parallel vector to a parallel vector**— Routine (Communication Pattern)

| Source Rank | Sends To | Out Degree | In Degree |
|---|---|---|---|
| **0** | 1 | 1 | 1 |
| **1** | 2 | 1 | 1 |
| **2** | 3 | 1 | 1 |
| **3** | 4 | 1 | 1 |
| **4** | 5 | 1 | 1 |
| **5** | 6 | 1 | 1 |
| **6** | 7 | 1 | 1 |
| **7** | 0 | 1 | 1 |

For simplicity, we demonstrate the communication load when utilizing 8 processes. In our evaluation, we extend this use case across multiple processes. This is an example of a simple neighborhood call where the load is balanced between processes. At scale, this sequence is also balanced similar to a ring fashion where every rank communicates with the next rank. We evaluate this routine on 64 GPUs on the Lassen system in Table III with PETSc basic sf_type and cuda vectors, and the current implementation in PETSc of cuda vectors with CPU-staging neighbor sf_type, and PETSc neighbor and cuda vectors using our proposed GPU-aware nonblocking neighborhood communication. We see approximately 7.67% improvement against the point-to-point implementation. For this simple example, the benefits seen at the GPU-level are attained from having GPU-aware underlying MPI support. The optimized designs would not have significant impact here with scheduling due to the balanced and simple workload. When comparing with a CPU-staging scheme, the added overhead of copying back and forth to the GPU with every collective call can be significant and evident here where we show approximately 24% enhancements against the CPU-staging schemes. We present this example to demonstrate the benefit of a GPU-aware implementation of neighborhood communication.

In Table VI, we demonstrate the scenario in PETSc where data is scattered from a sequential vector to a parallel vector. In this routine, many processes communicate data to a specific smaller set of processes. This can be seen in Figure 3, in an 8 process case, where processes 4-7 do not communicate with each other but would communicate over the network (in a 4PPN case). Scheduling the processes here does not yield optimal setup in SFSetUp where we see that the performance is not as efficient as the alternatives, but does improve performance when utilizing GPU-aware MPI protocols. This is because the processes are all communicating to a smaller set of processes. We see quite improved performance for VecScatterBegin and VecScatterEnd compared to the CPU-staging technique, and the basic approach, respectively when utilizing GPU-aware MPI. Through the neighborhood exchange, we are able to show 8.55% improvement against the point-to-point approach, and approximately 14.11% against the CPU-staging approach.

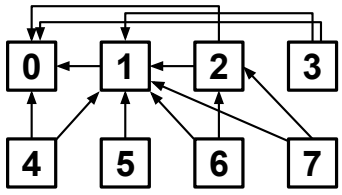


Fig. 3. PETSc —**Scatter from a sequential vector to a parallel vector**— Routine (NOTE: Example shows the communication pattern on 8 Processes)

In the final example, routine 11, where the structure scatters from a parallel vector to a sequential vector, we see the case where this design yields enhanced performance for a very network-congested communication pattern. In Table VII, we see that in each step all the processes are typically sending to the same process. This can lead to a lot of congestion over the network and leaves several communication paths and links idle when they can be utilized to progress other communication between processes. This is further depicted step-by-step in Figure 4. Through the optimized schemes, where the scheduling is enhanced to provide better performance, as depicted in Figure 5, we are able to show performance gain at 64 GPUs on NVIDIA and AMD GPUs. We evaluate it on NVIDIA GPUs on the Lassen system in Table VIII, where we show 8.25% improvement over the basic approach, and 30.90% improvement over the CPU-staging approach. We also evaluate this on AMD GPUs on the Spock system in Table IX, demonstrating 33.25% and 29.52% improvement against the

TABLE V
PETSc —**Scatter from a sequential vector to a parallel vector**— Routine
Performance Evaluation on 64 GPUs (16 nodes and 4 GPUs per Node) on the Lassen System

| **Event** | Time (seconds) | | | % Improvement | |
|---|---|---|---|---|---|
| | -sf_type basic -vec_type cuda | -sf_type neighbor -vec_type cuda | -sf_type neighbor -vec_type cuda w/ GPU-aware MPI Designs | Compared to -sf_type basic -vec_type cuda | Compared to -sf_type neighbor -vec_type cuda |
| **VecScatterBegin:** | 4.61E-03 | 6.20E-03 | 4.30E-03 | 6.60% | 30.65% |
| **VecScatterEnd:** | 1.09E-03 | 5.73E-04 | 6.04E-04 | 44.75% | -5.40% |
| **SFSetUp:** | 2.52E-03 | 1.98E-03 | 2.61E-03 | -3.60% | -32.17% |
| **SUM of Events:** | 8.22E-03 | 8.75E-03 | 7.52E-03 | **8.55%** | **14.11%** |

