Exploiting Virtual Topology for Load- and Communication-Aware Neighborhood Collectives

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 key communication paradigm in MPI, enabling efficient localized communication that is essential for many HPC applications with sparse communication patterns. This paper proposes new algorithms for neighborhood allgather and alltoall, aimed at reducing inter-node communication overhead. The algorithms utilize virtual topology information to construct balanced and optimized communication patterns and employ message combining to reduce the number of messages. This approach leverages virtual topology information to allow ranks on a node to collaboratively restructure their communication patterns. By distributing incoming messages across multiple local receivers, our design reduces communication time and balances the network load, utilizing shared memory as the underlying transport in intra-node communication.**

The proposed algorithms were implemented in Open MPI and evaluated through a comprehensive performance study using random sparse graphs and Moore microbenchmarks, across a wide range of message sizes and neighborhood densities, as well as a sparse matrix-matrix multiplication kernel. Our experimental results demonstrate speedups of up to 18.75x over the default Open MPI implementation for neighborhood allgather and up to 5.61x for neighborhood alltoall. In addition, the proposed algorithm shows up to 4.86x performance improvement over the Distance-halving neighborhood allgather algorithm.

***Index Terms*—MPI, Neighborhood Collectives, Virtual Topology, Allgather, Alltoall, Shared Memory**

I. Introduction

Modern high-performance computing (HPC) systems increasingly rely on fine-grained communication patterns to support sparse workloads such as stencil communications. In many HPC applications, communication primarily revolves around localized interactions. Applications in different areas, such as molecular simulations [1], quantum chemistry [2], weather forecasting [3], and parallel solvers [4] communicate in a neighborhood. Neighborhood collectives play a pivotal role in enabling efficient data exchange between adjacent processes determined by a user-defined communication graph.

The Message Passing Interface (MPI) [5] is a widely used standard for parallel programming. This standard outlines interfaces and semantics for implementing a portable message-passing programming model. MPI supports various communication patterns, including neighborhood communication. In neighborhood collectives, the interactions of each process are limited to a neighborhood. Users define the neighborhoods using the MPI virtual topology interface. This interface offers a variety of operations that allow users to express the communication patterns within their applications through graph or Cartesian topologies.

Neighborhood collectives were introduced in MPI 3.0 [6] to manage the localized communication patterns frequently encountered in HPC applications. A recent survey [7] highlighted that in 85% of HPC applications in their study, communication within a restricted neighborhood is the dominant communication pattern among various types. In 46% of these cases, processes engage in communication within a limited, fixed neighborhood. Furthermore, 29% of the applications aim to integrate neighborhood communication in their upcoming versions, while 9% are looking to implement neighborhood collectives for performance-critical components. However, state-of-the-art implementations of MPI like Open MPI [8], MPICH [9] and MVAPICH [10] use a naïve algorithms for neighborhood allgather and alltoall. These algorithms utilize direct non-blocking point-to-point communication with incoming and outgoing neighbors, leading to high communication latency and scalability issues.

Beyond message-passing capabilities, MPI also incorporates additional routines for communication and synchronization through shared memory, providing more flexibility for intra-node communication. Rather than message exchange, processes can communicate through a shared memory region. This is achieved using MPI shared memory windows.

This research proposes new neighborhood collective algorithms that leverage virtual topology information to establish a load-aware communication pattern. This approach considers virtual topology information to enable ranks on the local node to collaboratively create their communication patterns by evenly distributing incoming messages from remote ranks across multiple local receivers. This strategy balances the communication load across the various inter-node links and intra-node interconnects. Our design shifts costs away from slower inter-node communication to faster intra-node communication.

Our neighborhood allgather design eliminates redundant inter-node messages, reducing both the number of messages and total volume of data transferred over the network. Our neighborhood alltoall design utilizes message combining to limit communication overhead. Additionally, we leverage shared memory to accelerate intra-node data transfer, lowering the total communication latency in neighborhood collectives.

