ROCm-Aware Leader-based Designs for MPI Neighbourhood Collectives

[*]Yıltan Hassan Temuçin, [+]Mahdieh Gazimirsaeed, [*]Ryan E. Grant, [*]Ahmad Afsahi
[*]*ECE Department, Queen's University*, Kingston, ON, Canada
[+]*DCGPU and Accelerated Processing Advanced Micro Devices Inc* Austin, TX, USA
[*]{yiltan.temucin | ryan.grant | ahmad.afsahi}@queensu.ca
[+]mahdieh.ghazimirsaeed@amd.com

***Abstract*—MPI neighborhood collectives were introduced in the MPI-3.0 standard to support sparse communication patterns used by many applications. Simultaneously, GPU-Aware MPI communication has become a prominent part of modern systems. With the rise of AMD GPUs and their incorporation into upcoming exascale systems like Frontier, it has become essential to optimize communication libraries for AMD platforms. In this paper, we take advantage of the hardware and networking features of AMD GPUs to design efficient and scalable neighborhood collective operations: allgather and allgatherv. We evaluate the performance of the proposed design for Random Sparse Graph and Moore neighborhood micro-benchmarks as well as an SpMM kernel. The results show that we obtain up to 7.03x speedup for the Random Sparse Graph micro-benchmark, up to 3.82x for the Moore neighborhood micro-benchmark, and up to 2.29x speedup for the SpMM kernel.**

***Index Terms*—MPI, Neighborhood Collectives, AMD, ROCm, Topology**

I. Introduction

Heterogeneous computing has been essential to deploying Exascale systems. Graphics Processing Units (GPUs) have played a prominent role in accelerating scientific [1] and Artificial Intelligence (AI) workloads [2]. The Frontier system at Oak Ridge National Laboratory (ORNL), which is equipped with AMD EPYC processors and AMD Instinct^{TM} MI250X GPUs [3], is the top-ranked supercomputer in the world as of November 2023 [4]. Moreover, seven of the top ten High-Performance Computing (HPC) systems use GPUs and four of them are AMD-based platforms. The broad deployment of AMD GPUs in top supercomputers emphasizes the importance of optimizing AMD GPU communications. Improvements to Inter-GPU communication bandwidth have been found to correspond to improvements in application performance [5]. The multi-GPU computing nodes in AMD platforms are equipped with the Infinity Fabric^{TM} link. These interconnects provide significantly improved bandwidth between devices compared to traditional PCIe interconnects [5], so it is desirable to utilize the Infinity Fabric^{TM} efficiently to provide improved performance on AMD GPU-based systems.

The Message Passing Interface (MPI) is the de-facto standard for communication in HPC [17]. MPI offers various methods such as point-to-point, partitioned point-to-point, remote memory access (RMA), and collective communication to handle different communication patterns between the processes. With point-to-point communication, the user can implement each pattern with send/receive operations between pairs of processes. Using point-to-point operations exclusively burdens the application programmer with optimizing the communication pattern for specific network topologies, ensuring its correctness, and debugging potential deadlocks. Moreover, MPI collective operations require all processes in a communicator to participate in the communication, imposing inherent scalability issues, and only supporting a set of predetermined communication patterns (e.g., allgather and broadcast). To address these limitations, the MPI standard introduced **Neighborhood Collectives**. With neighborhood collective operations, application developers can define any arbitrary communication pattern. Application developers construct the topology graph of the processes by specifying the outgoing/incoming neighbors of each process. This information is then attached to the MPI communicator and is passed as the input of the neighborhood collective operation.

A recent survey on MPI usage within the U.S Exascale Computing Project (ECP) shows that 29% of exascale applications use neighborhood collectives in performance-critical portions of their code [18]. Also, it is recorded that 80% of applications are expected to use accelerators [18]. To our knowledge, there is little literature that utilizes GPU communication for MPI neighborhood collectives. Table I shows the current literature on neighborhood collectives. Most of the research on neighborhood collectives is not GPU-aware. The only exception is recently published work[16], where the authors optimized GPU-Aware MPI neighborhood collectives for PETSc by changing the order in which the messages are sent to the outgoing neighbors. This way they could avoid contention on certain processes. In contrast to [16], we consider the physical topology and hierarchy of the processes and propose leader-based designs to improve the performance of neighborhood collectives for large messages. Our contributions in this paper are as follows:

- We develop a hierarchical ROCm-aware MPI Neighborhood Collective that considers the GPU interconnect topology to optimize the performance. The proposed design takes advantage of the high bandwidth links between certain GPUs and reduces the traffic load on links with lower bandwidth.
- We implement the new neighborhood allgather and allgatherv operations based on the proposed communication pattern design.

TABLE I: Comparison of Related Works

	Hoelfer [6]	Sameer [7]	Hoefler [8]	Träff [9], [10], [11], [12]	Mirsadeghi [13]	Ghazimirsaeed [14]	Ghazimirsaeed [15]	Khorassani [16]	Proposed Design
Any Comm. Pattern	✓	✗	✗	✗	✓	✓	✓	✓	✓
Msgs. $\geq$ 1M	✓	✗	✗	✗	✗	✗	✓	✗	✓
Topology-Aware	✗	✗	✗	✗	✗	✓	✓	✗	✓
Hierarchical	✗	✗	✗	✗	✗	✗	✓	✗	✓
Load-Aware	✗	✗	✗	✗	✗	✗	✓	✓	✓
GPU-Aware	✗	✗	✗	✗	✗	✗	✗	✓	✓

- We show the efficiency of the proposed design on two representative cluster architectures. The results show that we can achieve up to 7.03x and 3.82x speedup for Random Sparse Graph and Moore micro-benchmarks, respectively. We also achieve up to 2.29x speedup for a distributed SpMM kernel.

II. Background

A. MPI Topology Interface and Neighborhood Collectives

HPC applications often communicate in communication patterns that do not conform to the one-to-all, all-to-one, or all-to-all communication patterns provided by MPI [19]. Although these can be expressed using MPI point-to-point communication, it is difficult to optimize an application without unnecessary complexity. MPI allows users to define the logical topology of its processes. This information, known as process topology or virtual topology, is attached as an optional argument to the MPI communicator object. Virtual topology provides the opportunity to gather information about the communication pattern among the processes.

One common way to describe virtual topology is through the distributed graph interface in MPI. The distributed graph interface describes each process as a vertex of the graph and presents the communication relationship between the processes as edges. `MPI_Dist_graph_create_adjacent` is one of the most common distributed graph constructors in which each process specifies its own outgoing and incoming neighbors.

In neighborhood collectives, each process only communicates with the processes that are defined as one of its outgoing/incoming neighbors. The neighbors are specified using the communication pattern derived from the virtual topology graph of the processes. So there are two steps to handle neighborhood collective operations. First, we need to create the communication pattern with the virtual topology interface. This captures the irregular/complex communication patterns and stores them in an MPI communicator. Then we use this information to schedule MPI neighborhood collective operations.

B. AMD Instinct™ MI200 Series Accelerators

The AMD Instinct™ MI200 series accelerators are CDNA2-based compute accelerators for AI and HPC workloads [3]. The MI250 accelerator family includes the MI210, MI250, and MI250X accelerators. Each MI250 and MI250X GPUs consist of a pair of Graphics Compute Dies (GCDs) with four HBM2e modules per die for a total of 128GB memory. Therefore, each GCD has access to 64GB of memory. The two GCDs are connected within the GPU with a high-performance interconnect. The key difference between the MI250 and MI250X is the number of compute units. For the context of this paper, the number of compute units has minimal impact on communication optimization, as we do investigate collectives with a compute component. We use both variants to evaluate our designs on different Infinity Fabric™ and Networks. See Section VI-A for additional details.

