Cascade: a Collaborative Algorithm for Scalable And Efficient Neighborhood Allgather

Hamed Sharifian, Amirhossein Sojoodi, and Ahmad Afsahi
Department of Electrical and Computer Engineering
Queen's University
Kingston, Ontario, Canada
{hamed.sharifian, amir.sojoodi, ahmad.afsahi}@queensu.ca

***Abstract*—Neighborhood collectives are a critical feature of MPI, enabling efficient communication in applications with sparse communication patterns. This research proposes Cascade, a new algorithm for neighborhood allgather collective that organizes computing nodes along multiple paths based on their distance to the current node. In this approach, messages are forwarded along these paths and propagated until all outgoing neighbors receive them, reducing the communication time. Three performance models are developed to analyze the efficiency of the Cascade algorithm, the default Open MPI algorithm, and the recently proposed Distance-halving neighborhood algorithm in the literature, offering insight into communication cost, scalability, and expected behavior of the algorithms across different system configurations. Experimental results demonstrate that the Cascade algorithm achieves up to 9.54x and 7.05x speedup over Open MPI for random sparse graphs and Moore neighborhoods, respectively. Additionally, the algorithm improves performance by up to 5.25x for a sparse matrix-matrix multiplication kernel. The Cascade algorithm outperforms the Distance-halving neighborhood algorithm by up to 2.57x and 4.81x speedup for random sparse graphs and Moore neighborhoods, respectively. Moreover, Cascade achieves up to 1.61x performance gain over the Distance-halving neighborhood for the sparse matrix-matrix multiplication kernel. The predictions of our performance models closely match the experimental results.**

***Index Terms*—MPI, Neighborhood Collectives, Allgather, Performance Modeling**

I. Introduction

The Message Passing Interface (MPI) [1] is a standard programming model widely used in high-performance computing (HPC). The MPI standard defines interfaces and semantics for implementing the message-passing programming model in distributed memory systems. MPI defines several communication paradigms. The two most common paradigms are point-to-point and collective communication. Point-to-point is the most fundamental communication in MPI, where data is transferred directly from the memory of one process to another. Collective communication involves communication among a group of processes.

Neighborhood collectives are a newer class of communication in MPI. Neighborhood collectives focus on localized communication, where each process communicates in a limited neighborhood. In contrast, conventional collectives are global, involving either communication among all processes in a communicator or between each process and a root process. The neighborhoods are defined through the MPI virtual topology interface. The MPI virtual topology interface provides different operations, and the user gives the list of incoming and outgoing neighbors for each process.

MPI 3.0 introduced neighborhood collective communications to address the limitations of conventional collectives to handle the communication patterns commonly found in the applications. In many HPC applications, communications are dominated by localized interactions. Problems in different areas, such as scientific simulations, graph processing, and numerical methods, often involve data exchanges between neighboring elements in a grid, mesh, or graph. For instance, in weather modeling like WRF [2], computations need the data from adjacent regions, and in molecular dynamics applications like GROMACS [3], interactions occur between neighboring particles.

Neighborhood collectives in the state-of-the-art MPI libraries, including Open MPI [4], MPICH [5], and MVAPICH [6], suffer from scalability issues and performance bottlenecks because they use a naïve approach that relies on direct point-to-point communication with incoming and outgoing neighbors, without considering network topology or hardware characteristics.

In this research, a new algorithm for neighborhood allgather, called Cascade, is proposed. In this approach, each process sends its message along multiple paths. These paths include remote computing nodes. The nodes in each path are sorted based on their distance to the current node. Messages are initially sent to one of the outgoing neighbors on the nearest node in a path, which then forwards the message to the next node in the sequence of nodes in that path. This procedure continues with each receiver process forwarding the message to the subsequent node until all outgoing neighbors in the path have received it.

Our algorithm constructs a communication pattern based on virtual topology information, focusing on efficiency and scalability by reducing network congestion and the number of hops messages traverse. Our approach also employs message combining, if possible, to further reduce the communication overhead. Several algorithms are proposed for neighborhood allgather on graph topologies in the literature [7], [8], [9], [10], [11]. However, our work is the first work that considers the locality of the processes, network topology, hop count, and memory footprint at the same time. The contributions of this

paper are as follows:

- We propose the Cascade algorithm for neighborhood allgather. This algorithm improves the scalability and performance of neighborhood allgather for medium and large messages by reducing contention, memory consumption, and the number of hops messages traverse.
- We develop an analytical model for the Cascade algorithm, along with two additional algorithms, the default Open MPI and the Distance-halving neighborhood [10], to analytically evaluate their performance. These models serve as a tool for analyzing the behavior of the algorithms in different scenarios and facilitate autotuning on real-world HPC platforms. Our model is the first probabilistic performance model for random sparse graphs based on the Max-rate Model [12].
- We perform empirical evaluation on a large cluster to analyze the efficiency of the proposed algorithm. We implemented our algorithm in Open MPI and measured its performance with random sparse graphs and Moore neighborhood microbenchmarks, and a sparse matrix-matrix multiplication kernel. We compare our algorithm against the default Open MPI and the Distance-halving neighborhood algorithm [10].
- We gather the performance characteristics of our cluster and use our performance models to calculate the communication latency of the aforementioned algorithms. We then compare the experimental and analytical results to assess our performance models.

The rest of this paper is organized as follows: Section II provides the necessary background and motivation for this work. Section III reviews the relevant research in the literature. The Cascade algorithm is presented in Section IV, followed by the analytical modeling in Section V. Section VI details the evaluation of the proposed algorithm, and Section VII discusses conclusion and future work.

II. Background and Motivation

According to a survey [13], 46% of high-performance computing applications currently employ fixed neighborhood communication patterns. Moreover, 29% anticipate adopting neighborhood collectives in future updates, and 9% consider them essential for their performance-critical operations.

MPI Neighborhood allgather operation `MPI_Neighbor_Allgather` is a collective communication operation where each process receives data from all its incoming neighbors while sending its data to all its outgoing neighbors. Fig. 1 illustrates this operation based on a virtual topology in a communicator containing four ranks. Communicators are a group of processes. In MPI, each process within a communicator is assigned a unique identifier called a rank. In the rest of this paper, we use process and rank interchangeably. In Fig. 1, each rank sends all data in its send buffer to the outgoing neighbors and gathers the data from the incoming neighbors in its receive buffer according to the order of its incoming neighbors. In

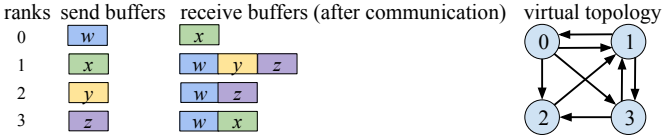


Fig. 1: Visualization of the send and receive buffers in a neighborhood allgather operation for an MPI communicator with four processes (ranks), illustrating buffer organization and message flow based on the given virtual topology.

the given example, the direction of the arrows in the directed graph indicates the flow of communication or the relationship between neighboring ranks, with the arrow pointing from one rank to its outgoing neighbor.

The most important challenges with the current implementations of neighborhood allgather are poor performance and scalability due to congestion, ignoring the system and network topology, and memory consumption. As message sizes increase, network congestion becomes a significant challenge, reducing overall performance and limiting the scalability of neighborhood collectives.

Network congestion is more likely with large messages because they occupy network resources for longer periods, leading to increased queuing and contention. Although avoiding congestion is important, it is equally crucial to maximize the use of available network bandwidth to minimize communication time. Efficient management of message transfers ensures that the network bandwidth is fully utilized, avoiding delays. This balance is essential for higher throughput, better performance, and scalability in HPC systems. This leads to the question: **How can we design an algorithm for neighborhood allgather that exploits the available network bandwidth and avoids congestion simultaneously?**

Avoiding congestion becomes even more crucial when traffic depends on the bisection bandwidth of the network. Managing traffic distribution is essential in topologies such as fat-tree and torus, where the bisection bandwidth is often lower than the injection bandwidth. It is important to prevent congestion, specifically over the bisection links, while also effectively utilizing the available bandwidth across all network links. Balancing traffic flow across the network topology ensures that no single link, especially those critical to the bisection, becomes overloaded, thereby maintaining overall network performance and reducing bottlenecks. This prompts the question: **How can we limit communication over the bisection links of the network in neighborhood allgather?**