In our design, messages from remote ranks are placed in shared memory regions, allowing local outgoing neighbors to copy the data without incurring excessive inter-node messages. When a rank has only one outgoing neighbor on a node, shared memory regions are bypassed to eliminate unnecessary copying. Our contributions in this research are as follows:

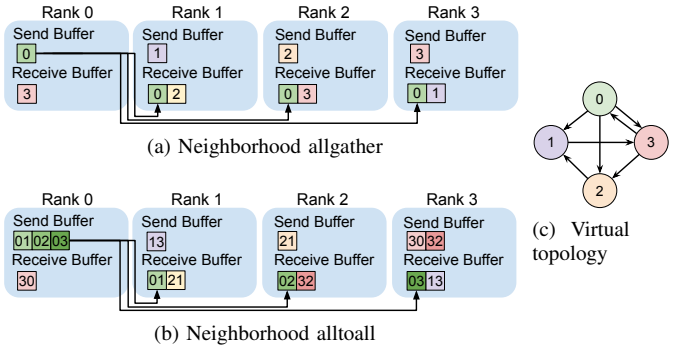
- We propose a balanced communication pattern for neighborhood allgather and alltoall that leverages the virtual topology information to reduce inter-node communication cost.
- We propose a mechanism for communication and synchronization through portable shared memory features of MPI to reduce intra-node communication cost and improve our neighborhood collective design.
- We implement our algorithm in Open MPI to conduct empirical evaluations on a large-scale cluster.
- We evaluate the performance of the proposed algorithm against default Open MPI and the Distance-halving neighborhood allgather [11], one of the state-of-the-art algorithms in the literature, using two microbenchmarks and a sparse matrix-matrix multiplication kernel. We conduct an extensive analysis of the proposed algorithm's scalability by utilizing a wide range of ranks and diverse message sizes across various scenarios.

The rest of this paper is organized as follows. Section II presents the background and motivation for this work. Section III reviews related work. Section IV describes the proposed algorithm, followed by the evaluation of the proposed method in Section V. Finally, Section VI concludes the paper and discusses future work.

II. Background and Motivation

In MPI terminology, communication is performed within a group of processes called a communicator, and each process in a communicator is identified by a unique number called a rank. This paper uses rank to refer to a process, following a common practice in MPI literature. MPI neighborhood collectives utilize the MPI virtual topology interface as their foundation. MPI offers two categories of interfaces for establishing virtual topologies: graph topologies and Cartesian topologies. Among various routines provided by the virtual topology interface of MPI for defining graph topologies, `MPI_Dist_graph_create_adjacent` stands out as the most scalable option and is used in this research. In this distributed interface, each process is only responsible for its incoming and outgoing neighbors, without needing to know the neighbors of other processes. The virtual topology is attached to the communicator and used in neighborhood collective calls.

MPI supports two groups of neighborhood operations: neighborhood allgather and neighborhood alltoall. In a neighborhood allgather, each process sends the same message to all outgoing neighbors, while in a neighborhood alltoall, it sends different messages to each outgoing neighbor. Fig. 1 shows an example of these operations based on a virtual topology. Due to these differences, designing a single communication



(a) Neighborhood allgather

(b) Neighborhood alltoall

(c) Virtual topology

Fig. 1. Send and receive buffers in neighborhood allgather and alltoall for a communicator with four processes based on the given virtual topology. Receive buffers show the data once the neighborhood collective has been completed. The arrows indicate messages sent by rank 0. Arrows for other ranks are omitted for clarity.

pattern that supports both neighborhood allgather and neighborhood alltoall is challenging. However, since establishing a communication pattern incurs additional overhead, using a unified pattern for both operations can reduce the overall cost. In this research, we create a communication pattern that can be used for both operations.

In HPC clusters, inter-node links are typically much slower than intra-node interconnects, and reducing inter-node communication time can therefore lead to significant performance improvements [12]. This can be achieved either by decreasing the total volume of data exchanged or by reducing the number of messages. Lowering the communication volume requires modifying the communication pattern to eliminate unnecessary data transfers, as in operations such as neighborhood allgather. Reducing the number of messages can be accomplished through message combining, where messages destined for the same node are aggregated. Additionally, leveraging virtual topology information to evenly distribute messages, by shifting traffic from heavily loaded ranks to less busy ones, can redistribute network traffic and further reduce communication time.

As the number of cores per node increases, the overhead of redundant data movement inside the nodes increases, and utilizing shared memory for implementing global collectives in MPI becomes more common [13]. Shared memory provides low-latency communication, higher bandwidth, more scalability, and reduced overhead. From an intra-node perspective, communication through shared memory is faster than point-to-point communication due to its ability to bypass the tag-matching engine and point-to-point protocols like rendezvous for large messages [14].