The ROCm (Radeon Open Compute) [20] platform is a collection of open-source libraries specifically designed for developing high-performance software targeting AMD GPUs. Most GPU-Aware MPI implementations including OpenMPI [21], MVAPICH [22], and Cray MPICH [23] have support for ROCm.

C. Communication Concerns in MI250 and MI250X GPUs

On MI250 and MI250X GPUs, there are two main methods to transfer between devices, using the System Direct Memory Access (SDMA) engines or launching a copy kernel [24]. SDMA engines provide users the ability to overlap computation and communication, but they are limited to a theoretical maximum bandwidth of 50 GiB/s between GCDs. This is only 25% of the theoretical bandwidth between GCDs (200GiB/s), whereas a copy kernel has been shown to provide around ≈70% of the theoretical bandwidth between GCDs when transferring a 16MiB message[24]. Unfortunately, this is at the expense of using compute units or CUs (SMs in NVIDIA terminology) which can increase GPU utilization and less opportunity to overlap computation and communication. In this paper, we launch a copy kernel to handle communications as it provides higher bandwidth.

Figure 1 shows a diagram of a single node with four MI250 GPUs or eight GCDs. Each red box shows a single MI250 GPU with a pair of GCDs as blue circles. The black lines in the figure represent the Infinity Fabric™ link between GCDs. The figure illustrates varying the number of Infinity Fabric™ links between GCDs. For instance, GCDs within the same GPU are interconnected with four links, whereas GCDs across different GPUs are linked by either one or two links. One

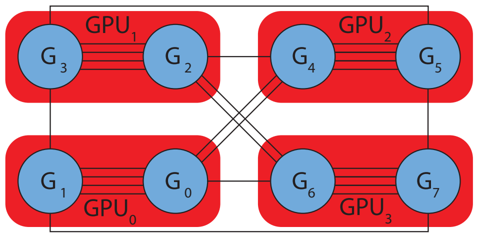


Fig. 1: Diagram of a node with 4 MI250 GPUs (8 GCDs)

other observation from the figure is that there are varying numbers of hops between GCDs. For instance, GCD 0 and 4 are connected through a minimum of one hop while GCD 0 and 2 are connected through a minimum of two hops. Having that said, two factors impact the maximum communication bandwidth between GCDs: 1) The number of Infinity Fabric links between GCDs 2) The number of hops between GCDs. For the remainder of this paper, the diagrams will not include the red box displaying the GPU for clarity.

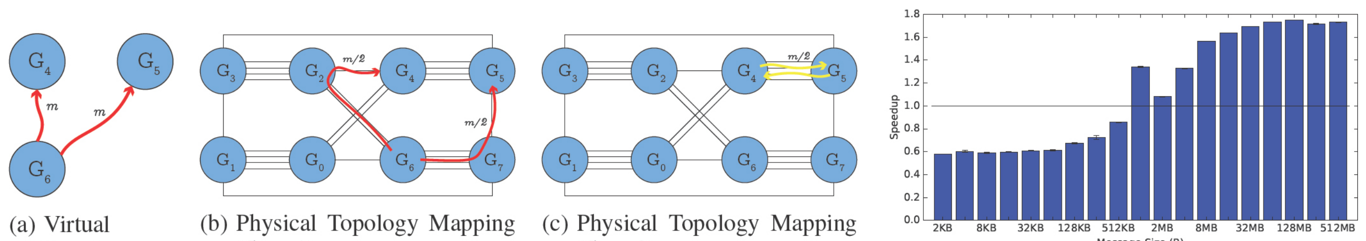
III. Motivation

As discussed earlier, the communication bandwidth between GCDs can vary between 38 GiB/s to 142 GiB/s for the same message size depending on the number of Infinity Fabric^{TM} links and the number of hops between them [24]. This raises the question **How can we design an efficient neighborhood collective operation that considers the number of infinity fabric^{TM} links and the number of hops between GCDs to improve performance?**

To validate the results in [24] within the context of MPI neighborhood collectives, we developed a simple benchmark test. Our goal is to show the impact of considering the physical topology of GPUs when communicating with multiple GCDs simultaneously. In this test, GCD 6 on GPU 3 sends a message of size m to GCDs 4 and 5 on GPU 2. The communication is shown in Figure 2(a). We consider two methods to handle this communication. In the first method, GCD 6 directly sends a message of size m to GCDs 4 and 5, as per the virtual topology. In the second approach, the communication is done in two steps. First, we send the upper half of the buffer to GCD 4 and the lower half of the buffer to GCD 5 (Figure 2(b)). Then, we exchange the partial data between GCDs 4 and 5 (Figure 2(c)). The second approach reduces the size of the messages in inter-GPU communication at the expense of increasing intra-GPU communications. Figure 2(d) shows the speedup of the scattering approach over the direct send approach, which provides up to 1.75x speedup. This experiment shows that considering the number of infinity fabric^{TM} links between GCDs in communication pattern design can benefit communication performance. This observation raises the following question: **How can we design an MPI neighborhood collective for large messages that minimizes inter-GPU communication and offloads it to intra-GPU links?**

In Figure 2, we chose a specific communication pattern and manually mapped it to the physical topology of the system, but MPI neighborhood collectives should ideally support any arbitrary communication pattern. For an MPI application with n processes, we have $2^{\binom{n}{2}}$ possible communication patterns that the MPI neighborhood collective could be invoked with. This is an incredibly large solution space. Similarly, we would have the same possible number of physical topologies to account for in our design. Pragmatically speaking, the system integrator provides a handful of physical topologies with each hardware generation, and we have tools to extract this information from the system at run-time. AMD provides ROCm System Management Interface (ROCm SMI) Library [25] for its platforms. It allows users to gather various metrics about GPU utilization, the number of Infinity Fabric^{TM} links between GPUs, NUMA domains, bandwidth between GPUs, etc. Although we have these tools to gather the physical topology, we do not want to create a mapping for each possible virtual topology. Therefore, we ask ourselves: **How can we extract the intra-node physical topology and create a heuristic that can map the virtual topology of the neighborhood collective to suit the hardware for arbitrary communication patterns and adjust the load between neighbors?**

As MPI applications scale, overheads often grow proportionally. MPI provides Cartesian, graph, and distributed graph topologies to represent the virtual topology of an application. In this paper, we focus on the distributed graph topologies as these have the lowest overhead and each process does not require information from every rank. This restriction makes optimization challenging, given the presence of partially incomplete data upon which our decisions must rely. It has been previously shown that hierarchical collectives improve collective communication [26], [27], [2]. Although hierarchical MPI neighborhood collectives have been studied before, they do not consider GPU communication [15]. This creates a new challenge that we aim to solve in this work: **How can we design a hierarchical neighborhood collective algorithm that minimizes inter-node communication while still maintaining a distributed graph topology and considering the GCD/GPU proximity?**



(a) Virtual Topology

(b) Physical Topology Mapping (Step 1)

(c) Physical Topology Mapping (Step 2)

(d) Motivational Results

Fig. 2: An example of how a virtual topology can be mapped to the physical topology to minimize inter-GPU communication and offload it to intra-GPU links using the proposed GCD scatter approach. Motivational results are shown to highlight its potential for large message sizes

IV. Related Work

Hoelfer and Träff [6] discuss the challenges of programming with point-to-point communication at scale and how collectives can simplify programmability for MPI users. They also note that traditional collectives do not fit common communication patterns such as 9-point stencils or 2D meshes. Therefore, they propose various MPI neighborhood collective APIs that can help users. Sameer et al. [7] showed the importance of neighborhood communication in various applications such as 3D-FFT, 4D Near Neighbor Exchange, NAno Scale Molecular Dynamics (NAMD), and Neural Networks. Hoefler and Schneider [8] used the topology interface to schedule communication and hardware level optimizations in collectives using RDMA for inter-node communication and XPMEM for intra-node messages.