Intra-node communication is considerably faster than inter-node communication. However, for large messages, the growing number of CPU cores per node can rapidly saturate the available memory bandwidth, thus limiting the performance of both inter-node and intra-node communication. As a result, while it is essential to leverage both network and memory bandwidths simultaneously to improve overall communication efficiency, it is equally important to mitigate memory contention to avoid performance degradation. Apart from that,

copying the messages between temporary and final buffers is more costly for large messages due to its impact on the overall performance of the neighborhood collectives. Considering this, the question is: **How can we develop an algorithm that uses both inter-node and intra-node bandwidth, while considering the memory copy overhead for large messages?**

In neighborhood collective communication, reducing the hop count is critical to improving efficiency, as it directly impacts latency and bandwidth utilization. By minimizing the number of hops, data transfer becomes faster, reducing congestion and optimizing bandwidth usage. This can be achieved by considering the underlying topology of the network to ensure that communication occurs along the shortest paths. This raises the question: **How can we reduce the number of hops messages traverse in neighborhood allgather?**

The increasing number of ranks per node participating in the neighborhood collective operations introduces another limitation for algorithms: memory consumption. The allocation of temporary buffers for communication can rapidly lead to substantial memory consumption, particularly for large messages. This can limit the scalability of neighborhood collective algorithms, as the memory requirements grow with the number of ranks per node. As a result, efficient memory management becomes crucial to maintain performance at scale. This brings up the question: **How can we devise an algorithm for neighborhood allgather considering the memory limitations to effectively manage the memory footprint?**

Performance modeling is one of the approaches for designing new algorithms and discovering issues in the application performance [12]. Performance modeling also facilitates comparisons with other algorithms, enabling integration with autotuning engines in MPI implementations and predicting how the algorithms perform on other clusters. These capabilities make performance modeling a valuable approach for both algorithm design and evaluation in diverse HPC environments. A key research question is: **How to construct an analytical model that accurately captures the communication behavior and performance characteristics of neighborhood allgather algorithms across varying virtual topologies and message sizes?**

III. Related Work

Hoefler and Träff [14] discuss sparse communication in MPI applications in a small neighborhood. Their work shows the advantages of neighborhood collectives for applications with sparse communication patterns. They introduced the neighborhood allgather, alltoall, and reduce operations, of which the first two were later added to the MPI standard. Collom et al. [15] introduce topology objects to support dynamic virtual topologies in MPI, enabling efficient communication for applications with changing patterns. They demonstrate how neighborhood collectives can replace point-to-point communication in such scenarios, improving scalability and performance.

Kumar et al. [16] introduce a collective interface for the Deep Computing Messaging Framework (DCMF) to implement neighborhood collective communications in MPI. DCMF is a messaging library developed for the Blue Gene/P machines. Kandalla et al. [17] used the Core-Direct offload feature in Mellanox ConnectX 2 InfiniBand network adapters to develop scalable designs for non-blocking neighborhood collective operations applied to graph algorithms in the CombBLAS library.

Ovcharenko et al. [18] develop a communication package on top of MPI to use the sparse communication pattern in the applications. Their work shows the sparsity in the communication pattern of applications and how this sparsity can be used to improve communication performance. Hoefler and Schneider [11] explore various strategies to enhance the performance of neighborhood collective operations. They introduce two innovative heuristics specifically designed for optimizing neighborhood alltoall and allgather operations. They leverage user-level constraints via the `MPI_Info` argument, such as predefined communication channels or specified message sizes, to enhance the performance of neighborhood collective operations. In contrast, our algorithm is not bound to any constraints.

Träff et al. [19], [20], [21], [22] extend the neighborhood collective interface to support isomorphic communication patterns. Their approach leverages message combining to optimize these patterns, but it does not apply to general graph-based neighborhoods.

Lübbe [23] establishes self-consistent performance guidelines for MPI neighborhood collective operations and introduces microbenchmarks as a methodology to evaluate these guidelines. This approach enables detailed insights into the performance of MPI libraries on various architectures.

Mirsadeghi et al. [7] and Ghazimirsaeed et al. [8] improve neighborhood allgather in graph virtual topologies by incorporating message combining in a group of $K \geq 2$ ranks. Their approach involves using the shared outgoing neighbors of the ranks in the groups. Each rank in the group takes responsibility for forwarding messages on behalf of the others to a subset of the neighbors. This method reduces the total number of message transmissions, enhancing communication efficiency and minimizing overhead. However, unlike our approach, their algorithm is optimized for small messages up to 4KB.

Ghazimirsaeed et al. [9] introduce a hierarchical algorithm to enhance the performance of neighborhood collectives with medium-sized and large messages in graph virtual topologies. Each rank sends a portion of its message to its outgoing neighbors on remote nodes. The ranks on these remote nodes then exchange data to gather the complete message. Collom et al. [24] introduce a leader-based and locality-aware algorithm for persistent neighborhood alltoallv.

The most recent work on neighborhood allgather is the Distance-halving neighborhood [10]. This method recursively divides the communicator into halves, selecting an agent from the opposite half at each step to offload outgoing neighbors, thereby reducing communication with distant ranks. In this paper, we perform a comparative analysis of our proposed approach against this algorithm and update their performance model to incorporate the Max-rate Model [12] for a more

accurate evaluation.

Khorasani et al. [25] propose an algorithm for GPU-aware neighborhood alltoallv, targeting both AMD and NVIDIA GPUs. Temuçin et al. [26] develop a mechanism for GPU-aware neighborhood allgather and allgatherv for AMD GPUs. Although GPU clusters are not the target of this research, our algorithm can be adapted for such clusters.

IV. Cascade Algorithm

This section introduces the proposed Cascade algorithm. This algorithm is a topology-aware solution designed to reduce communication time for neighborhood allgather. By restricting communication to the nearest nodes, the algorithm reduces traffic on the bisection links, the number of hops, and network congestion. Prior to presenting the details, we first provide an overview of the algorithm and define the terminology used in this paper.

A. Overview of the Cascade Algorithm

In the Cascade algorithm, each rank organizes the computing nodes into multiple paths based on the network topology and sorts the nodes in each path according to their distance from the current node. The messages are sent in these paths in multiple steps until all outgoing neighbors have received the message. Fig. 2 illustrates two steps of the Cascade algorithm in one path based on a virtual topology. First, nodes are organized based on their distance. In Step 0, all ranks send their message to one of their outgoing neighbors on the closest node in the path. For instance, ranks a and b send their messages to e, while e simultaneously forwards its own message to h. It skips f and g since they are not its outgoing neighbors. In Step 1, messages received in Step 0 are forwarded to the next nodes. This process is continued in the next steps, which are not shown in the figure. We define the following terms:

- **Upstream ranks**: The ranks that send messages to the current rank in the current step.
- **Downstream ranks**: The ranks that receive messages from the current rank. In Fig. 2b, a and b are upstream ranks of e, and e is the downstream rank of a and b.
- **Source ranks**: The original ranks whose data is being communicated in the current step. In Fig. 2c, Rank e sends data of a and b as a combined message to f. In this communication, we refer to a and b as source ranks.

The messages are forwarded through successive downstream ranks chosen on the closest node in each path. In this way, the message traverses a shorter path compared to the naïve algorithm, because the downstream rank is closer to the next outgoing neighbors compared to the current rank. This also reduces traffic over the bisection links. The downstream rank also delivers the message to other receiver ranks on its node, and we choose at most one downstream rank on a node. This approach reduces the inter-node communication and results in less congestion.

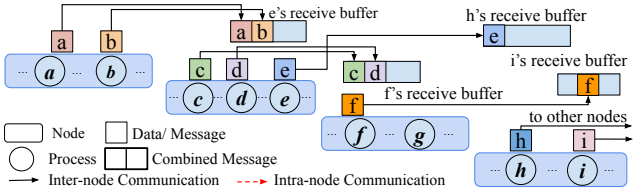
In each step, a rank selects its downstream ranks among the outgoing neighbors of the source ranks whose message is

Rank	Incoming Neighbors	Outgoing Neighbors
a	-	*c, e , f*
b	-	*e, g*
c	*a*	*f, g, h*

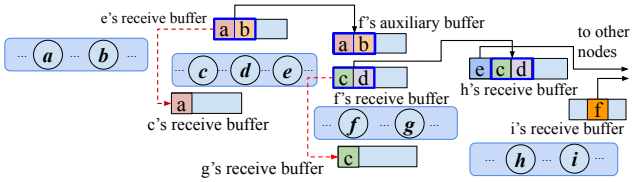
Rank	Incoming Neighbors	Outgoing Neighbors
d	-	*f, h*
e	*a, b*	*h*
f	*a, c, d*	*i*

Rank	Incoming Neighbors	Outgoing Neighbors
g	*b, c*	-
h	*e, c, d*	-
i	*f*	-

(a) Partial Virtual Topology



(b) Step 0: All ranks send their data to their downstream ranks in the closest node in the path and receive messages from upstream ranks.