Current MPI implementations do not use shared memory for neighborhood collectives; however, they employ shared-memory mechanisms for intra-node point-to-point and global collectives, and there has been extensive research on using shared memory for global collectives [13], [15], [14]. However, applying shared memory in neighborhood collectives presents a fundamentally different and more complex challenge than in global collectives. Additional constraints, like

the virtual topology and the need to balance load among ranks, introduce additional difficulties beyond those encountered in global collectives. Balancing the load is challenging because communication in neighborhood collectives is sparse, the margin for performance improvement is limited, and even small inefficiencies in design, such as extra synchronization costs, can significantly reduce the overall efficiency of the algorithm. In this research, we use shared memory to improve our neighborhood algorithms.

III. Related Work

Hoefler and Träff [16] discuss the sparse pattern of communication in the applications and the necessity of neighborhood communication and reduction interfaces. They propose these operations as neighborhood gather, alltoall, and reduce in both regular and irregular (vector) variants. Among these operations, gather and alltoall were accepted by the MPI forum and added to the standard as neighborhood allgather and alltoall in MPI 3.0.

Ovcharenko et al. [17] study the communications of various MPI applications and show that the communications of many applications are limited to a fixed neighborhood. They propose a communication package on top of MPI to take advantage of this communication pattern to provide a better performance.

Hoefler and Schneider [18] investigate various approaches aimed at improving the efficiency of neighborhood allgather and alltoall. They utilize user-defined constraints through the `MPI_Info` argument, which allows for features like predefined communication channels and specified message sizes, thereby optimizing the performance of neighborhood collective operations.

Träff et al. [19–22] propose a new interface for applications that have an isomorphic communication pattern. In such applications, the processes are on the virtual grid, but communicate with indirect neighbors in addition to their direct neighbors. MPI Cartesian virtual topology does not support such communication patterns. Their approach is restricted to isomorphic neighborhoods and does not extend to general graph-based neighborhoods.

Lübbe [23] presents a specialized benchmarking suite to evaluate the performance of MPI neighborhood collectives as a group of self-consistent performance guidelines for neighborhood collective operations and proposes microbenchmarks to evaluate these guidelines. Mirsadeghi et al. [24] and Ghazimirsaeed et al. [25] enhance neighborhood aggregation in graph virtual topologies by introducing a message combining approach for groups of K processes. Their approach utilizes the common outgoing neighbors of the processes within each group, and each process sends a combined message containing its own message plus the messages of the other $K-1$ processes to a subset of the common neighbors. Their algorithm does not rely on shared memory.

A hierarchical neighborhood allgather algorithm for medium and large messages is proposed by Ghazimirsaeed et al. [26]. In this algorithm, each process divides its message into multiple chunks and distributes these chunks among its outgoing neighbors located on each remote node. The ranks on the remote node subsequently exchange data and compile the complete message. Persistent neighborhood collectives were introduced in MPI 4.0 [27]. Collom et al. [28] present a new leader-based, locality-aware algorithm designed to optimize persistent alltoallv communication.

Another line of research investigates hardware-assisted techniques for accelerating neighborhood collectives. Kumar et al. [29] propose an interface for Deep Computing Messaging Framework (DCMF) using record and reply mechanism to implement persistent communication in applications that communicate in a neighborhood, running on BlueGene/P machines. Kandalla et al. [30] proposed a NIC-offloaded neighborhood collective mechanism. They discuss how CoreDirect capability in ConnectX-2 network adapters can help in creating scalable non-blocking neighborhood collectives.

On GPU-aware neighborhood collectives, Khorasani et al. [31] propose an algorithm for neighborhood alltoallv that schedules communication with nearest neighbors first to reduce network congestion and better utilize available communication links. Temuçin et al. [32] propose an efficient mechanism for performing GPU-aware allgather and allgatherv operations specifically designed for AMD GPUs.

BoostN [33] captures point-to-point communication patterns in HPC applications that happen in a neighborhood and improves performance using zero-copy, message combining, and process mapping mechanisms. In the Distance-halving neighborhood allgather [11], the communicator is recursively halved, and an agent is chosen on the opposite half in each step. The message is sent to the agent to be delivered to the outgoing neighbors in the other half, to reduce communication with distant ranks. In the Cascade neighborhood allgather [34], messages are forwarded along optimized communication paths that are selected based on the topology of the network.

Our approach is distinct compared to the related work in many ways. Our work targets both neighborhood allgather and alltoall compared to works focusing only on neighborhood allgather or alltoall. Compared to research focusing on Cartesian or isomorphic virtual topologies, our algorithm works with general graphs. Our algorithm does not rely on specific hardware or user-provided information. We propose load-aware leader selection and use shared memory for intra-node communication. we also introduce a mechanism to bypass shared memory when using shared memory adds extra overhead.