Träff et al. [9], [10], [11], [12] propose extensions to the neighborhoods interface for isomorphic neighborhoods using Cartesian communicators and further optimize them using message-combining. Although these extensions outperform neighborhood collectives, they are not applicable to any sparse communication pattern. Lubbe [28] formulates performance expectations for MPI neighborhood collective operations and presents a micro-benchmark to assess these guidelines. Mirsadeghi et al. [13] and Ghazimirsaeed et al. [14] improve the performance of small message neighborhood collectives by utilizing message combining techniques and collaborative communication mechanisms, respectively. Ghazimirsaeed et al. [15] developed a hierarchical and load-aware design to improve the performance of large message neighborhood collectives.

All of the works discussed so far are CPU-based MPI neighborhood collectives. To our knowledge, the only research paper on GPU-based MPI neighborhood collectives is [16], where the authors change the order in which messages are sent to the outgoing neighbors to avoid contention on some GPUs. In contrast to [16] which focuses on neighborhood alltoall operations, we improve the performance of neighborhood allgather operations. Moreover, we consider the physical topology and hierarchy of the processes and propose leader-based designs to improve the performance of neighborhood collectives for large messages.

V. Design

The proposed design consists of two main parts. First, we create the communication pattern based on the physical GPU topology. The communication pattern design is incorporated in `MPI_Dist_graph_create_adjacent` function. Then, we use this information dynamically to design a communication schedule which is incorporated in the neighborhood collective functions, `MPI_Neighbor_allgather` and `MPI_Neighbor_allgatherv`.

A. Terminology and Definitions

To help guide readers through the design section, we first define the terminology and symbols that will be used in this section. A summary is shown in Table II. Incoming neighbors

TABLE II: Definition of the symbols used in the Neighborhood Collective design

Symbol	Description
I	Incoming Neighbors
I_t	Intra-Node Incoming Neighbors
I_r	Inter-Node Incoming Neighbors
O	Outgoing Neighbors
O_t	Intra-Node Outgoing Neighbors
O_r	Inter-Node Outgoing Neighbors
T_h	Intra-Node GPU Hop Topology Matrix
T_l	Intra-Node GPU Links Topology Matrix
GL_r	Inter-Node GCD Leaders
N_{out}	Number of GCD Leaders on a node
L_t	Intra-node Communication Load Matrix
L_r	Inter-Node Scattered Load Matrix
RM	Intra-Node Redistribution Matrix
PEM	Peer Exchange Matrix
NI_r	Intra-node Incoming Neighbors
PI_t	Peer GCD Incoming Intra-Node Neighbors
PI_r	Peer GCD Incoming Inter-Node Neighbors

I are defined as processes we are receiving messages from. We partitioned the incoming neighbors into two disjoint sets, int**ra**-node incoming neighbors I_t and int**er**-node incoming neighbors I_r. The processes to which we are sending messages are defined as outgoing neighbors O. We use to same convention to define the symbols int**ra**-node (O_t) and int**er**-node (O_r) outgoing neighbors.

We construct two topology matrices, T_h to define the number of hops and T_l to store the number of links between GCD pairs. The row and column index reference the sending and receiving GCDs. The value of the matrix at those indexes is either the number of hops or the number of links. The construction of these matrices and how we use them will be explained in detail in Section V-B.

We refer to the GCDs that are located on the same GPU as **peers**. For example, on GPU_0 we have two GCDs: G_0 and G_1. Therefore, G_0 is the peer GCD of G_1, and vice versa. On each GPU, we select the even index GCD as the **leader**. We create a set from the GCD leaders, and we call them GL_r. This is a 1D vector where each index contains the rank associated with each leader GCD.

To construct the communication pattern, each GCD needs to know the intra-node incoming neighbors (I_t) of its peer. This information can be obtained with a simple send/receive and is stored in the int**ra**-node Peer Incoming neighbors matrix PI_t. This is a 1D matrix where each entry is an int**ra**-node incoming neighbor of the peer GCD.

Using the above topology matrices, we construct an int**ra**-node L_t load matrix where each matrix value is the amount of data we send. Again, the rows/columns are GCD indexes and the value is the load. We have a similar matrix for the int**er**-node load L_r. Despite representing load, the structure differs. It is a list of lists, where we have two lists, one for incoming and another outgoing neighbors. Each element in the sub-list is the payload that should be sent to each neighbor. The specifics of construction and usage of the data structures outlined above will be discussed in detail in the remainder of this section.

The inter-node algorithm uses two other matrices that are yet to be defined: Intra-Node Redistribution Matrix (RM) and Peer Exchange Matrix (PEM). The RM matrix is a list of lists, the rows are the number of inter-node incoming neighbors and each sublist has the GCDs that the data must be redistributed to. The PEM matrix is a 1D list with a length equal to the number of inter-node incoming neighbors. The values of this matrix are either zero or one depending on if data should be exchanged with the peer GCD.

B. Intra-Node Communication Pattern Design

As mentioned earlier, two main metrics impact the performance of the GCD-to-GCD communication: the number of hops between GCDs and the number of Infinity Fabric™ links between them. We need this information to design efficient neighborhood collective operations based on the physical topology of the system. We use ROCm SMI library [25] to gather these metrics. We save this information into two matrices T_h and T_l which represent the number of hops and links, respectively.

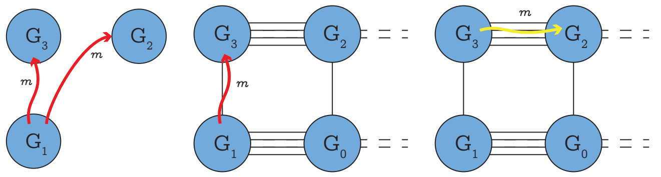
Although the focus of this work is on AMD GPUs and Infinity Fabric™ links, we believe that the presented collective algorithms are vendor-neutral. However, our implementation would need minor modifications, for example, we would replace rocm-smi with nvidia-smi for it to run on NVIDIA GPUs.

Using the physical topology of the system and the notion of GCD peers, we create an intra-node communication pattern. Our proposal is generalizable to any arbitrary communication pattern, but we choose two scenarios to aid with our explanation. In Figure 2, we present an example where the neighbors are not directly connected via any Infinity Fabric™ links. In this scenario, we create the communication pattern based on the number of hops between the processes. We obtain the total number of hops (n_h) to each outgoing neighbor by summing the appropriate entries of the T_h matrix. Then, we calculate the load to each outgoing neighbor using Equation 1. Using the calculated $load_H$ we modify the count parameter of the `MPI_Send/Recv` operations in the neighborhood collective call and use the scattering approach explained in Section III.

$$\text{Load}_h(\text{neighbor}) = \frac{n_h - T_h[\text{my\_rank}][\text{neighbor}]}{n_h} \quad (1)$$

The other illustrative case can be seen in Figure 3. In this example, G_1 and G_3 are directly connected via an Infinity Fabric™ link while G_1 and G_2 are not directly connected. We use the matrix T_l to count to total number of links to each neighbor and store it in the parameter n_l. Then, we set the appropriate load for each neighbor using Equation 2. For the example in Figure 3, $Load_l$(neighbor) would be 1 for G_3 and 0 for G_2 using Equation 2. So, G_1 sends the whole message directly to G_3 (Figure 3(b)) and it is forwarded from G_3 to G_2 (Figure 3(c)).

$$\text{Load}_l(\text{neighbor}) = \frac{T_l[\text{my\_rank}][\text{neighbor}]}{n_l} \quad (2)$$



(a) Virtual Topology (b) Physical Topology Mapping (Step 1) (c) Physical Topology Mapping (Step 2)

Fig. 3: Intra-Node communication pattern when GCD has two neighbors on the same device that are attached with *at least one* Infinity Fabric™ link.