(c) Step 1: The messages received in Step 0 are forwarded to the next downstream ranks on the next nodes in the Path. Some messages are combined.

Fig. 2: Illustration of two steps of the Cascade algorithm in one path with 4 nodes, based on a virtual topology

received in the previous step. For example, consider rank e in Fig. 2. In Step 0, e sends its own message to h. In the next step, e forwards the messages it received from a and b to f, since f is an outgoing neighbor of both a and b. So each rank sends its message to its downstream in Step 0, and in the following steps, it works as the downstream/upstream rank to forward the messages of other ranks.

B. Message Combining and Buffers

In the Cascade algorithm, messages can be either regular or combined. The data originating from a rank's send buffer is known as a primary message. When multiple primary messages are aggregated, they form what we refer to as combined messages. In Fig. 2, ranks c and d send their messages to f in Step 0, which are then forwarded to h in the next step as a combined message. Message combining reduces the number of messages and improves communication performance. The selection of downstream ranks plays a key role in determining how many messages are combined. We propose a distributed algorithm for selecting downstream ranks to increase the number of combined messages. In this algorithm, the downstream ranks are not limited to the outgoing neighbors of the source ranks; they can also be selected among other ranks.

At each step, if the incoming message contains only data from the incoming neighbors, it is received in the receive buffer. Otherwise, the message is directed to a temporary buffer known as the *auxiliary buffer*. In Fig. 2c, f receives the combined message from e in an auxiliary buffer, because

the combined message includes the data of b, which is not an incoming neighbor of f. In this step, h receives a combined message from f, including the data of source ranks c and d. However, in this case, the message is directly received in h's receive buffer, because c and d are incoming neighbors of h. A *double buffering* technique is used, utilizing one buffer for incoming messages and another for outgoing messages. As the algorithm progresses through its steps, it alternates between the two buffers to communicate with upstream and downstream ranks simultaneously, utilizing the available bandwidth.

Cascade considers memory footprint, reducing the use of auxiliary buffers by allocating limited memory resources and reusing them during neighborhood collective communication. Although our algorithm supports message combining, it does not accumulate the messages of multiple steps in auxiliary buffers. Additionally, MPI derived datatypes are utilized to leverage the scatter-gather capabilities of InfiniBand networks, supported by the low-level libraries such as Unified Communication X (UCX) [27], to handle non-contiguous buffers efficiently. This method minimizes memory copying and enhances data transfer performance. The algorithm utilizes message combining to decrease the total number of messages while maximizing the data transferred at each step.

The rest of this section presents the Cascade algorithm, which comprises two parts: communication pattern creation routines and neighborhood operations. A communication pattern is an object that defines how messages are sent and received during neighborhood allgather operations. This object is created when the virtual topology is established and is attached to the communicator along with the virtual topology. The communication pattern is used every time a neighborhood operation is called. The communication pattern is created once and reused multiple times.

C. Notations

Table I provides a summary of the notations in the rest of this paper. The rank of the current process is denoted by p, with its incoming and outgoing neighbors represented by I and O, respectively. The incoming and outgoing neighbors of another rank, say rank x, are denoted by I_x and O_x, respectively. When the current rank receives the message of x, either directly or indirectly, it must deliver that message to a subset of O_x, which is shown by O'_x. This subset can be as large as O_x itself ($O'_x \subseteq O_x$). The sizes of I and O are defined by $indegree$ and $outdegree$, respectively.

The size of the communicator, which corresponds to the total number of ranks, is shown by n. Parameter C represents the average number of ranks per node. The size of the primary messages is shown by m. As messages may be combined, the size of an inter-node/intra-node message is shown by m_{inter} or m_{intra}. The number of such messages is shown by n_{inter} and n_{intra}. The variable U_C refers to the set of upstream candidates, with U being a subset of these candidates. Ds denotes the downstream ranks.

The incoming source ranks, denoted by S_i, show the source of the data received from U. The list of outgoing source ranks is shown by S_o, representing the source of the data sent to Ds. In the first step, all ranks send their own data, which means $S_i^{(0)} = U^{(0)}$ if $I \neq \phi$ and $S_o^{(0)} = \{p\}$ if $O \neq \phi$. However, in the subsequent steps, these sets are disjoint and we have: $S_i^{(t)} \cap U^{(t)} = \phi$ and $p \notin S_o^{(t)}, \forall t > 0$. The superscript (t) indicates the parameter's value in Step t. The data of ranks in $S_i^{(t)}$ is received in Step t and is forwarded to the downstream ranks (if any) in Step $t+1$. In other words, $S_o^{(t+1)}$ is equivalent to $S_i^{(t)}$ for all ranks.

TABLE I: Description of notations

Notation	Description
p	rank of current process
I	incoming neighbors of the current rank
O	outgoing neighbors of the current rank
I_x	incoming neighbors of rank x
O_x	outgoing neighbors of rank x
$O'_x \subseteq O_x$	outgoing neighbors of x, that p is aware of
$indegree$	size of I
$outdegree$	size of O
n	size of the communicator ($comm$)
C	the average number of ranks per node
m	size of message in $sendbuf$
m_{inter}	size of inter-node messages
m_{intra}	size of intra-node messages
n_{inter}	number of inter-node messages
n_{intra}	number of intra-node messages
U_c	upstream candidates
$U \subset U_c$	(confirmed) upstream ranks
Ds	downstream ranks
S_o	outgoing source ranks (sent to Ds)
S_i	incoming source ranks (received from U)
$S_{o/inter}$	ranks in S_o whose data is sent to remote nodes
$S_{o/intra}$	ranks in S_o whose data is sent to ranks on local node
$S_{i/inter}$	ranks in S_i whose data is received from remote nodes
$S_{i/intra}$	ranks in S_i whose data is received from the local node
$0 \leq \delta \leq 1$	density of random sparse graphs
$P(.)$	probability mass/density functions (PMF/PDF)
$\mid . \mid$	size function
$R[\,][\,]$	paths
R_1	number of paths
R_2	average number of nodes per path
α	fixed latency for each message in the Max-rate Model [12]
R_N	injection bandwidth in the Max-rate Model
R_C	maximum communication rate of a rank in the Max-rate Model
N_o	number of outgoing nodes in a path

The parameter δ is the density of random sparse graphs, introduced in Section V. We use $P(.)$ to show the probability of a variable getting a value in the random sparse graphs, and $|.|$ shows the size of different variables. $R[\,][\,]$ represents the paths, including nodes. Each path includes multiple nodes, sorted in ascending order based on their distance to the current node. R_1 shows the number of paths, and R_2 shows the average number of nodes per path across all ranks. α, R_N and R_C are the parameters of the Max-rate Model, and N_o is the number of outgoing nodes, all introduced in Section V.

D. Building the Communication Pattern

The main routine for building the communication pattern is the *build_cp* function shown in Algorithm 1. This function is called inside `MPI_Dist_graph_create_adjacent`. The inputs I, O, $indegree$, and $outdegree$ are equivalent to $sources$, $destinations$, $indegree$, and $outdegree$ in the

signature of the mentioned MPI operation. The parameters n and p are extracted from the communicator. The $build\_cp$ function takes these inputs and returns the communication pattern as output.

The variables are initialized in Lines 4–7. Since incoming neighbors in I may select the current rank as their downstream in the first step, U_c is initialized to I in Line 5. As the source ranks are still unknown, both S_o and S_i are set to ϕ. The variable t indicates the current step number.

Each iteration of the while loop, spanning Lines 8-21, adds a new step to the communication pattern. In the first step, all ranks send their $sendbuf$ to their downstream; therefore, in the first step, the current rank is the only source rank, added to the S_o in Step 0, shown in Line 11. In the following steps, the incoming source ranks of the previous step, S_i, are copied to S_o (Line 13). The value of S_i comes from the previous iteration of the loop and includes the source ranks whose data is received in the previous step and is going to be sent out to other ranks in the current step, if they have some outgoing neighbors left.