IV. Neighborhood Collectives Design

To improve the efficiency of the neighborhood collectives, we propose new topology- and load-aware algorithms to reduce communication overhead. A key advantage of our approach is its adaptability to different collective patterns: in neighborhood allgather, we reduce both the number and total volume of inter-node messages by eliminating redundant data transfers, while in neighborhood alltoall, we reduce the number of messages by combining distinct data into a single transfer. By utilizing shared memory for local distribution and

consolidating traffic destined for the same node, our method significantly reduces communication time.

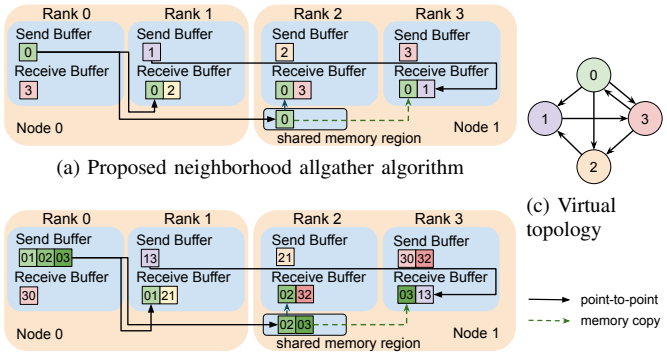
Fig. 2 illustrates the proposed algorithm using the same scenario depicted in Fig. 1. The neighborhood allgather operation is illustrated in Fig. 2a. From rank 0's point of view, ranks 2 and 3 are on the same remote node. Rather than sending two separate messages to 2 and 3, rank 0 sends its message to rank 2. The message is received in a shared memory region by rank 2. In the next step, ranks 2 and 3 copy the received data to their receive buffers. In this way, we reduce the inter-node traffic and intra-node communication overhead. Rank 2 is called the leader of node 1 for rank 0. Rank 3 is the only outgoing neighbor of rank 1 on node 1. To avoid additional memory copy, rank 1 sends its message directly to rank 3, which stores it in its receive buffer.

Fig. 2b illustrates the proposed neighborhood alltoall algorithm, where each outgoing neighbor receives a distinct message. In this example, rank 0 sends a combined message to rank 2, its leader on node 1. The message is received in a shared memory region by rank 2. Ranks 2 and 3 then copy their respective data into their receive buffers. For alltoall, the volume of message transfers is not reduced because the messages are distinct and must be transferred. However, sending them as a combined message and through shared memory enables faster communication.

To elaborate on how we gain a performance boost, we use a mathematical analysis using Hockney's model [35]. This model defines the communication time T for a message of size m using the linear equation $T = \alpha + \frac{m}{\beta}$, where α represents the startup latency and β represents bandwidth. The proposed algorithm enhances efficiency by strategically reducing these variables based on the underlying network. By consolidating distinct data into a single transfer in neighborhood alltoall, the method reduces the total number of messages, thereby minimizing the cumulative impact of the fixed latency cost α. Furthermore, by eliminating redundant data transfers in neighborhood allgather, the algorithm reduces the total message volume, which directly lowers the communication cost. This performance gain is further amplified by exploiting architectural locality; by utilizing shared memory for local distribution and restricting inter-node traffic to consolidated messages, the algorithm avoids the high α and β costs associated with slower network links in favor of the much faster intra-node interconnects. Additionally, by distributing the load between different processes, our algorithm balances the load between inter-node and intra-node links.

A. Notations and Symbols

A summary of symbols used in this section is shown in Table I. The rank of the current process is shown by p, and all variables are defined from p's point of view. The ranks located on the same node as p are called local ranks. The ranks on the other nodes are called remote ranks. $comm$ is the communicator, and n shows its size. The number of local ranks is shown by n_l. The set I includes the incoming neighbors of the current rank, and its size is $indgr$. O presents the set of



(a) Proposed neighborhood allgather algorithm

(c) Virtual topology

(b) Proposed neighborhood alltoall algorithm

Fig. 2. Communication pattern in the proposed neighborhood allgather and allotall algorithms for 2 nodes based on the virtual topology. Data movement from node 1 to node 0 is omitted for clarity.