Now that we have set the stage on how the loads are calculated, we use Algorithm 1 to explain the proposed intra-node communication pattern design that works for any virtual topology. First, we construct the intra-node communication load matrix L_t using intra-node neighbors I_t and O_t and the topology matrices T_h and T_l. This matrix is populated using Equation 1 and Equation 2, respectively. In the creation of this matrix, we first iterate over outgoing intra-node neighbors O_t. Then in Line 2, we check if this neighbor has a peer neighbor and adjust the load if it exists. In Line 3, we obtain the total number of links from our GCD to each GCD pair. If we have adjacent links, we assign a load to each neighbor using Equation 2. If there are no links, in Lines 8-9, we adjust the load to each GPU based on the number of hops to each peer using Equation 1.

To calculate the adjusted load for the incoming neighbors (I_t), we obtain our peer GCD's incoming neighbors and store it in (PI_t). This is required since we need information on how data is now distributed between the two GCDs. In Line 14, we iterate over $I_t \cap PI_t$ as we only need to modify common incoming neighbors. Then as previously explained, we adjust the load based on the number of links or hops.

Algorithm 1: Intra-Node Communication Load Matrix

```
Inputs : $O_t$, $I_t$, $T_l$, $T_h$
Output: $L_t$
1  foreach nbr in $O_t$ do
2      if nbr has peer GCD in $O_t$ then
3          $n_l$ = $T_l$[rank][nbr] + $T_l$[rank][peer]
4          if $n_L$ != 0 then
5              $L_t$[nbr]  = Load$_l$(nbr)
6              $L_t$[peer] = Load$_l$(peer)
7          else
8              $L_t$[nbr]  = Load$_h$(nbr)
9              $L_t$[peer] = Load$_h$(peer)
10         end
11     end
12 end
13 $PI_t$ = get_peer_in_nbrs()
14 foreach nbr in $\{I_t \cap PI_t\}$ do
15     $n_l$ = $T_l$[rank][nbr] + $T_l$[peer_rank][nbr]
16     if $n_L$ != 0 then  $L_t$[nbr]  = Load$_l$(nbr)
17     else $L_t$[nbr]  = Load$_h$(nbr)
18 end
```

C. Inter-Node Communication Pattern Design

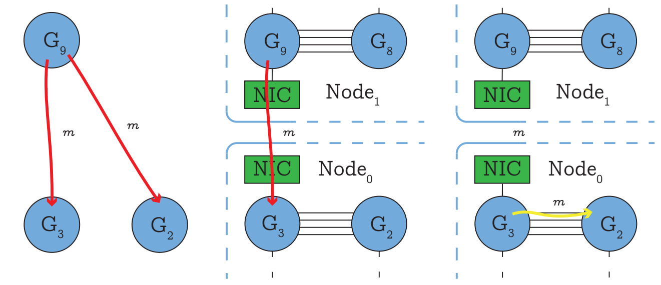
In this section, we explain the proposed inter-node communication pattern design, which is implemented in a hierarchical manner on top of the intra-node communication pattern discussed in Section V-B. To better explain the proposed design, first, we showcase three possible communication patterns presented in Figure 4, Figure 5, and Figure 6. Then, we discuss how the proposed design handles each case. Finally, we present the proposed algorithm that can be applied on any arbitrary communication pattern.

Figure 4 shows the scenario where a GCD has two neighbors on another node which are peers. Rather than the original communication pattern in which G_9 sends the message to each neighbor individually, we send the data to one GCD, then have it forwarded to its peer. This way, instead of sending two messages of size m across the network, only one message is sent across the network. In other words, we reduce the number of inter-node communications and offload it to the intra-node interconnect. Since the inter-node interconnects have significantly lower bandwidth than intra-node interconnects [5], this can improve communication performance.

In Figure 5, we consider the scenario that a GCD has multiple neighbors on another node where some are peer GCDs. In this example, G_g on node 1 sends a message of size m to the GCDs G_0, G_1, G_2, and G_3 on node 2, where G_0 and G_2 are peers of G_1 and G_3, respectively. In Step 1, we select a leader GCD in each GPU and scatter the message to those GCDs. In this example, G_0 and G_2 are the leaders. This way, instead of sending a message of size m across the network to each GCD ($4 \times m$), we send a message of size $m/2$ to G_0 and G_2 ($2 \times (m/2)$). In other words, we reduce the inter-node network load by $4X$. Then, in Step 2, the leader GCDs exchange their data so that they have the whole message of size m. Finally, in Step 3, the leader GCDs send the message to peer GCDs.

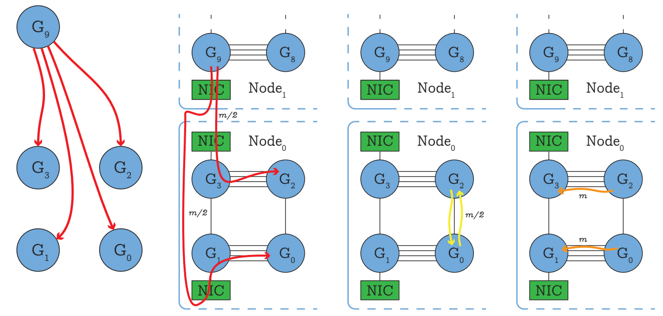
The last illustrative case shown in Figure 6 presents an example that we have multiple neighbors on another node where none are peers. This example follows the same scatter-gather Steps 1 and 2 from the previous example.

So far in this section, we discussed how to create inter-



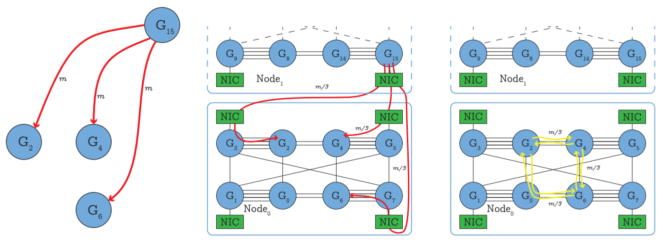
(a) Virtual Topology (b) Step 1 (c) Step 2

Fig. 4: Inter-Node Case 1: A GCD has two neighbors on another node that are *all* peer GCDs.



(a) Virtual Topology (b) Step 1 (c) Step 2 (d) Step 3

Fig. 5: Inter-Node Case 2: A GCD has multiple neighbors on another node where *some* are peer GCDs.



(a) Virtual Topology (b) Step 1 (c) Step 2

Fig. 6: Inter-Node Case 3: A GCD has multiple neighbors on another node where *none* are peer GCDs.

node communication patterns for different scenarios. In the remainder of this section, we explain the proposed inter-node communication pattern design that works for any virtual topology. To this purpose, we define three matrices:

1) Inter-Node Scattered Load Matrix (L_r)
2) Intra-Node Redistribution Matrix (RM)
3) Peer Exchange Matrix (PEM)

The three matrices correspond to each step in the given examples. However, some matrices may be populated with empty values if the specific neighborhood communication pattern is not required. For example, in Inter-Node Case 3 shown in Figure 6, L_r and RM will be populated with values, but PEM will be empty since there are no peer GCDs. Therefore, as desired, no Step 3 will occur. The goal of these matrices is to create a new communication pattern that will minimize inter-node communication and offload that work within the node to better utilize the intra-node bandwidth capabilities. To construct these matrices, we rely upon our inter-node incoming (I_r) and outgoing (O_r) neighbors.