TABLE VI
PETSc —**Scatter from a sequential vector to a parallel vector**— Routine (Communication Pattern)

| **Source Rank** | Sends To | | Out Degree | In Degree |
|---|---|---|---|---|
| **0** | - | - | 0 | 4 |
| **1** | 0 | - | 1 | 6 |
| **2** | 0 | 1 | 2 | 2 |
| **3** | 0 | 1 | 2 | 0 |
| **4** | 0 | 1 | 2 | 0 |
| **5** | 1 | - | 1 | 0 |
| **6** | 1 | 2 | 2 | 0 |
| **7** | 1 | 2 | 2 | 0 |

basic and CPU-staging schemes, respectively. The negative improvement, emphasized with a (*) in the table are a result of the MPI_Dist_graph_create operation used for Neighborhood calls to generate the graph topology. It makes sense that at this scale with this heavy load of a communication pattern, there will be some overhead to generate the graph topology. The smaller value for the basic approach is a result of this being a point-to-point scheme that does not utilize a graph topology, and therefore does not call this API.

TABLE VII
PETSc —**Scatter from a parallel vector to a sequential vector**— Routine

| **Source Rank** | Sends To | | | | | | | Out Degree | In Degree |
|---|---|---|---|---|---|---|---|---|---|
| **0** | - | - | - | - | - | - | - | 0 | 7 |
| **1** | 0 | 2 | - | - | - | - | - | 2 | 6 |
| **2** | 0 | 1 | 3 | 4 | 5 | - | - | 5 | 6 |
| **3** | 0 | 1 | 2 | 4 | 5 | 6 | 7 | 7 | 5 |
| **4** | 0 | 1 | 2 | 3 | 5 | 6 | 7 | 7 | 5 |
| **5** | 0 | 1 | 2 | 3 | 4 | 6 | 7 | 7 | 4 |
| **6** | 0 | 1 | 2 | 3 | 4 | 5 | 7 | 7 | 4 |
| **7** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 4 |

## V. Related Work

In [25], Hoefler et. al addressed the need for representation of stencil computations being addressed in MPI beyond using standard collective calls. They proposed various principles

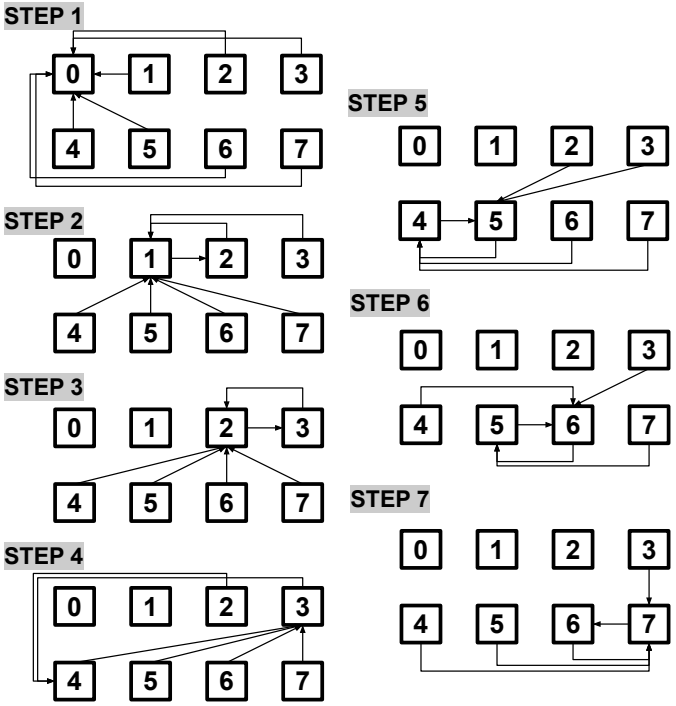


Fig. 4. PETSc —**Scatter from a parallel vector to a sequential vector**— Routine. (NOTE: Example shows the communication pattern on 8 Processes. For simplicity we demonstrate an 8 GPUs per node (GPN) case in the diagram here but to demonstrate scalability, we do our evaluation with a higher GPN case where the communication pattern is even denser and more congestion is incurred by many more processes sending simultaneously to one process). Note the congestion in each of these steps when all/many processes are sending to the same rank. Through the optimized designs, this congestion is handled through determining the nearest-neighbor and utilizing more efficient scheduling to send the data across the network.