The downstream ranks are determined by the $find\_downstreams$ function, which is shown in Algorithm 2. This function is invoked in Line 16 of $build\_cp$. Once the downstream ranks are identified, the $exchange\_notifications$ function is called. This function performs two main operations: (1) It notifies the downstream ranks of their role, providing them with a list of source ranks and their remaining outgoing neighbors, which they add to their S_i and O'. The current rank is a new upstream for those ranks; therefore, they also add the rank of the current process to their upstream ranks in $step.U$ variable; and (2) It notifies the outgoing neighbors of the ranks in S_i about the newly selected downstream ranks, asking them to update their candidate upstream ranks, U_c. The current rank also receives the notification from U_c and updates its U_c and other variables. This gathered information is saved in the communication pattern in Lines 18-19. We continue iterating in the loop, adding new steps, as long as there are remaining candidate upstream ranks or outgoing neighbors for the outgoing source ranks S_o.

Algorithm 2 presents the $find\_downstreams$ function. This algorithm is designed to identify the downstream ranks for communication. It begins by initializing a priority array. In the nested loops of Lines 4-15, we iterate over the ranks in S_o and their outgoing neighbors. If $o \in O'_s$ is on the local node, its message is directly delivered to it. So it is added as a downstream rank to $step$. Otherwise, we set the number of messages it should receive as its priority. The ranks on the nearest nodes with a greater number of incoming neighbors in S_o have a higher priority. This prioritization helps in selecting which ranks to communicate with first, resulting in fewer hops traversed in the network and a greater chance of message combining. In the while loop, starting at Line 16, the algorithm iterates over the outgoing neighbors of $s \in S_o$ and selects the downstream rank with the highest priority to forward data of s for a path. Once a downstream rank is chosen for s, the algorithm checks if any other ranks have their closest outgoing neighbors located on the same node as the selected downstream rank. If such ranks are found, the selected downstream rank is assigned as the downstream for them. This strategy helps reduce communication overhead by enabling message combining and limiting the number of messages sent to each remote node to a single message. In Line 20, the communication path is marked as covered once

Algorithm 1: Building the Communication Pattern

```
1  Input: comm, I, indegree, O, outdegree, p, n
2  Output: communication_pattern cp
3  Function build_cp:
4      communication_pattern cp
5      U_c ← I
6      S_o, S_i ← φ
7      t ← 0
8      while (U_c ≠ φ or ∃ s ∈ S_i | O'_s ≠ φ) do
9          step← new_step(t)
10         if (t = 0 and outdegree > 0) then
11             S_o ← {p}
12         else if (t > 0) then
13             S_o ← S_i
14             S_i ← φ
15         end
16         find_downstreams(comm, step, S_o)
17         exchange_notifications(comm, step, S_i, S_o, U_c)
18         add S_i and S_o to step
19         add step to cp.steps
20         t++
21     end
22     return cp
23 end
```

Algorithm 2: Finding Downstream Ranks

```
1  Input: comm, step, S_o
2  Function find_downstreams:
3      priority[] = {0, 0, ... , 0}
4      foreach (s ∈ S_o) do
5          foreach (o in O'_s) do
6              if o is on the local node then
7                  add s to step.list[o]
8                  if o ∉ step.Ds then
9                      add o to step.Ds
10                 end
11             else if (o is on the nearest node among nodes(O'_s))
                   then
12                 priority[o]++
13             end
14         end
15     end
16     while all paths for all ranks in S_o not covered do
17         d ← max_priority_item(priority) // downstream
18         foreach (s ∈ S_o | d ∈ O'_s and d is on a remote node )
             do
19             add s to step.list[d]
20             mark path(d) as covered for s
21         end
22         add d to step.Ds
23         priority[d] ← 0
24     end
25 end
```

the rank is selected, ensuring that all necessary communication paths are processed. The rank repeats until all communication paths for all ranks in S_o are covered.

E. Neighborhood Operation

Algorithm 3 shows the neighborhood operation. Every time the `MPI_Neighbor_allgather` is called in the application, this function is invoked to perform the communication. In the first step of the algorithm in Line 3, we retrieve the communication pattern from the communicator. Each iteration of the foreach loop spanning Lines 4-29 corresponds to one step of the communication pattern. Lines 6 and 7 initiate the double buffering procedure and show how switching between the two auxiliary buffers is performed. In each step, the buffer $raux$ (receive auxiliary buffer) is used for receiving messages of $step.S_i$. The buffer $saux$ (send auxiliary buffer) includes the data of $step.S_o$, received in the previous step, and is sent to downstream ranks in the current step.

In Lines 9-11, the algorithm posts the receive requests from upstream ranks. Subsequently, in Lines 13-19, communication with downstream ranks is performed. Following these operations, an `MPI_Waitall` function is invoked in Line 20 to ensure the completion of all communications. Once the communication phase is complete, the received messages are transferred from $raux$ to `recvbuf`, provided that an incoming neighbor exists within the set $step.S_i$ and its data has been received in $raux$.

Algorithm 3: `MPI_Neighbor_Allgather`

```
1  Input: sendbuf,sendcount, sendtype,
          recvbuf,recvcount, recvtype, comm
2  Function MPI_Neighbor_allgather_cascade:
3      communication_pattern cp ← comm.cp
4      foreach (step ∈ cp.steps) do
5          \\swap auxiliary buffers
6          raux ← (step.t % 2 = 0) ? cp.aux0: cp.aux1
7          saux ← (step.t % 2 = 0) ? cp.aux1: cp.aux0
8          \\post receive from upstream ranks
9          foreach (Upstream u in step.U) do
10             irecv from u in raux or recvbuf
11         end
12         \\post send to downstream ranks
13         foreach (Downstream d in step.Ds) do
14             if (step.t = 0) then
15                 isend sendbuf to d
16             else
17                 isend data of step.list[d] to d from recvbuf or
                     saux
18             end
19         end
20         MPI_Waitall
21         foreach (s in step.S_i) do
22             if (s ∈ I and data of s received in raux) then
23                 copy data of s to recvbuf
24             end
25             if (s received in sendbuf and combined with other
                 messages in auxiliary buffers in next step ) then
26                 copy data of s to raux
27             end
28         end
29     end
30 end
```

V. Analytical Modeling

This section presents the analytical models developed for the Cascade, Distance-halving neighborhood [10], and the naïve Open MPI algorithms. These models are built based on a virtual topology generated using the Erdős–Rényi graph generation model [28]. In this approach, a random sparse graph $G(V, E)$ is generated using $|E|$ Bernoulli trials with parameter $0 \leq \delta \leq 1$. The set V represents the vertices corresponding to ranks, and E denotes the directed edges between them. δ represent the probability of each directed edge in E being present in the graph.

We use the Max-rate Model [12] that defines the inter-node communication time as $\alpha + km/min(kR_C, R_N)$, where α represents the message latency, R_C denotes the maximum communication rate for one rank, and R_N is the injection bandwidth of a node. If the ranks do not constrain each other, they can communicate at their maximum rate R_C. However, as the number of ranks (k) increases, contention arises, and the total communication rate becomes restricted by the node injection bandwidth R_N. We utilize this model due to its high accuracy and its ability to capture contention.

Although designed for inter-node communication, this model also applies to intra-node communication due to similar message passing and contention effects. We measured the parameters for intra-node and inter-node communication separately to account for their differences. $\{\alpha^{inter}, R_C^{inter}, R_N^{inter}\}$ and $\{\alpha^{intra}, R_C^{intra}, R_N^{intra}\}$ refer to values for inter-node and intra-node values, respectively. In the rest of this paper, we use CS and DH as the short forms of the Cascade and the Distance-halving algorithms, respectively.

A. Volume of Messages in the Cascade Algorithm

In this section, we find the expected volume of the messages in Step t. Given the fact that the probability of a rank being outgoing neighbor of another rank is equal to δ and there are an average of C ranks per node, the probability that a rank has no outgoing neighbors on a remote node is given by $(1-\delta)^C$; therefore, the probability that it has at least one outgoing neighbor on a remote node is $1-(1-\delta)^C$. An outgoing node of a rank is a remote node where the rank has at least one outgoing neighbor on the node. The probability of having j outgoing nodes among R_2 nodes is:

$$P(N_o = j) = \binom{R_2}{j}(1-\delta)^{C(R_2-j)}(1-(1-\delta)^C)^j \qquad (1)$$