1) Inter-Node Scattered Load Matrix: To better utilize inter-node network bandwidth we want to scatter the data across the inter-node neighbors so that each node has a partial copy of the required data. For the outgoing neighbors, this is relatively simple, as O_r contains the necessary information. First, we create the outgoing GCD leaders GL_r by extracting the neighbors with an even GCD index from the O_r list. This allows us to reduce load to GPUs with two neighboring processes. Then, in Lines 2-5 of Algorithm 2, we iterate over GL_r to count how many leader GCDs we have on each node and store it in N_{out}. In Lines 6-13, we modify the load to each neighbor. If a neighbor is a non-leader GCD, its load would be zero. Otherwise, we send a fraction of the data to it. These loads are placed onto L_r[out].

Obtaining L_r[in] is somewhat more complicated, as we need to know the incoming neighbors of all ranks on this node. We use an intra-node allgatherv operation to obtain I_r from all other ranks. This new matrix NI_r is also used in Section V-C2 and Section V-C3. Please note that the overhead of the allgatherv operation is minimal since it is a small message allgatherv happening within a node with a maximum of 8 processes per node (we have a maximum of 8 GCDs per node in most platforms). Also, it is called during the creation of the communication pattern. As stated earlier, the communication pattern is created once and used several times for different neighborhood collective calls.

In Line 15 we iterate over each GCD_{leader} and we count the number of incoming neighbors in Line 16-18 and store it in N_{in}. Then in Line 19 we obtain the non-leader GCD and we iterate over that. In Lines 20-23 we only count a neighbor if it does not have a common neighbor with the leader GCD. Finally, in Line 25 we populate L_r[in].

Algorithm 2: Inter-Node Scatter Load Matrix

```
   Inputs : $O_r$, $I_r$
   Output: $L_r$
1  $GL_r$ = get_GCD_leaders($O_r$)
2  foreach nbr in $GL_r$ do
3      node = get_node(nbr)
4      $N_{out}$[node]++
5  end
6  foreach nbr in $O_r$ do
7      if nbr in $GL_r$ then
8          node = get_node(nbr)
9          $L_r$[out][nbr] = 1.0 / $N_{out}$[node]
10     else
11         $L_r$[out][nbr] = 0.0
12     end
13 end
14 $NI_r$ = intra_node_allgatherv($I_r$)
15 foreach $GCD$ in $GCD_{leader}$ do
16     foreach nbr in $NI_r$[GCD] do
17         $N_{in}$[nbr]++
18     end
19     $GCD_{peer}$ = get_peer($GCD$)
20     foreach nbr in $NI_r$[$GCD_{peer}$] do
21         if nbr not in $NI_r$[GCD] then $N_{in}$[nbr]++
22     end
23 end
24 foreach nbr in $I_r$ do
25     $L_r$[in][nbr] = 1.0 / $N_{in}$[nbr]
26 end
```

2) Inter-Node Redistribution Matrix: As we intend to scatter the message across neighbors on a remote node, we need a matrix that will inform us on how to redistribute the data to the appropriate ranks within the nodes (Step 2 in Figure 5 and Figure 6). To this purpose, we create the Redistribution Matrix (RM) as shown in Algorithm 3. We use the matrix NI_r from Section V-C1. For each GCD in the node, we append each of its neighbors to RM if that neighbor is a leader GCD.

3) Peer Exchange Matrix: Finally, we create the Peer GCD Redistribution Matrix in Algorithm 4 so that data can be exchanged within the GPU (Step 2 in Figure 4 and Step 3 in Figure 5). The values of PEM are initialized to zero. We iterate over our incoming neighbors (I_r) and check if the neighbor exists on our peer GCD. If the neighbor exists, we set a flag (PEM) to determine whether we should exchange data.

D. Neighborhood Collective Design

Sections V-B and V-C discussed the communication pattern design and saved the communication pattern information in four matrices. In this section, we use those matrices to develop the neighborhood collective operations: `MPI_Neighbor_allgather` and `MPI_Neighbor_allgatherv`. The neighborhood collective design has three main steps:

1) Data is scattered across intra and inter-node neighbors using the information from L_t and L_r matrices.
2) Then in parallel, intra-node data is exchanged between peer GCDs using the matrix L_t and our inter-node data is also redistributed to our leader GCDs using RM.
3) Finally, leader GCDs copy to their intra-node peers.

Algorithm 5 shows the details of each step. Lines 1-17 correspond to the first step of the design. First, we post our `MPI_Irecv` calls for each neighbor in I. The load for that neighbor

Algorithm 3: Intra-Node Redistribution Matrix

```
  Inputs : $NI_r$
  Output: $RM$
1 foreach $GCD$ do
2     foreach nbr in $NI_r$[GCD] do
3         if nbr is leader GCD then
4             $RM$[nbr] += [$GCD$]
5         end
6     end
7 end
```

Algorithm 4: Peer Exchange Matrix

```
  Inputs : $I_r$, $PI_r$
  Output: $PEM$
1 foreach nbr in $I_r$ do
2     if nbr in $PI_r$ then
3         $PEM$[nbr] = 1
4     end
5 end
```

Algorithm 5: MPI_Neighbor_allgather Design

```
   Inputs: $I$, $I_r$, $I_t$, $L_t$, $L_r$, $O$, $O_r$, $O_t$, $RM$, $PEM$
1  foreach nbr in $I$ do
2      if nbr $\in I_t$ then
3          load = $L_t$[nbr]
4      else if nbr $\in I_r$ then
5          load = $L_r$[in][nbr]
6      end
7      irecv(load, nbr)
8  end
9  foreach nbr in $O$ do
10     if nbr $\in O_t$ then
11         load = $L_t$[nbr]
12     else if nbr $\in O_r$ then
13         load = $L_r$[out][nbr]
14     end
15     isend(load, nbr)
16 end
17 wait_all
18 foreach nbr in $I$ do
19     if nbr $\in I_t$ then
20         load = $L_t$[nbr]
21         isend(load, nbr)
22         irecv(load, nbr)
23     else if nbr $\in I_r$ then
24         if leader GCD then
25             foreach nbr_rdist in $RM$ do
26                 load = $RM$[nbr]
27                 isend(load, nbr_rdist)
28                 irecv(load, nbr_rdist)
29             end
30         end
31 end
32 wait_all
33 foreach nbr in $I_r$ do
34     if $PEM$[nbr] == 1 then
35         if Rank is leader GCD then
36             isend(nbr)
37         else
38             irecv(nbr)
39         end
40     end
41 end
42 wait_all
```

is chosen in Lines 2-6 where we check whether the neighbor is intra- or inter-node and use the appropriate matrix (L_t or L_r). If the load value in L_t/L_r is zero, `MPI_Isend/Irecv` does not transfer any data. We do the same procedure in Lines 9-16 for the corresponding `MPI_Isend` calls. To complete this step, we issue an `MPI_Waitall`. We communicate with each neighbor in $I \cup O$ as theoretically, these messages should not cause any congestion and conflict with each other during this step as intra-node communication is over the Infinity Fabric$^{\text{TM}}$ network and all inter-node messages are transferred with the InfiniBand/Slingshot network.

Then in Step 2 (Lines 18-32), we start the redistribution of the data for the incoming neighbors so we only iterate over I. For each neighbor we check if it is intra-node (in I_t); if so, we obtain the load L_t in Line 20 and exchange the data. If that neighbor does not need to exchange data, the load value in L_t would be zero, causing the `MPI_Isend/Irecv` function to promptly return without any communication taking place. For the inter-node incoming neighbors (I_r) we exchange the data with the GCD leaders using RM so that each leader has a complete set of data in Lines 24-29.