TABLE I
SUMMARY OF NOTATIONS

Symbol	Description
p	rank of current process
$comm$	MPI communicator
n	size of $comm$
n_l	number of local ranks
m	size of message (bytes)
I	incoming neighbors of p
I^L	local incoming neighbors of p
I^R	remote incoming neighbors of p
I^N	the union of I^R of all ranks on the current node
O	outgoing neighbors of p
O^L	local outgoing neighbors of p
O^R	remote outgoing neighbors of p
O^N	the union of O^R of all ranks on the current node
$indgr_x$	size of I^x, $x \in \{\phi, l, r, n\}$
$otdgr_x$	size of O^x, $x \in \{\phi, l, r, n\}$
$\lvert . \rvert$	size function for sets and vectors

outgoing neighbors of the current process, and $otdgr$ shows its size. The local and remote incoming neighbors of p are shown by I^L and I^R, respectively ($I^L \cup I^R = I$). The size of these sets is shown by $indgr_l$ and $indgr_r$, respectively. Similarly, the local and remote outgoing neighbors of p are shown by O^L and O^R ($O^L \cup O^R = O$), and their sizes are $otdgr_l$ and $otdgr_r$, respectively. I^N and O^N are the node-level incoming and outgoing neighbors defined as the remote incoming and the remote outgoing neighbors of all local ranks, respectively.

B. Shared Memory Synchronization

A key challenge in using shared memory for communication is synchronization. The algorithm must synchronize the leader and receiver ranks to ensure data validity and safe buffer reuse in subsequent neighborhood operations. Fig. 3 illustrates the layout of a shared memory region in the proposed method. Each rank allocates a shared memory region that is partitioned into multiple frames, with each frame designated to receive messages from a remote rank. We design our algorithm as a single-writer, multiple-reader problem, using two shared variables to synchronize access to the shared frame. The $readers$ variable represents the number of readers

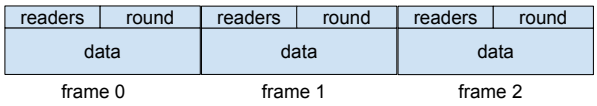


Fig. 3. Layout of shared memory regions

that have not copied the message yet, while $round$ indicates the current communication round. Each communication round corresponds to one invocation of the neighborhood operation. Upon receiving a message, the $readers$ variable is set to the total number of readers for that message. Then $round$ is incremented by one, signaling to the readers that the data of the new round has been received and is ready for copying. Every receiver rank decrements $readers$ after copying the message. When $readers$ is 0, it means all receiver ranks have copied the message of the current round, and the frame can be reused for the next round.

C. Building a Balanced Communication Pattern

The communication pattern is an object that contains all the essential information for neighborhood operations. This includes details about both remote and local outgoing neighbors, the addresses of shared memory regions of the current process and other processes. The communication pattern is created when `MPI_Dist_graph_create_adjacent` is called and attached to the communicator along with the virtual topology. Each rank builds its communication pattern based on virtual topology and locality information, collaborating with both local and remote ranks to optimize message passing and minimize communication overhead. The communication pattern and shared memory regions are created once and reused every time the neighborhood collectives are called.

$build_communication_pattern()$ routine shown in Algorithm 1 illustrates the process of generating the communication pattern. This routine is invoked following the creation of the virtual topology. The inputs are $comm$, I, $indgr$, O, $outdgr$, p, n and the output is the communication pattern cp. The communicator $comm$ is identical to $comm_dist_graph$, which is the output of the virtual topology creation function (`MPI_Dist_graph_create_adjacent`).

In Lines 4–7 of Algorithm 1, the local variables are initialized. The $split_inedges()$ routine, called in Line 8, partitions the incoming neighbors I into two sets: local incoming neighbors I^L and remote incoming neighbors I^R. It also determines their sizes, denoted by $indgr_l$ and $indgr_r$, respectively. Similarly, $split_outedges()$, called in Line 9, partitions the outgoing neighbors O into local and remote sets, O^L and O^R, and determines their corresponding sizes, $outdgr_l$ and $outdgr_r$. In Lines 10 and 11, I^L and O^L are added to cp for communication with local incoming and outgoing neighbors. This communication is performed using direct point-to-point communication.