In the first step, the data of ranks with at least one outgoing node is sent. In the second step, the data of ranks with at least two outgoing nodes is sent. More generally, in Step t, the data of ranks with at least $t+1$ outgoing nodes is sent. Considering this, the expected value of the total volume of inter-node messages sent by each rank in Step t in R_1 paths

is calculated as:

$$\mathbb{E}(m_{inter}^{(t)}(CS)) = mR_1 \sum_{j=t+1}^{R_2} P(N_o = j) \tag{2}$$

For intra-node messages, the number of receiver ranks for $S_{o/intra}^{(t)}$ messages is like the binomial distribution of $B(|S_{o/intra}^{(t)}|, \delta)$, and the expected size of the message is equal to:

$$\mathbb{E}(m_{in}^{(t)}(CS)) = \delta m |S_{o/intra}^{(t)}| \tag{3}$$

B. Number of Messages in the Cascade Algorithm

A rank receives the data of $|S_{o/inter}^{(t)}|$ source ranks in Step $t-1$ and forwards this data to the downstream ranks in Step t, either as a combined or a regular message. The expected value of the number of outgoing source ranks whose data is sent to other nodes in Step t is calculated as:

$$\mathbb{E}(|S_{o/inter}^{(t)}|) = \frac{\mathbb{E}(m_{inter}^{(t)}(CS))}{m} \tag{4}$$

One of the ranks on the j-th node in a path is selected as a downstream rank when there is no receiver rank on the $j-1$ closer nodes and at least one receiver rank on Node j. The probability of a rank on Node j being selected as a downstream rank for rank $s \in S_{o/inter}^{(t)}$ is equal to:

$$P(Ds[s] \text{ on } j) = (1-\delta)^{jC}(1-(1-\delta)^{C}) \tag{5}$$

We have data of outgoing source ranks $S_{o/inter}^{(t)}$ to decide for, over R_2 nodes in each path. Since we combine the messages sent to each node, each rank sends at most one message to a remote node, either combined or just as a regular message. Y_j shows the number of primary messages sent to Node j, among $|S_{o/inter}^{(t)}|$ primary messages. We calculate the probability of sending no message to j as:

$$\begin{aligned} P(Y_j = 0) &= P\left(Ds[s] \text{ not on } j, \forall s \in S_{o/inter}^{(t)}\right) \\ &= (1 - P(Ds[s] \text{ on } j))^{|S_{o/inter}^{(t)}|} \end{aligned} \tag{6}$$

We calculate the expected number of messages sent to j:

$$\begin{aligned} \mathbb{E}(n_{inter}^{(t)}(CS, j)) &= 1 \times P(Y_1 = 1) + 0 \times P(Y_j = 0) \\ &= P(Y_j = 1) = 1 - P(Y_j = 0) \end{aligned} \tag{7}$$

Given that, we can find the expected number of messages sent to all nodes in a path:

$$\begin{aligned} \mathbb{E}\left(n_{inter}^{(t)}(CS, *)\right) &= \mathbb{E}(\sum_{j=0}^{R_2-1} n_{inter}^{(t)}(CS, j)) \\ &= \sum_{j=0}^{R_2-1} \left(\mathbb{E}(n_{inter}^{(t)}(CS, j))\right) \end{aligned} \tag{8}$$

In the first step, ranks send their own data in all paths. In the next steps, the ranks forward messages entered from a path in the previous step to other $R_1 - 1$ paths. Given this, the expected number of actual messages sent to other nodes from a rank in Step t is calculated as follows:

$$\mathbb{E}(n_{inter}^{(t)}(CS)) = \begin{cases} R_1 \mathbb{E}\left(n_{inter}^{(t)}(CS, *)\right) & t = 0 \\ (R_1 - 1)\mathbb{E}\left(n_{inter}^{(t)}(CS, *)\right) & t > 0 \end{cases} \tag{9}$$

For intra-node messages, considering $\mathbb{E}(|S_{o/inter}^{(t)}|)$ primary messages, the probability of sending a message to each rank on the same node is equal to the probability of sending at least one of these messages to them which is calculated as $1-(1-\delta)^{\mathbb{E}(|S_{o/inter}^{(t)}|)}$. The expected number of intra-node messages each rank sends is:

$$\mathbb{E}(n_{intra}(CS)) = C(1-(1-\delta)^{\mathbb{E}(|S_{o/inter}^{(t)}|)}) \tag{10}$$

C. Constructing the Model

By substituting the size of the messages in the Max-rate Model, we get:

$$\mathbb{E}(t_{inter/1}^{(t)}(CS)) = \alpha + \frac{C\mathbb{E}(m_{inter}^{(t)}(CS))}{min(CR_C^{inter}, R_N^{inter})} \tag{11}$$

This is the expected communication time for one message from each rank; however, we know in this step, each rank on average sends $\mathbb{E}(n_{inter}^{(t)}(CS))$ messages and receives the equal number of messages, so the expected total communication time for each step is calculated as:

$$\mathbb{E}(t_{inter}^{(t)}(CS)) = 2\mathbb{E}(n_{inter}^{(t)}(CS))\mathbb{E}(t_{inter/1}^{(t)}(CS)) \tag{12}$$

We have a maximum of R_2 steps, so the total inter-node communication time in the Cascade algorithm is equal to:

$$\mathbb{E}(t_{inter}(CS)) = \sum_{t=0}^{R_2-1} \mathbb{E}(t_{inter}^{(t)}(CS)) \tag{13}$$

Similarly, for intra-node messages in Step t, the communication time is:

$$\begin{aligned} \mathbb{E}(t_{intra}^{(t)}(CS)) =& \mathbb{E}(n_{intra}(CS)) \times \\ & (\alpha^{intra} + \frac{C\mathbb{E}(m_{intra}^{(t)}(CS))}{min(CR_C^{intra}, R_N^{intra})}) \end{aligned} \tag{14}$$

And for all steps:

$$\mathbb{E}(t_{intra}(CS)) = \sum_{t=0}^{R_2-1} \mathbb{E}(t_{intra}^{(t)}(CS)) \tag{15}$$

The total communication time is equal to inter-node communication time plus intra-node communication time:

$$\mathbb{E}(t_{total}(CS)) = \mathbb{E}(t_{inter}(CS)) + \mathbb{E}(t_{intra}(CS)) \tag{16}$$

D. Modeling the Naïve and Distance-halving algorithms

From each rank's perspective in the naïve algorithm, there are $n - C$ ranks on other nodes, and on average $(n - C)\delta$ of them are the outgoing neighbors of the current rank, receiving a message with size m. So we have:

$$\mathbb{E}(t_{inter}(\textit{naïve})) = (n-C)\delta(\alpha^{inter} + \frac{Cm}{min(R_C^{inter}, CR_N^{inter})}) \tag{17}$$

Similarly, there are $C\delta$ intra-node outgoing neighbors per node for each rank and the expected intra-node communication is:

$$\mathbb{E}(t_{intra}(\textit{naïve})) = C\delta(\alpha^{intra} + \frac{Cm}{min(CR_C^{intra}, R_N^{intra})}) \tag{18}$$

For the Distance-halving neighborhood, we use the Max-rate Model instead of the Hockney Model in [10]. The number of halving steps, which is equal to the number of inter-socket messages, is:

$$\mathbb{E}(n_h(\textit{DH})) = min\left(\lceil log(\frac{n}{L})\rceil + 1, \delta(n - L)\right) \tag{19}$$

The size of messages in Step t is equal to $2^t m$. The expected communication time of the halving phase is equal to:

$$\mathbb{E}(t_h(\text{DH})) = \sum_{t=0}^{\mathbb{E}(n_h(DH))-1} (\alpha + \frac{2^t Cm}{min(CR_C, R_N)}) \tag{20}$$

Each node contains $\frac{C}{L}$ sockets, and the final $\log \frac{C}{L}$ halving steps take place within the nodes, between sockets. Accordingly, the parameters α, R_C, and R_N are set to intra-node or inter-node values based on whether Step t involves intra-node or inter-node communication. The expected number of intra-socket messages is:

$$\mathbb{E}(n_{intra}(\text{DH})) = \left(1 - (1-\delta)^{\lceil log(\frac{n}{L})\rceil + 2}\right) L \tag{21}$$

The expected intra-socket communication time is:

$$\begin{aligned}\mathbb{E}(t_{intra}(\text{DH})) =& \mathbb{E}(n_{intra}(\text{DH}))\times \\ & (\alpha^{intra} + \frac{\delta\mathbb{E}(n_{intra}(\text{DH}))m}{min(CR_C^{intra}, R_N^{intra})})\end{aligned} \tag{22}$$

The expected total communication time is equal to $\mathbb{E}(t_h(\text{DH})) + \mathbb{E}(t_{intra}(\text{DH}))$.

VI. Performance Evaluation and Analysis

This section presents the performance evaluation and overhead analysis of the Cascade algorithm, using the random sparse graphs and Moore neighborhood microbenchmarks and a sparse matrix-matrix multiplication kernel, compared against the default Open MPI implementation and the Distance-halving neighborhood algorithm [10]. We also conducted experiments with the Common-Neighbor algorithm [7], [8], which is optimized for small messages ($\leq$ 4KB). In contrast, our proposed algorithm targets medium and larger messages ($\geq$ 4KB). Our algorithm is faster in the target range as expected, and we omit the results of the Common Neighbor algorithm for brevity. The implementations of other neighborhood allgather algorithms in the literature are not public.