for optimizing communication for this kind of pattern by utilizing neighborhood communication as opposed to utilizing point-to-point operations. Neighborhood collective operations were then introduced in the MPI 3.0 standard. Hoefler et. al [26] proposed 3 different sparse nearest neighbor collective operations, called sparse gather, sparse all-to-all, and sparse reduction operations. Träff et. al [27] also proposed 3 algorithms for neighborhood collective operations for parallel stencil-like computations and discussed a mechanism that has fast, local computation of communication schedules. This work was extended in [28] when they proposed Cartesian Collective

TABLE VIII
PETSc —SCATTER FROM A PARALLEL VECTOR TO A SEQUENTIAL VECTOR— ROUTINE
PERFORMANCE EVALUATION ON 64 GPUS (16 NODES AND 4 GPUS PER NODE) ON THE LASSEN SYSTEM WITH NVIDIA GPUS AND INFINIBAND INTERCONNECT):

| Event | Time (seconds) | | | % Improvement | |
|---|---|---|---|---|---|
| | -sf_type basic -vec_type cuda | -sf_type neighbor -vec_type cuda | -sf_type neighbor -vec_type cuda w/ GPU-aware MPI Designs | Compared to -sf_type basic -vec_type cuda | Compared to -sf_type neighbor -vec_type cuda |
| **VecScatterBegin:** | 9.85E-02 | 1.73E-01 | 1.14E-01 | -16.17% * | 33.86% |
| **VecScatterEnd:** | 6.06E-02 | 3.96E-02 | 3.65E-02 | 39.85% | 7.91% |
| **SFSetUp:** | 9.16E-02 | 1.20E-01 | 7.92E-02 | 13.57% | 34.21% |
| **SUM of Events:** | 2.51E-01 | 3.33E-01 | 2.30E-01 | **8.25%** | **30.90%** |

TABLE IX
PETSc —SCATTER FROM A PARALLEL VECTOR TO A SEQUENTIAL VECTOR— ROUTINE
PERFORMANCE EVALUATION ON 64 GPUS (16 NODES AND 4 GPUS PER NODE) ON THE SPOCK SYSTEM WITH AMD GPUS AND SLINGSHOT INTERCONNECT):

| Event | Time (seconds) | | | % Improvement | |
|---|---|---|---|---|---|
| | -sf_type basic -vec_type hip | -sf_type neighbor -vec_type hip | -sf_type neighbor -vec_type hip w/ GPU-aware MPI Designs | Compared to -sf_type basic -vec_type hip | Compared to -sf_type neighbor -vec_type hip |
| **VecScatterBegin:** | 2.58E-02 | 3.87E-02 | 2.91E-02 | -12.51% * | 24.83% |
| **VecScatterEnd:** | 1.99E-02 | 1.28E-02 | 5.31E-04 | 97.33% | 95.84% |
| **SFSetUp:** | 4.62E-03 | 4.17E-03 | 4.03E-03 | 12.86% | 3.46% |
| **SUM of Events:** | 5.04E-02 | 5.56E-02 | 3.36E-02 | **33.25%** | **39.52%** |