Finally, we execute Step 3 in Lines 33-42. Here we check the peer exchange matrix (PEM) and transfer the data to the peer GCD, if required. We then wait for all communication on that process before exiting the collective operation.

VI. Experimental Results and Analysis

In this section, we evaluate the proposed design using two micro-benchmarks and an SpMM kernel. The micro-benchmarks allow us to find an upper-bound to the proposed algorithm and the SpMM kernel allows us to project the expectation for applications.

A. Experimental Setup

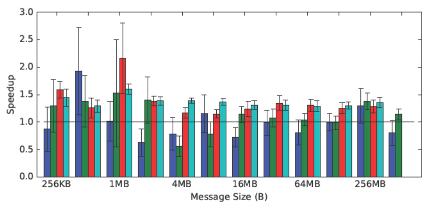
The experiments are conducted on two clusters. The first cluster, which we refer to as *Cluster A*, has four nodes connected with Mellanox ConnectX-6 InfiniBand cards. Each node has four AMD Instinct$^{\text{TM}}$ MI250 accelerators [3], so the cluster has a total of 16 GPUs (32 GCDs). These GPUs are connected with Infinity Fabric$^{\text{TM}}$ links as shown in Figure 1. *Cluster A* has an AMD EPYC 7643 48-Core processor and runs the GNU/Linux distribution Red Hat 8.5.0-10.

To evaluate the result on a different topology and at a larger scale, we use a second cluster which we refer to as *Cluster B*. This cluster has the same type of compute nodes as in Frontier. Each node has an AMD EPYC$^{\text{TM}}$ 7A53 64-Core CPU and four AMD Instinct$^{\text{TM}}$ MI250X accelerators. Figure 6 shows the Infinity Fabric$^{\text{TM}}$ topology of the GPUs in *Cluster B*. The nodes are connected with HPE's Slingshot Interconnect [29] and runs the GNU/Linux distribution SUSE Linux, kernel version 5.3.18.

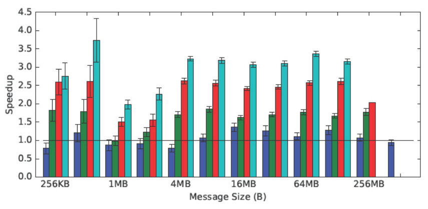
On both of these clusters, we used Open MPI from the master branch at the commit `450ae3a` (roughly version 5.0.x) and UCX version 1.11.2. At the time of developing this code libfabrics did not have full ROCm support on the Slingshot Network. Therefore, on *Cluster B*, we ran UCX over TCP for inter-node communication.

B. Random Sparse Graph

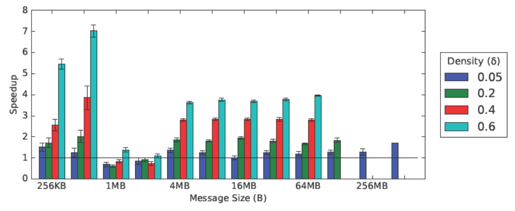
To evaluate the proposed design, we use the Erdős–Rényi Random Sparse Graph (RSG) micro-benchmark which has also been used in [8], [13], [14], [15]. The graph $G(E, V)$ is used to model a communication pattern where each vertex $v \in V$ corresponds to an MPI rank and each edge $e \in E$ corresponds to an outgoing neighbor from one process to



(a) 8 GCDs

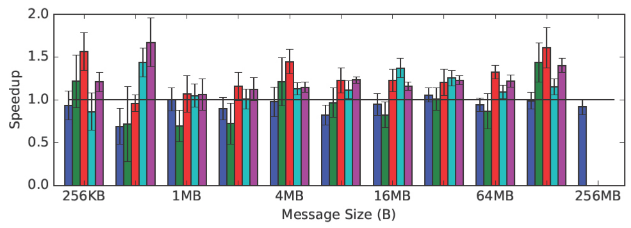


(b) 16 GCDs

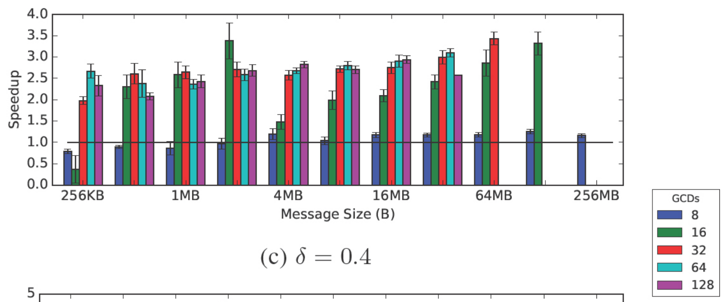


(c) 32 GCDs

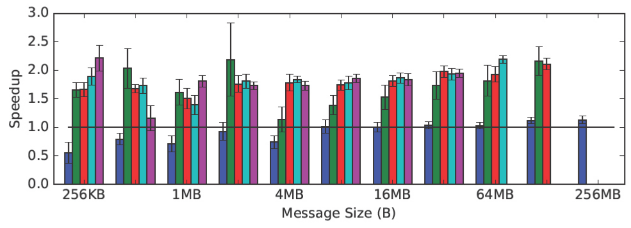
Fig. 7: Scaling of MPI_Neighbor_allgather with the Random Sparse Graph micro-benchmark for different density factors (δ) on *Cluster A*



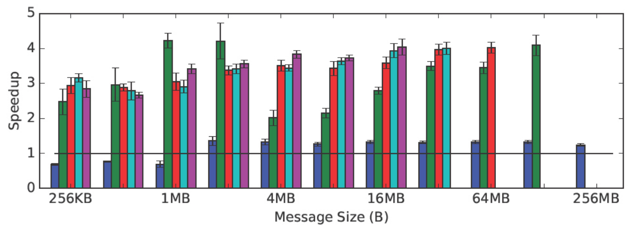
(a) $\delta = 0.05$



(c) $\delta = 0.4$



(b) $\delta = 0.2$



(d) $\delta = 0.6$

Fig. 8: Scaling of MPI_Neighbor_allgather with the Random Sparse Graph micro-benchmark for different density factors (δ) on *Cluster B*.

another. The outgoing neighbors of random sparse graph are randomly selected using a density factor $\delta \in (0, 1) \subset \mathbb{R}$.

Figures 7 and 8 evaluate the performance of the proposed design over the neighborhood allgather design in OpenMPI on *Cluster A* and *B*, respectively. The results are presented for four density factors (δ) in Random Sparse Graph. Figure 7(a) shows that for a single node run (8 GCDs) on *Cluster A*, we obtain a performance improvement of up to 2.16x and 1.60x for $\delta = 0.4$ and $\delta = 0.6$, respectively. The performance improvement is considerable for almost all message sizes shown in the figure. Please note that data points for messages larger than 256MB are not shown for large densities. The reason for this is that for large messages and higher densities, we require more memory as we have more incoming messages, and we are limited by 64GB memory for each GCD. One observation from the figure is that for extremely small densities (e.g., $\delta = 0.05$) we get little to no performance improvement. The reason for this is that in this case, the communication is so sparse that there is no room to take advantage of the proposed hierarchical design as we mostly have a single neighbor on each GPU.

Figures 7(b) and 7(c) show the same trends for larger process counts. Figure 7 shows that as we increase the scale from one node to four nodes, we achieve higher speedups, which shows the efficiency and scalability of the proposed inter-node communication pattern design.