A. Experimental Setup

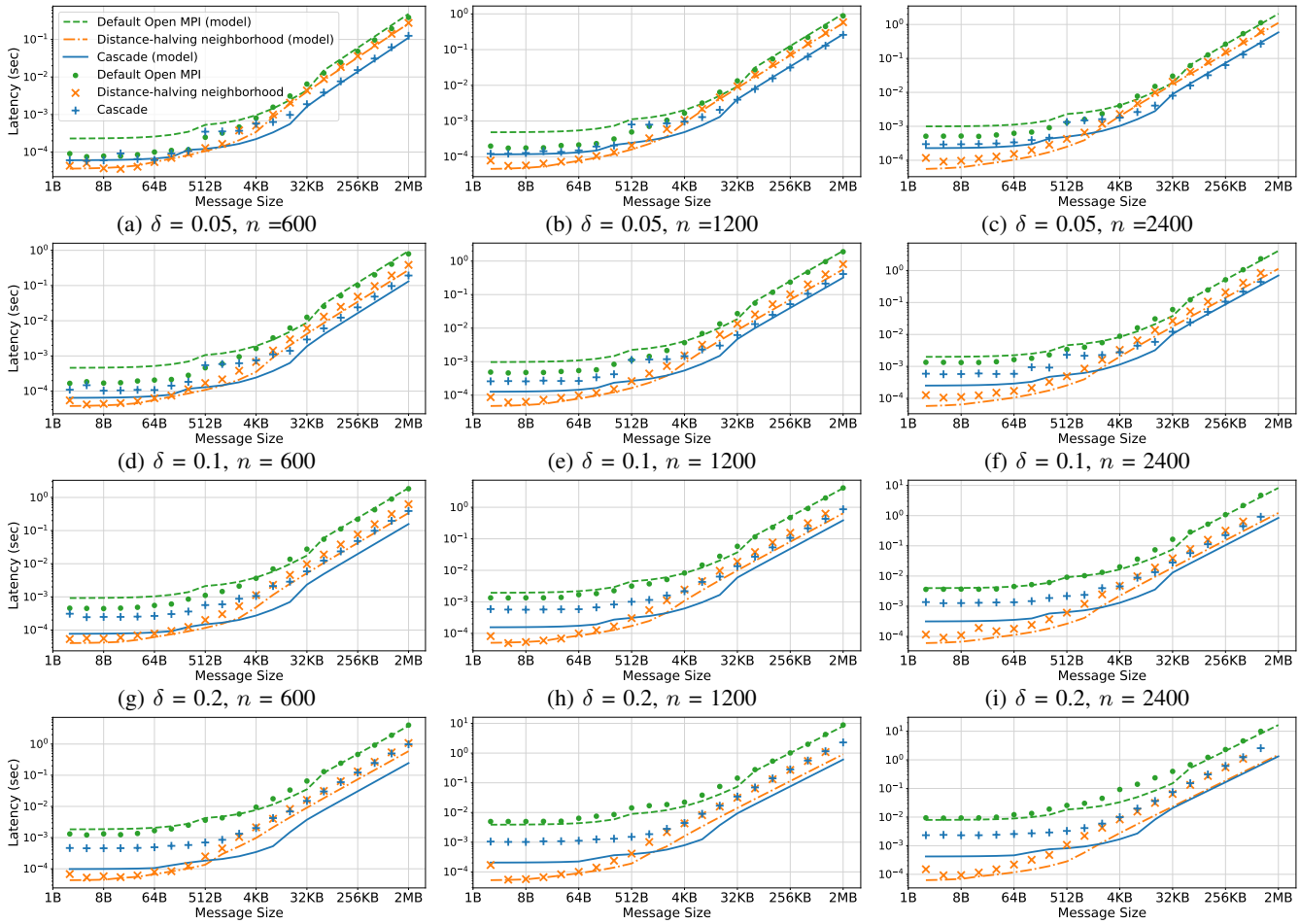
We implemented our algorithm in Open MPI v5.0.0rc12, running on top of UCX v1.17.0. We conducted our experiments on the Narval cluster in the Digital Research Alliance of Canada. The CPU partition of Narval consists of 1181 nodes, each equipped with two AMD EPYC 7532 or two AMD EPYC 7502 processors, offering a total of 64 cores per node. Each node of Narval has at least 249 GB of memory. The nodes are connected with InfiniBand HDR in a custom network topology. We allocated 60 cores per node to the jobs, reserving 4 cores for the OS. We used the OSU microbenchmarks v7.2 for random sparse graphs, Moore neighborhoods, and overhead analysis. Each experiment is conducted a minimum of three times, and we report the average after removing outliers.

B. Random Sparse Graphs

We used the Erdős–Rényi model to generate random sparse graphs of various sizes and densities. Fig. 3 shows the latency of the proposed algorithm compared to the Distance-halving neighborhood and the default Open MPI for random sparse graphs. We used 600, 1200, and 2400 ranks over 10, 20, and 40 nodes, respectively, to illustrate the scalability of the algorithm. In each case, we compared the latency of the proposed algorithm to other algorithms with multiple graphs with densities ranging from 0.05 to 0.4. Experimental results are shown by discrete data points, and the lines illustrate the modeling results.

To measure the parameters of the Max-Rate Model on Narval, the procedure outlined by the authors in [12] was followed. Initially, their provided tool was used to evaluate the point-to-point communication time among multiple ranks communicating concurrently across different nodes. We then used a Python script to fit the Max-rate curve for each point-to-point protocol using mean squared error. We developed a similar tool for intra-node communication and repeated the procedure to find the intra-node parameters.

Our algorithm, in almost all cases, outperforms the default Open MPI algorithm, regardless of the message size or the sparsity of the graph. The proposed algorithm demonstrates superior performance compared to the Distance-halving algorithm for message sizes of 4KB and larger in small densities. With $\delta = 0.4$, the latency of the Distance-halving is comparable to that of the Cascade algorithm for 4KB messages and above. However, when we consider the memory requirements associated with the Distance-halving neighborhood, the proposed algorithm stands out as the superior choice. The memory demand associated with the Distance-halving neighborhood accounts for the absence of data points observed for larger message sizes at higher densities, such as 1MB in Fig. 3i. The subtle fluctuations occur due to the transition between the short, eager, and rendezvous point-to-point protocols at the UCX level. Overall, our algorithm shows from 0.61x to 9.54x speedup over default Open MPI for random sparse graphs, with an average of 3.36x. For the target range of medium and large messages (4KB and larger), the average speedup for the proposed algorithm is 4.10x while the Distance-halving



(a) δ = 0.05, n =600 (b) δ = 0.05, n =1200 (c) δ = 0.05, n =2400

(d) δ = 0.1, n = 600 (e) δ = 0.1, n = 1200 (f) δ = 0.1, n = 2400

(g) δ = 0.2, n = 600 (h) δ = 0.2, n = 1200 (i) δ = 0.2, n = 2400

(j) δ = 0.4, n = 600 (k) δ = 0.4, n = 1200 (l) δ = 0.4, n = 2400

Fig. 3: The latency of the Cascade algorithm compared to the Distance-halving neighborhood and the default Open MPI for Random Sparse graphs with various densities (δ) and message sizes with 600, 1200, and 2400 ranks on Narval