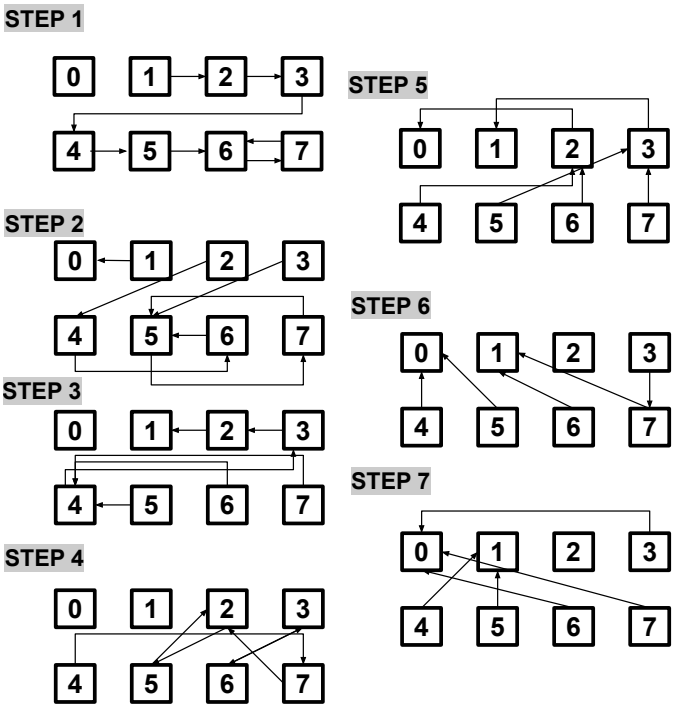


Fig. 5. PETSc Routine 11 Optimal Scheduling: The communication pattern used in PETSc with Nonblocking Neighborhood Alltoallv to scatter from a parallel vector to a sequential vector using 8 GPUs across two nodes with optimized Nonblocking Alltoallv to enhance the scheduling of the processes. This example was run on 64 GPUs in our evaluation where we will be unloading a lot of contention rescheduling 63 GPUs from sending to Process 0 initially leaving all other communication paths between GPUs idle.

Communication for isomorphic stencil patterns. They also proposed message-combining algorithms and implemented that on top of MPI for alltoall and allgather type Cartesian collective communications.

Mirsadeghi et. al addressed optimizations by exploiting common neighborhood patterns in [29], and proposed designs to build optimized message-combining communication schedules. Ghazimirsaeed et. al [30] proposed a distributed algorithm for neighborhood communication by modeling the part of problems as a weighted modeling problem in hypergraphs. They also considered the topology of the neighborhood and proposed topology-agnostic and topology-aware designs. This work was extended in [31], where Ghazimirsaeed et. al improved neighborhood communication for large messages by importing the virtual communication pattern and the physical topology of the cluster. They also considered the load balance issue and proposed a mathematical model to calculate the capacity and dispatch the communication flow.

## VI. CONCLUSION

In this paper, we proposed GPU-aware support to MPI Neighborhood alltoall operations. In particular, we proposed optimizations to MPI_Ineighbor_Alltoallv for high node-incast neighborhood communication. We identified the overlap potential in the communication and proposed enhanced designs to take advantage of optimal paths between GPUs for Neighborhood collective operations to utilize the underlying communication paths and links. We evaluated the performance of nonblocking MPI Neighborhood alltoallv on NVIDIA and

AMD GPUs on systems utilizing different interconnects including Slingshot and Infiniband.

We utilized the PETSc toolkit to examine three routines reliant on Nonblocking MPI neighborhood alltoallv implementations: scattering from a parallel vector to a parallel vector, scattering from a sequential vector to a parallel vector, and scattering from a parallel vector to a sequential vector using a star forest graph representation. We demonstrated over 30% improvement when utilizing our schemes compared to the existing approaches in PETSc. In the future, this work can be extended to add support for GPU-aware neighborhood collective communication to other applications that utilize sparse communication patterns implemented through point-to-point operations.

## VII. Acknowledgements

We would like to thank Dr. Sameer Shende (University of Oregon) and Dr. Olga Pearce (Lawrence Livermore National Laboratory) for providing access to the HPC systems used in the paper. We would also like to thank Dr. Richard Tran Mills and Dr. Junchao Zhang from the PETSc team at Argonne National Laboratory for their guidance in running the application used in this work.

## References

[1] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 4.0," https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf, 2021.

[2] E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, "TOP 500 Supercomputer Sites," http://www.top500.org, 1993.

[3] S. Balay, S. Abhyankar, M. F. Adams, S. Benson, J. Brown, P. Brune, K. Buschelman, E. M. Constantinescu, L. Dalcin, A. Dener, V. Eijkhout, W. D. Gropp, V. Hapla, T. Isaac, P. Jolivet, D. Karpeev, D. Kaushik, M. G. Knepley, F. Kong, S. Kruger, D. A. May, L. C. McInnes, R. T. Mills, L. Mitchell, T. Munson, J. E. Roman, K. Rupp, P. Sanan, J. Sarich, B. F. Smith, S. Zampini, H. Zhang, H. Zhang, and J. Zhang, "PETSc Web page," https://petsc.org/, 2022. [Online]. Available: https://petsc.org/

[4] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith, "Efficient management of parallelism in object oriented numerical software libraries," in *Modern Software Tools in Scientific Computing*, E. Arge, A. M. Bruaset, and H. P. Langtangen, Eds. Birkhäuser Press, 1997, pp. 163–202.