Figure 8 evaluates the performance of the proposed design on *Cluster B*. For a single node (8 GCDs), we observe a relatively modest performance improvement of up to 1.35x when $\delta = 0.6$. This is relatively similar to what we observed on a single node of *Cluster A*. These observations show that the proposed intra-node algorithm is portable as we have verified its improvement on multiple Infinity Fabric$^{\text{TM}}$ topologies.

Figure 8 also shows that the proposed design provides significant speedup as we scale to more nodes. Due to the larger number of nodes in *Cluster B*, we could scale up to 128 GCDs, utilizing a total of 16 nodes. The results show that we can achieve 1.32x to 4.02x as we increase the number of GCDs, and the speedup is consistent between densities as they scale.

The overheads with this topology creation went from around 11ms with the default Open MPI implementation to 22ms with our method for an 8MB message at 128 GCDs. This results in roughly doubling our overhead but to recover this cost roughly 22 iterations are needed. Many application iterate thousands of time so the cost of this topology creation will be recovered relatively quickly. This was the case regardless of graph density, so these statements will also hold true to the the overheads of the Moore neighborhoods we will discuss in the next section.

C. *Moore Neighborhood*

Moore neighborhoods are a generalization of the stencil communication patterns and can be defined with dimension (d) and radius (r). Stencil codes or Halo-Exchanges are commonly used to split up a problem that cannot fit on a single node [19]. The dimension corresponds to the number of grid dimensions and the radius is the maximum number of hops a process communicates with. The total neighbors can be calculated with $(2r+1)^d - 1$. Using different values of d and r, we can explore different communication patterns.

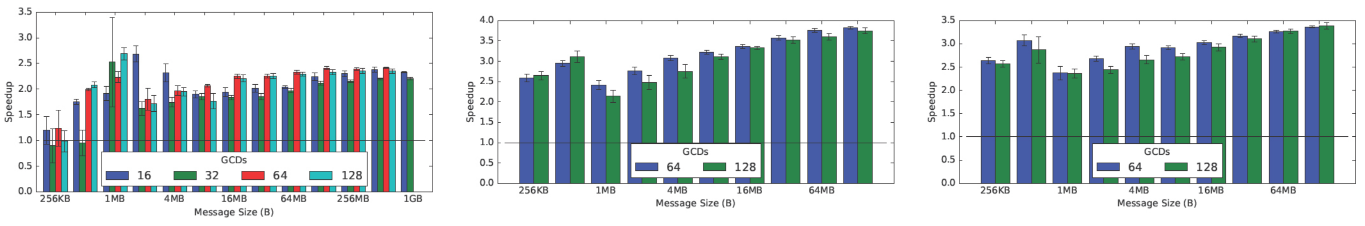
Figure 10 shows the performance improvement of the proposed design for Moore neighborhood on *Cluster A*. One observation from this figure is that we get higher speedups for messages of size 256KB and 512KB. This is because as the proposed design scatters the message into smaller chunks, these messages fall below the eager threshold and gain more performance improvement. For the remaining message sizes from 1MB to 1GB, the performance improvement is as expected, and we achieve a higher speedup as we increase the message size.

Figure 9 shows the Moore neighborhood speedup on *Cluster B*. In Moore neighborhood, the dimension and radius values are constrained by the number of ranks. Since we have access to more GCDs on *Cluster B*, we are able to explore more neighborhood dimensions and radiuses as can be seen in Figures 9(b) and 9(c). Figure 9 shows an almost similar trend as *cluster A* for different radiuses and diameters. In general, the experimental results in this section show that regardless of the platform and Moore neighborhood metrics (d and r), the proposed design provides considerable speedup for large messages.

D. *Sparse Matrix Matrix Multiplication Kernel*

Sparse matrix matrix multiplication (SpMM) is an important kernel in computational linear algebra and big data analytics [30]. SpMM calculates the multiplication of two matrices $A \times B = C$ where A is a sparse matrix and B is a dense matrix. To distribute the data among the processes, we employ the row-wise block-striped decomposition of the input matrices. The Allgather operation is used to gather the columns of matrix B at all processes. Due to the sparsity of matrix A, we can selectively collect data only for its non-zero elements and establish the communication pattern accordingly. For this, we extend the SpMM algorithm used in [14] so that computation and communications happen on GPUs. Where Random Sparse Graph and Moore neighborhood are developed using `MPI_Neighbor_allgather`, `MPI_Neighbor_allgatherv` is used to implement the SpMM kernel as it can account for different receive counts and displacements.

For the evaluations in this section, we use a variety of matrices from The University of Florida Sparse Matrix Collection [31]. To evaluate our design fairly, the cost of topology creation is included in our measured values and in our subsequent speedup calculation. Figure 11 shows that the proposed design provides 0.85x to 2.29x speedup depending on the the



(a) $d = 2$, $r = 1$ (b) $d = 2$, $r = 2$ (c) $d = 3$, $r = 1$

Fig. 9: Moore Neighborhood Speedup of MPI_Neighbor_allgather for different r and d values on *Cluster B*.

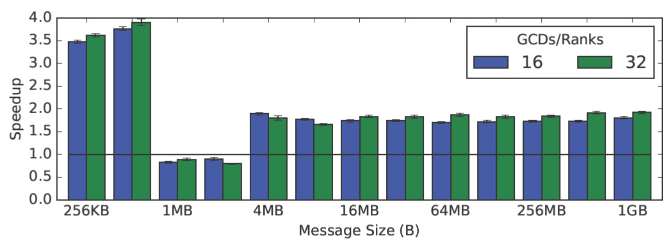


Fig. 10: Speedup of MPI_Neighbor_allgather with the Moore neighborhood on *Cluster A* with radius ($r = 1$) and diameter ($d = 2$).

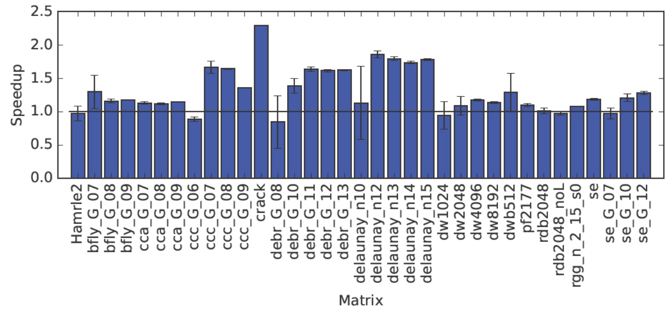


Fig. 11: Speedup of MPI_Neighbor_allgatherv with the SpMM Kernel for different matrices on *Cluster A* with 32 GCDs.

TABLE III: Sizes for the Matrices of Interest

Matrix	Rows	Columns	Non-Zeros
ccc_G_06	384	384	1152
debr_G_08	256	256	1018
delaunay_n10	1,024	1,024	6,112
crack	10,240	10,240	60,760

number of non-zero elements, the size of the matrices and the distribution of the non-zero element. We see a slowdown for small matrices such as 'ccc_G_06' as the message sizes fall below the large message threshold of 256KB we have designed our collective for. The matrix 'debr_G_08' and 'delaunay_ n10' are similar in size but 'delaunay_n10' has more non-zero elements and this is reflected in their speedups.

Table III shows the number of rows, columns, and the non-zero elements for a few matrices in Figure 11. The highest speedup belong to matrix 'crack'. This matrix has more number of non zero elements and consequently has more room to take advantage of the proposed hierarchical design. On the other hand, matrices 'ccc_G_06' and 'debr_G_08' have fewer number of non-zero elements and they achieve lower speedup.