In Line 12, a local communicator $comm_l$ is created. All ranks in $comm_l$ work together to balance the load for incoming messages from remote ranks. In the routine

Algorithm 1: Building the communication pattern

```
1  Input: comm, I, indgr, O, outdgr, p, n
2  Output: communication_pattern cp
3  Function build_communication_pattern:
4      communication_pattern cp
5      MPI_Comm comm_l                    // local communicator
6      I^R, I^L, I^N , O^R , O^L, receivers ← ∅
7      indgr_r, indgr_l, indgr_n, otdgr_r, otdgr_l, max, n_l ← 0
8      split_inedges (I, indgr, I^L, I^R, indgr_l, indgr_r)
9      split_outedges(O, otdgr, O^L, O^R, otdgr_l, otdgr_r)
10     cp.I^L ← I^L                       // local incoming neighbors
11     cp.O^L ← O^L                       // local outgoing neighbors
12     MPI_Comm_split_type(comm, MPI_COMM_TYPE_SHARED, 0,
         MPI_INF_NULL, &comm_l)           // initialize comm_l
13     MPI_Comm_size(comm_l, &n_l)        // get size of comm_l
14     exchange_inedges(I^R, indgr_r, I^N, indgr_n, receivers,
         comm_l)
15     assigned_I[s] ← ∅, ∀ local rank s
16     leaders[r] ← ∅, ∀r ∈ I^N
17     max ← ⌈indgr_n/n_l⌉
18     direct_I^R ← ∅
19     foreach r ∈ I^N do
20         if |receivers(r)| = 1 then
21             s ← receivers[r][0]
22             if |assigned_I[s]| < max then
23                 leaders[r] ← s
24                 add r to assigned_I[s]
25                 if s = p then
26                     add s to direct_I^R
27                 end
28             end
29         end
30     end
31     foreach r ∈ I^N do
32         if leaders[r] = ∅ then
33             s ← argmin_{i∈receivers[r]} |assigned_I[i]|
34             if |assigned_I[s]| = max then
35                 s ← argmin_{i∈local ranks} |assigned_I[i]|
36             end
37             leaders[r] ← s
38             add r to assigned_I[s]
39         end
40     end
41 end
42 cp.round ← 0
43 cp.assigned_I ← assigned_I
44 cp.receivers ← receivers
45 cp.leaders ← leaders
46 cp.max_readers[r] ← |receivers[r]|, ∀r ∈ assigned_I[p]
47 cp.direct_I^R ← direct_I^R
48 cp.indirect_I^R ← I^R − direct_I^R
49 allocate_shm(comm_l, I^N, I^R, indgr_r, p, cp)
50 exchange notifications
51 return cp
```

$exchange_inedges()$ in Line 14, all ranks in $comm_l$ share their remote incoming neighbors I^R, with each other in order to establish I^N, a set including all remote ranks sending a message to at least one of the ranks in $comm_l$. This function also sets $receivers$ variable, where $receivers[r \in I^N]$ is equal to the set of all outgoing neighbors of r that are in the local communicator $comm_l$.

The list of remote ranks for which the local rank s is the leader is saved in $assigned_I[s]$. The vector $leaders$ holds the same information for each remote rank. $leaders[r] = s$ means that the leader for the remote rank r is the local rank

s. In Lines 15 and 16, $assigned_I$ and $leaders$ are initialized.

We know $indgr_n$ remote ranks communicate with n_l local ranks and evenly distribute the remote ranks among the local ranks. Each local rank will serve as the leader for a maximum of $\lceil indgr_n/n_l \rceil$ remote ranks; this value is stored in variable max in Line 17. We first assign the remote ranks that have only one outgoing neighbor on the current node and bypass shared memory regions. The $foreach$ loop, spanning Lines 19 to 30, assigns the direct remote incoming neighbors to their corresponding processes in each iteration. If the current rank is the outgoing neighbor of that remote process, the remote rank is assigned to the current rank and added to $direct_I^R$ in Line 26.

If all outgoing neighbors on a remote rank have already reached the maximum, a non-outgoing neighbor is selected as leader to maintain balanced communication load; however, this increases the synchronization cost. To address this issue, a collaborative leader selection mechanism is employed to reduce the need for selecting leaders from non-outgoing neighbors while also balancing the communication load among ranks. In the $foreach$ loop, spanning Lines 31-40, we assign the leaders who work with shared memory. From the outgoing neighbors at remote rank r, we select the one to which we have assigned the smallest number of remote ranks and it has not reached the maximum in Line 33. If no such rank is found, we assign that remote rank to another process on that node in Line 35, which means the leader of the local ranks is not one of its outgoing neighbors.