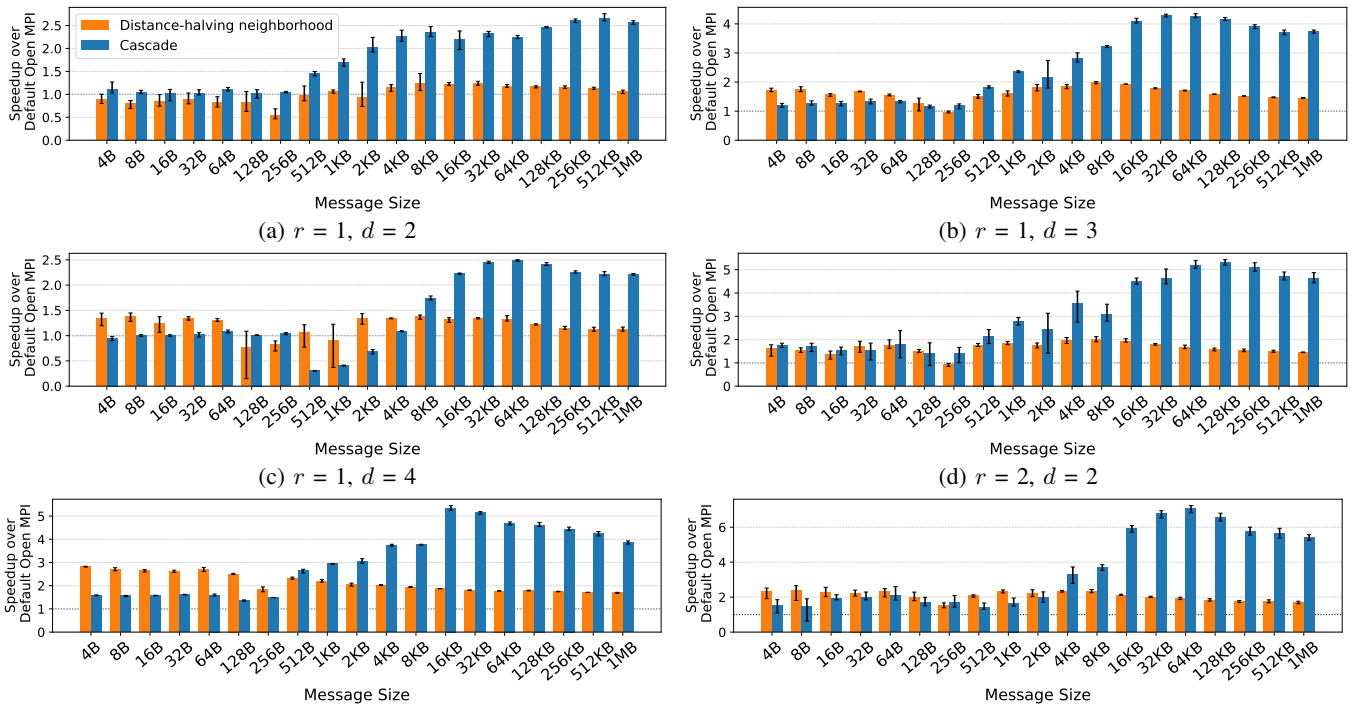
algorithm reaches 2.88x speedup on average, which shows the Cascade algorithm is 40% faster than the Distance-halving neighborhood for the target range.

Our models predict that the proposed algorithm outperforms the naïve algorithm for all message sizes and works better than the Distance-halving neighborhood for messages equal to or larger than 4KB. The experimental results closely follow the trends of modeling. This demonstrates our models can accurately reveal the performance of all three algorithms, confirming the correctness of our models. Regardless of the errors in the absolute values in some cases, our model can effectively uncover the trends in the performance of all three algorithms. It can identify critical message sizes where one algorithm begins to outperform another, serving as thresholds for switching between various algorithms in auto-tuning.

C. Moore Neighborhoods

Moore neighborhoods are one of the important stencil communication patterns in MPI applications. In Moore neighborhoods, ranks are placed in a d-dimensional grid and communicate with $(2r+1)^d - 1$ ranks at a Chebyshev distance equal to or smaller than r.

Fig. 4 compares the speedup of the proposed algorithm against the Distance-halving neighborhood over the default Open MPI with 2400 ranks. Regardless of a few cases, Cascade consistently outperforms the other algorithms for messages exceeding 4KB. In contrast, the Distance-halving algorithm performs better for messages smaller than 4KB in most cases. Notably, the Cascade algorithm also demonstrates better performance for small messages in small Moore neighborhoods like Moore ($r = 1$, $d = 2$). The best speedup of our algorithm occurs with 64KB messages in the Moore configuration ($r = 3$, $d = 2$). In this scenario, the Distance-halving neighborhood outperforms the default Open MPI with around a 2x performance boost. At the same time, our algorithm achieved 3.5x performance improvement against the Distance-halving neighborhood and 7.05x against the default Open MPI. Overall, our algorithm outperforms the default Open MPI algorithm with a 2.54x speedup in Moore neighborhoods. For the target range of 4KB and larger, our algorithm delivers greater efficiency as expected, reaching average speedups of 3.81x over Open MPI and 2.38x over the Distance-halving neighborhood.



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

(c) $r = 1$, $d = 4$ (d) $r = 2$, $d = 2$

(e) $r = 2$, $d = 3$ (f) $r = 3$, $d = 2$

Fig. 4: Speedup of the Cascade algorithm compared to the Distance-halving neighborhood algorithm over the default Open MPI, evaluating various Moore neighborhoods across different message sizes, using 2400 ranks on 40 nodes on Narval

D. Sparse Matrix-Matrix Multiplication

Sparse matrix-matrix multiplication is a critical operation in many scientific and deep learning applications. This section presents the results of a sparse matrix-matrix multiplication kernel that computes the product of two matrices, $X[i \times j]$ and $Y[j \times k]$, in parallel, resulting in the matrix $Z[i \times k]$ such that $Z = X \times Y$. The rows of matrices X and Y are partitioned into n strips, which are then distributed across the ranks. `MPI_Neighbor_allgather` is used to collect the columns of Y. After the computation completes, each rank retains a strip of rows from the resulting matrix Z. The virtual topology is created based on the non-zero blocks of X.

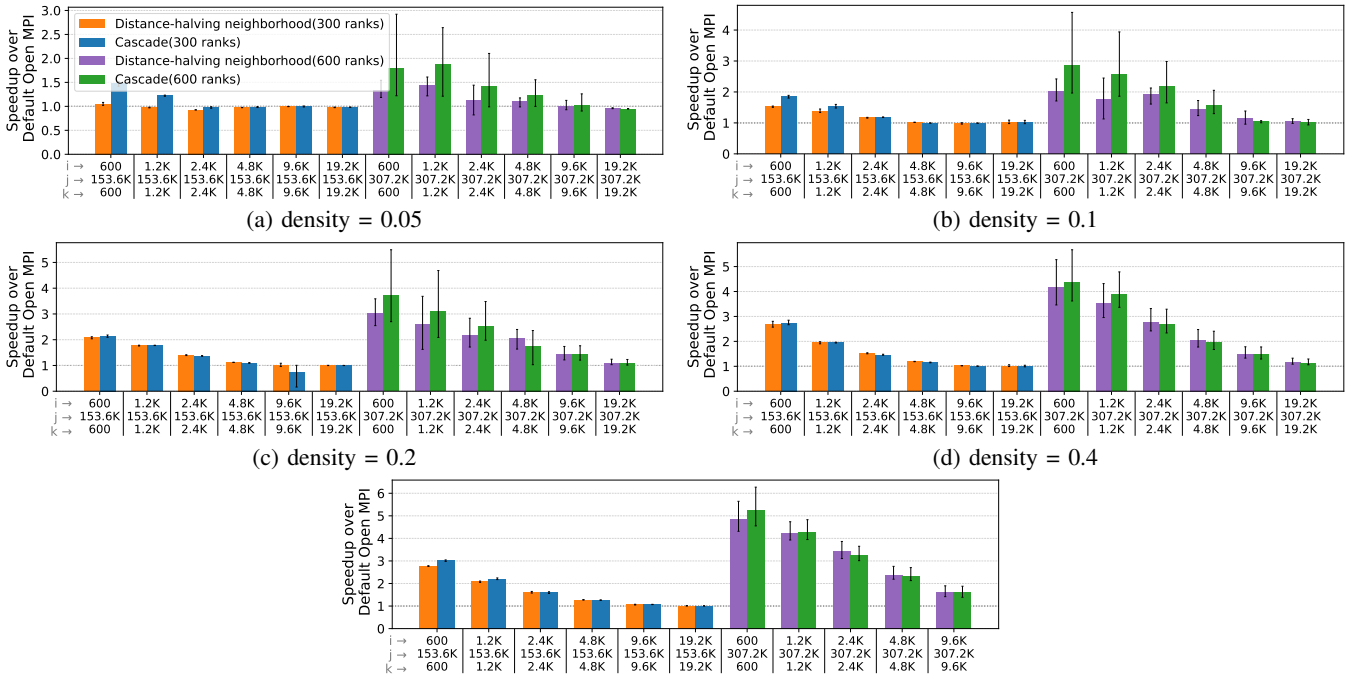
We measured the speedup of the algorithms for random matrices with various sizes and levels of sparsity levels. Since our algorithm is designed for medium and large messages, we should use very large matrices. Since we could not fit very large square matrices in the memory of our system, we chose smaller i's and k's. The size of the messages depends on j, so we selected large values for j.