VII. Conclusion

HPC workloads have become ever more GPU-centric, and we have seen the uptake of AMD Instinct^{TM} MI200 Series Accelerators on large-scale state-of-the-art systems such as Frontier. In this paper we address the challenges associated with ROCm-Aware MPI neighborhood collectives using the complex Infinity Fabric^{TM} topology that connects the GPUs. We have proposed an intra- and inter-node communication pattern creation that uses ROCm-SMI to obtain system topology information to create a hierarchical and leader-based communication pattern. Using these new communication pattern, we have proposed new `MPI_Neighbor_allgather` and `MPI_Neighbor_allgatherv` collectives. We evaluate the performance of the new MPI implementation using a variety of benchmarks and a SpMM kernel on multiple platforms. For Sparse Random Graph micro-benchmark, we observe up to 2.16x speedup for intra-node algorithm and up to 7.03x for inter-node. With the Moore neighborhood micro-benchmark, we observe up to 3.82x speedup. Finally, with SpMM kernel, we obtain up to 2.29x speedup.

A. Future Work

In Section VI-C we saw significant speedup for the Moore neighborhood communication pattern. As a Moore neighborhood is a generalized stencil computation, we believe that porting an MPI application which uses this pattern such as as NekBone/hipBone could yield good performance improvement [32]. For inter-node communication we adjust the amount of data sent to each neighbor based upon whether it is a GCD peer. We would like to extend this to more complex heuristics to adjust communication between nodes but this also faces the challenges of maintaining the distributed topology requirements that we desire. As we saw our proposals were successful for dense neighborhoods, we should certainly explore the possibility of including these ideas in to traditional MPI collectives as they are inherently dense. Finally, as AMD's MI300 APUs are becoming available in the near future we would like to this design for that platform as there are more complex GCD arrangements on each GPU.

VIII. Acknowledgements

We would like to thank AMD colleagues Christopher Kime and Alex Habeger for providing access to compute resources, Edgar Gabriel for advice on installing ROCm-Aware Open MPI, and Bill Brantley for reviewing the paper.

References

[1] J. Glaser, T. D. Nguyen, J. A. Anderson, P. Lui, F. Spiga, J. A. Millan, D. C. Morse, and S. C. Glotzer, "Strong scaling of general-purpose molecular dynamics simulations on gpus," *Computer Physics Communications*, vol. 192, pp. 97–107, 2015.

[2] Y. H. Temuçin, A. H. Sojoodi, P. Alizadeh, B. Kitor, and A. Afsahi, "Accelerating Deep Learning Using Interconnect-Aware UCX Communication for MPI Collectives," *IEEE Micro*, vol. 42, no. 2, pp. 68–76, 2022.

[3] AMD. (2022) AMD INSTINCT™ MI200 SERIES ACCELERATOR. [Online]. Available: https://www.amd.com/system/files/documents/amd-instinct-mi200-datasheet.pdf

[4] T. 500. (2023) TOP500 November 2023. [Online]. Available: https://www.top500.org/lists/top500/2023/11/

[5] 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*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 118–136. [Online]. Available: https://doi.org/10.1007/978-3-030-78713-4_7

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

[7] S. Kumar, P. Heidelberger, D. Chen, and M. Hines, "Optimization of applications with non-blocking neighborhood collectives via multisends on the Blue Gene/P supercomputer," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[8] 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.

[9] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold, "Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations," in *Proceedings of the 22nd European MPI Users' Group Meeting*, ser. EuroMPI '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2802658.2802663

[10] J. L. Träff, A. Carpen-Amarie, S. Hunold, and A. Rougier, "Message-Combining Algorithms for Isomorphic, Sparse Collective Communication," *CoRR*, vol. abs/1606.07676, 2016. [Online]. Available: http://arxiv.org/abs/1606.07676

[11] J. L. Träff and S. Hunold, "Cartesian collective communication," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP 2019. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337848

[12] J. L. Träff, S. Hunold, G. Mercier, and D. J. Holmes, "Collectives and Communicators: A Case for Orthogonality: (Or: How to Get Rid of MPI Neighbor and Enhance Cartesian Collectives)," in *27th European MPI Users' Group Meeting*, ser. EuroMPI/USA '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 31–38. [Online]. Available: https://doi.org/10.1145/3416315.3416319

[13] 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.

[14] 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.

[15] S. M. Ghazimirsaeed, Q. Zhou, A. Ruhela, M. Bayatpour, H. Subramoni, and D. K. D. Panda, “A Hierarchical and Load-Aware Design for Large Message Neighborhood Collectives,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–13.

[16] K. S. Khorassani, C.-C. Chen, H. Subramoni, and D. K. Panda, “Designing and optimizing gpu-aware nonblocking mpi neighborhood collective communication for petsc*,” in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 646–656.

[17] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.0*, Jun. 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[18] D. E. Bernholdt, S. Boehm, G. Bosilca, M. Gorentla Venkata, R. E. Grant, T. Naughton, H. P. Pritchard, M. Schulz, and G. R. Vallee, “A survey of mpi usage in the us exascale computing project,” *Concurrency and Computation: Practice and Experience*, vol. 32, no. 3, pp. 1–19, 2020, e4851 cpe.4851. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4851

[19] P. G. Raponi, F. Petrini, R. Walkup, and F. Checconi, “Characterization of the communication patterns of scientific applications on blue gene/p,” in *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, 2011, pp. 1017–1024.

[20] AMD. (2023) AMD ROCm™ Open Ecosystem. [Online]. Available: https://www.amd.com/en/graphics/servers-solutions-rocm

[21] O. MPI. (2022) Open MPI: Open Source High Performance Computing. [Online]. Available: https://www.open-mpi.org/

[22] MVAPICH. (2022) MPI over InfiniBand, Omni-Path, Ethernet/iWARP, and RoCE. [Online]. Available: http://mvapich.cse.ohio-state.edu/

[23] C. MPICH. (2023) Cray MPICH documentation. [Online]. Available: https://cpe.ext.hpe.com/docs/mpt/mpich/index.html

[24] AMD. (2023) GPU-aware MPI with AMD ROCm™. [Online]. Available: https://gpuopen.com/learn/amd-%20lab-notes/amd-lab-notes-gpu-aware-mpi-readme/

[25] ——. (2022) ROCm System Management Interface (ROCm SMI) Library. [Online]. Available: https://github.com/RadeonOpenCompute/rocm_smi_lib

[26] I. Faraji, S. H. Mirsadeghi, and A. Afsahi, “Topology-aware gpu selection on multi-gpu nodes,” in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 712–720.

[27] C.-H. Chu, P. Kousha, A. A. Awan, K. S. Khorassani, H. Subramoni, and D. K. D. K. Panda, *NV-Group: Link-Efficient Reduction for Distributed Deep Learning on Modern Dense GPU Systems*. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3392717.3392771

[28] F. D. Lübbe, “Micro-benchmarking mpi neighborhood collective operations,” in *Euro-Par 2017: Parallel Processing*, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro, Eds. Cham: Springer International Publishing, 2017, pp. 65–78.

[29] 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*, ser. PEARC ’22. New York, NY, USA: Association for Computing Machinery, 2022. [Online]. Available: https://doi.org/10.1145/3491418.3530773

[30] S. Acer, O. Selvitopi, and C. Aykanat, “Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems,” *Parallel Computing*, vol. 59, pp. 71–96, 2016, theory and Practice of Irregular Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167819116301041

[31] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: https://doi.org/10.1145/2049662.2049663

[32] N. Chalmers, A. Mishra, D. McDougall, and T. Warburton, “HipBone: A performance-portable GPU-accelerated C++ version of the NekBone benchmark,” 2022. [Online]. Available: https://arxiv.org/abs/2202.12477