In Line 42, we initialize $cp.round$ to zero for the first communication round. In Lines 43-45, the variables $assigned_I$, $receivers$, and $leaders$ are added to the communication pattern. The current rank p needs to know how many local ranks will copy messages received by it from its shared memory frames. Therefore, we keep track of it in $cp.max_readers$. For each remote rank $r \in assigned_I[p]$, we set $cp.max_readers[r]$ to the number of outgoing neighbors of r on the current node at Line 46. The remote incoming neighbors of the current process whose data is received in the shared memory are stored in $cp.indirect_I^R$. The function $allocate_shm()$ allocates shared memory regions. This function is presented in Section IV-D. In Line 50, the remote ranks I^N are notified about their leaders. The local ranks also receive notification from remote ranks and update $cp.remote_leaders$, including remote ranks that are leader of the current process on their nodes.

D. Allocating Shared Memory Regions

Allocating shared memory regions and exposing them to other processes requires OS kernel intervention, resulting in additional overhead. To mitigate this, we preallocate shared memory regions and reuse them. For improved algorithm portability, we utilize the collective method `MPI_Win_allocate_shared`, which allocates shared memory regions.

Function $allocate_shm()$ shown in Algorithm 2 allocates the shared memory regions in our design. An MPI shared memory window object `MPI_Win` is defined in Line 3. Variables my_ptr, $remote_ptr$ are used to keep the pointers to the local window and the windows of other ranks, respectively. In Line 6, the size of the shared memory region is calculated based on the number of remote ranks for which the current process is the leader and whose data is received in the shared memory region. We allocate shared memory regions with size win_size in Line 7. Function `MPI_Win_allocate_shared` sets the my_ptr and win variables. In the loop starting at Line 8, pointers to the shared memory frames are added to the communication pattern. For each incoming neighbor $r \in I^R$ whose data is received by another rank $leaders[r]$, we use `MPI_Win_shared_query` to obtain the base address of $leaders[r]$'s shared memory region in $remote_ptr$. We use this pointer to calculate the frame of rank r in the shared memory region of $leaders[r]$. For remote ranks received by the current process in the shared memory region, we use my_ptr to calculate the pointers to the frames in Line 13.

Algorithm 2: Allocating shared memory regions

```
1  Input: comm_l, I^N, I^R, indgr_r, p, cp
2  Function allocate_shm:
3      MPI_Win win                 // MPI shared memory window
4      void* my_ptr, remote_ptr ← NULL  // base pointers
5      int disp_unit ← 1
6      MPI_Aint win_size ← calc_wind_size(cp)
7      MPI_Win_allocate_shared(win_size, disp_unit,
         MPI_INFO_NULL, comm_l, &my_ptr, &win)
8      foreach r ∈ I^N do
9          if r ∈ I^R and cp.leaders[r]! = p then
10             MPI_Win_shared_query(win, cp.leaders[r], &win_size,
                 &disp_unit, &remote_ptr)
11             use remote_ptr to calculate pointers in cp.frames[r]
12         else if cp.leaders[r] = p and r ∉ cp.direct_I^R then
13             use my_ptr to calculate pointers in cp.frames[r]
14         end
15     end
16 end
```

E. Neighborhood Allgather

Algorithm 3 leverages the communication pattern created in Section IV-C to run the neighborhood allgather. The inputs are equivalent to the inputs of `MPI_Neighbor_allgather`. The communication pattern established earlier is retrieved in Line 3, and the subsequent communications are carried out according to it. In Line 4, $cp.round$, a private variable to each rank, is incremented, and its value is used as a signal between processes. Two flag arrays, pst_flags and rcv_flags, are initialized to track the posted and completed receive requests, respectively. Following this setup, in Lines 8-10, the algorithm addresses direct incoming neighbors, stored in $cp.direct_I^R$, whose messages are directly received in the receive buffer.

In the while loop spanning Lines 12-20, non-blocking receive requests are posted for the remote ranks that selected the current rank as their leader on the current node, and their messages are received in the shared memory $(cp.assigned_I - cp.direct_I^R)$. We delay posting a receive request until the data from the previous neighborhood call has