As illustrated in Fig. 5, the Cascade algorithm demonstrates improvements in communication speed. As the size of the matrices grows, the speedup of both algorithms diminishes. This decline is due to the decreasing communication-to-computation ratio that occurs with larger matrices. As the density of the matrices increases, our algorithm demonstrates significant speedup improvements, indicating its ability to effectively address challenges that the naïve algorithm encounters. Our algorithm achieves a speedup ranging from 0.71x to 5.25x for the sparse matrix multiplication kernel, with an average speedup of 1.84x. In the Distance-halving neighborhood, the values range from 0.92x to 4.86x, with an average of 1.73x.

E. Overhead Analysis

Fig. 6 compares the virtual topology creation overhead of the proposed algorithm with the Distance-halving neighborhood in random sparse graphs. As expected, smaller communications and sparser virtual topologies exhibit lower latency. This overhead is incurred each time a new virtual topology is established. The proposed algorithm consistently achieves lower overhead across all cases and, in some instances, more than twice as fast. This improvement is primarily due to the time spent on leader selection in the Distance-halving neighborhood, which is an important part of that algorithm. However, this overhead is a one-time cost, as the virtual topology is created once and reused multiple times, making the overhead amortized over successive neighborhood calls.

The overhead depends on the number of ranks and the density of the graph. The communication time and the time saved by the proposed algorithm depend on the message size and the graph density. As an example, in a random sparse graph δ=0.05, n=600 with m=2MB, 0.45s is saved in each collective call. The overhead of our algorithm is ~0.4s, and that of the default algorithm is ~0.3s. In this case, the overhead is amortized in the first collective call. For a 2B message, we save around ~30μs in each collective call, and so we need more calls to amortize the overhead; however, our algorithm is not designed for small messages



(a) density = 0.05

(b) density = 0.1

(c) density = 0.2

(d) density = 0.4

(e) density= 0.6

Fig. 5: Speedup of the Cascade algorithm compared to the Distance-halving neighborhood algorithm over the default Open MPI algorithm for the sparse matrix-matrix multiplication kernel using various random matrices on Narval

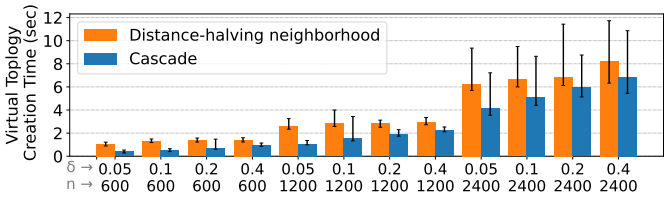


Fig. 6: The overhead of the Cascade algorithm against the Distance-halving neighborhood in random sparse graphs with various densities (δ) and 600, 1200, and 2400 ranks.

F. Memory Footprint Analysis

The primary difference in memory footprints between the algorithms lies in their use of temporary buffers and the memory required to store the communication pattern. Since the memory used to store the communication pattern is in the kilobyte range and therefore negligible, we focus on the temporary buffers.

Cascade uses two auxiliary buffers. The messages received in Step t are forwarded to the next node in Step $t+1$, or copied to the receive buffer. The whole buffer is free and can be reused, while in the Distance-halving algorithm, the messages are accumulated in the buffer. If we consider the first experiment on random sparse graphs with $n = 600$ and $\delta = 0.05$, for 1MB messages, around a 32MB temporary buffer is required on average across the ranks. However, in the Cascade algorithm, on average, 4MB for auxiliary buffers is sufficient. Accumulating messages in a buffer becomes challenging for large messages in dense virtual topologies with a large number of ranks. For $n = 1200$, $\delta = 0.4$, the Distance-halving algorithm needs around 64MB of temporary buffer on average, while for the Cascade algorithm, the number remains around 4MB.

VII. Conclusion and Future Work

In this research, we proposed Cascade, a new topology-aware algorithm for neighborhood allgather targeting medium and large messages. In this algorithm, each process organizes its outgoing nodes in multiple paths based on their distance and sends messages in all paths. The messages are forwarded by the receiving processes to another process in that path until all outgoing processes receive the message. Experimental results show that the Cascade algorithm achieves speedups of up to 9.54x for random sparse graphs and 7.05x for Moore neighborhoods. Furthermore, the algorithm enhances performance by as much as 5.25x for a sparse matrix-matrix multiplication kernel with very large matrices.

Our algorithm has shown its performance on CPUs. As for future work, we plan to extend the algorithm to large-scale GPU clusters. We will also explore other categories of neighborhood collectives, including persistent neighborhood collectives.

VIII. Acknowledgment

This research was supported in part by the Digital Research Alliance of Canada (alliancecan.ca). Computations were performed on the Narval cluster, managed by Calcul Québec.

REFERENCES

[1] "MPI Forum," 2025. [Online]. Available: https://www.mpi-forum.org/

[2] W. Skamarock, J. Klemp, J. Dudhia, D. O. Gill, Z. Liu, J. Berner, W. Wang, J. G. Powers, M. G. Duda, D. Barker, and X.-Y. Huang, "A description of the advanced research WRF model version 4.1," 2019.

[3] S. Páll, A. Zhmurov, P. Bauer, M. Abraham, M. Lundborg, A. Gray, B. Hess, and E. Lindahl, "Heterogeneous parallelization and acceleration of molecular dynamics simulations in GROMACS," *The Journal of Chemical Physics*, vol. 153, no. 13, p. 134110, 10 2020.

[4] "Open MPI," 2025. [Online]. Available: https://www.open-mpi.org/

[5] "MPICH," 2025. [Online]. Available: https://www.mpich.org/

[6] "MVAPICH," 2025. [Online]. Available: https://mvapich.cse.ohio-state.edu/

[7] 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)*. IEEE, 2017, pp. 348–357.

[8] 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)*. IEEE, 2019, pp. 781–792.

[9] 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: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–13.

[10] H. Sharifian, A. Sojoodi, and A. Afsahi, "A topology- and load-aware design for neighborhood allgather," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 131–142.

[11] 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*. IEEE, 2012, pp. 1–10.

[12] W. Gropp, L. N. Olson, and P. Samfass, "Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test," in *Proceedings of the 23rd European MPI Users' Group Meeting*, ser. EuroMPI '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 41–50.

[13] 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 (CCPE)*, no. 3, pp. 1–16, 2020.

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

[15] G. Collom, D. Schafer, A. Bienz, P. Bridges, and G. Shipman, "Optimizing neighbor collectives with topology objects," in *2024 IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 120–130.

[16] 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 Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2010, pp. 1–11.

[17] K. Kandalla, A. Buluç, H. Subramoni, K. Tomko, J. Vienne, L. Oliker, and D. K. Panda, "Can network-offload based non-blocking neighborhood MPI collectives improve communication overheads of irregular graph algorithms?" in *2012 IEEE International Conference on Cluster Computing Workshops*. IEEE, 2012, pp. 222–230.

[18] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard, "Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications," *Parallel Computing*, vol. 38, no. 3, pp. 140–156, 2012.

[19] 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*, 2015, pp. 1–10.

[20] J. Larsson Träff, A. Carpen-Amarie, S. Hunold, and A. Rougier, "Message-combining algorithms for isomorphic, sparse collective communication," *arXiv e-prints*, pp. arXiv–1606, 2016.

[21] J. L. Träff and S. Hunold, "Cartesian collective communication," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[22] J. L. Träff, S. Hunold, G. Mercier, and D. J. Holmes, "MPI collective communication through a single set of interfaces: A case for orthogonality," *Parallel Computing*, vol. 107, p. 102826, 2021.

[23] F. D. Lübbe, "Micro-benchmarking MPI neighborhood collective operations," in *Euro-Par 2017: 23rd International Conference on Parallel and Distributed Computing*. Springer, 2017, pp. 65–78.

[24] G. Collom, R. P. Li, and A. Bienz, "Optimizing irregular communication with neighborhood collectives and locality-aware parallelism," in *SC-W '23: Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, New York, NY, USA, 2023, p. 427–437.

[25] 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.

[26] Y. H. Temuçin, M. Gazimirsaeed, R. E. Grant, and A. Afsahi, "ROCm-aware leader-based designs for MPI neighbourhood collectives," in *39th International Supercomputing Conference (ISC) High Performance*, Hamburg, Germany, 2024, pp. 1–12.

[27] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: an open source framework for HPC network APIs and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[28] P. Erdős and A. Rényi, "On random graphs I," *Publicationes Mathematicae Debrecen*, vol. 6, pp. 290–297, 1959.