[5] "Petsc dev documentation," 2020. [Online]. Available: https://wg-beginners.readthedocs.io/en/latest/index.html

[6] NVIDIA, "NCCL," https://github.com/NVIDIA/nccl, 2016.

[7] AMD, "ROCm Communication Collectives Library," https://github.com/ROCmSoftwarePlatform/rccl, 2016, Accessed: September 24, 2023.

[8] Open MPI, "Open MPI: Open Source High Performance Computing," https://www.open-mpi.org/, 2004, Accessed: September 24, 2023.

[9] IBM, "IBM Spectrum MPI: Accelerating high-performance application parallelization," https://www.ibm.com/us-en/marketplace/spectrum-mpi, 2018, Accessed: September 24, 2023.

[10] NERSC, "Cray MPICH," https://docs.nersc.gov/development/programming-models/mpi/cray-mpich/.

[11] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, "The mvapich project: Transforming research into high-performance mpi library for hpc community," *Journal of Computational Science*, p. 101208, 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877750320305093

[12] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.0," https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, 2012.

[13] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, Oct 2014.

[14] NVIDIA, "NVIDIA GPUDirect," https://developer.nvidia.com/gpudirect.

[15] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI Communication on Multi-GPU Systems Using CUDA Inter-Process Communication," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, May 2012, pp. 1848–1857.

[16] Davide Rossetti, "NA fast GPU memory copy library based on NVIDIA GPUDirect RDMA technology," https://github.com/NVIDIA/gdrcopy, Accessed: September 24, 2023.

[17] K. Shafie Khorassani, J. Hashmi, C.-H. Chu, C.-C. Chen, H. Subramoni, and D. K. Panda, "Designing a rocm-aware mpi library for amd gpus: early experiences," in *High Performance Computing: 36th International Conference, ISC High Performance 2021, Virtual Event, June 24–July 2, 2021, Proceedings*. Springer, 2021, pp. 118–136.

[18] K. Shafie Khorassani, C. C. Chen, B. Ramesh, A. Shafi, H. Subramoni, and D. Panda, "High performance mpi over the slingshot interconnect: Early experiences," in *Practice and Experience in Advanced Research Computing*, 2022, pp. 1–7.

[19] Lawrence Livermore National Laboratory, "Lassen," https://hpc.llnl.gov/hardware/platforms/lassen, 2020, Accessed: September 24, 2023.

[20] Oakridge Leadership Computing Facility (OLCF), "Spock Quick-Start Guide," https://docs.olcf.ornl.gov/systems/spock_quick_start_guide.html, 2022, Accessed: September 24, 2023.

[21] ORNL, "Frontier," https://www.olcf.ornl.gov/frontier/.

[22] R. T. Mills, M. F. Adams, S. Balay, J. Brown, A. Dener, M. Knepley, S. E. Kruger, H. Morgan, T. Munson, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and J. Zhang, "Toward performance-portable petsc for gpu-based exascale systems," *Parallel Computing*, vol. 108, p. 102831, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781912100079X

[23] Intel, "Aurora supercomputer," https://www.intel.com/content/www/us/en/high-performance-computing/supercomputing/exascale-computing.html.

[24] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, and S. Zampini, "The petscsf scalable communication layer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, 2022.

[25] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.

[26] T. Hoefler and J. L. Traff, "Sparse collective operations for mpi," in *2009 IEEE International Symposium on Parallel Distributed Processing*, 2009, pp. 1–8.

[27] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold, "Isomorphic, sparse mpi-like collective communication operations for parallel stencil computations," ser. EuroMPI '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2802658.2802663

[28] J. L. Träff and S. Hunold, "Cartesian collective communication," ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337848

[29] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize mpi neighborhood collectives," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 348–357.

[30] S. M. Ghazimirsaeed, S. H. Mirsadeghi, and A. Afsahi, "An efficient collaborative communication mechanism for mpi neighborhood collectives," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 781–792.

[31] S. Ghazimirsaeed, Q. Zhou, A. Ruhela, M. Bayatpour, H. Subramoni, and D. Panda, "A hierarchical and load-aware design for large message neighborhood collectives," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2020, pp. 1–13. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SC41405.2020.00038