Algorithm 3: Neighborhood allgather design

```
1  Input: sendbuf, sendcount, sendtype, recvbuf, recvcount,
          recvtype, comm
2  Function MPI_Neighbor_allgather:
3      communication_pattern cp ← comm.cp
4      cp.round ← cp.round + 1
5      pst_flags ← {0, 0, ..., 0}
6      rcv_flags ← {0, 0, ..., 0}
7      MPI_Comm_rank(comm, &p)
8      foreach r ∈ cp.direct_I^R do
9          irecv from r in recvbuf
10     end
11     c ← |cp.assigned_I[p] − cp.direct_I^R|
12     while c > 0 do
13         foreach r ∈ (cp.assigned_I[p] − cp.direct_I^R) do
14             if cp.frames[r].readers = 0 and pst_flags[r] = 0 then
15                 irecv from r in cp.frames[r]
16                 pst_flags[r] ← 1
17                 c ← c − 1
18             end
19         end
20     end
21     foreach r ∈ cp.remote_leaders do
22         isend from sendbuf to r
23     end
24     c ← |cp.assigned_I[p] − cp.direct_I^R|
25     while c > 0 do
26         index ← wait_any on submitted requests
27         if index is related to r ∈ cp.assigned_I[p] at Line 15 then
28             atomic_set(cp.frames[r].readers, cp.max_readers[r])
29             atomic_set(cp.frames[r].round, cp.round)
30             c ← c − 1
31         end
32     end
33     foreach r ∈ cp.I^L do
34         irecv from r in recvbuf
35     end
36     foreach r ∈ cp.O^L do
37         isend from sendbuf to r
38     end
39     c ← |cp.indirect_I^R|
40     while c > 0 do
41         foreach r ∈ cp.indirect_I^R do
42             if !rcv_flags[r] then
43                 if cp.frames[r].round = cp.round then
44                     copy data from cp.frames[r] to recvbuf
45                     atomic_decrement(cp.frames[r].readers)
46                     rcv_flags[r] ← 1
47                     c ← c − 1
48                 end
49             end
50         end
51     end
52     wait_all
53 end
```

Algorithm 4: Neighborhood alltoall design

```
1  Input: sendbuf, sendcount, sendtype, recvbuf, recvcount,
          recvtype, comm
2  Function MPI_Neighbor_alltoall:
3      communication_pattern cp ← comm.cp
4      cp.round ← cp.round + 1
5      pst_flags ← {0, 0, ..., 0}
6      rcv_flags ← {0, 0, ..., 0}
7      MPI_Comm_rank(comm, &p)
8      foreach r ∈ cp.direct_I^R do
9          irecv from r in recvbuf
10     end
11     c ← |cp.assigned_I[p] − cp.direct_I^R|
12     while c > 0 do
13         foreach r ∈ (cp.assigned_I[p] − cp.direct_I^R) do
14             if cp.frames[r].readers = 0 and pst_flags[r] = 0 then
15                 irecv from r in cp.frames[r]
16                 pst_flags[r] ← 1
17                 c ← c − 1
18             end
19         end
20     end
21     foreach r ∈ cp.remote_leaders do
22         isend single message to r, including data of all remote
             outgoing neighbors that are on the same node with r
23     end
24     c ← |cp.assigned_I[p] − cp.direct_I^R|
25     while c > 0 do
26         index ← wait_any on submitted requests
27         if index is related to r ∈ cp.assigned_I[p] at Line 15 then
28             atomic_set(cp.frames[r].readers, cp.max_readers[r])
29             atomic_set(cp.frames[r].round, cp.round)
30             c ← c − 1
31         end
32     end
33     foreach r ∈ cp.I^L do
34         irecv from r in recvbuf
35     end
36     foreach r ∈ cp.O^L do
37         isend data intended for r from sendbuf to r
38     end
39     c ← |cp.indirect_I^R|
40     while c > 0 do
41         foreach r ∈ cp.indirect_I^R do
42             if !rcv_flags[r] then
43                 if cp.frames[r].round = cp.round then
44                     calculate the offset of data in cp.frames[r]
45                     copy data from cp.frames[r] to recvbuf
46                     atomic_decrement(cp.frames[r].readers)
47                     rcv_flags[r] ← 1
48                     c ← c − 1
49                 end
50             end
51         end
52     end
53     wait_all
54 end
```

been copied by all receiver processes. To verify this, we check $cp.frames[r].readers$ in Line 14. A value of zero indicates that all receivers have finished copying the data, and the shared memory frame of r can be reused. At this point, we post a receive request from r and update the flags accordingly in Line 16. The flag $pst_flags[r]$ is updated to prevent duplicate postings. This process is repeated until all remote receive requests are posted.

Subsequently, in Lines 21-23, the current rank sends its data to the ranks stored in $cp.remote_leaders$. Once the sends are issued, the algorithm waits for the completion of the previously posted receives from remote ranks. This step happens in the while loop spanning Lines 25-32. To monitor the completion of any receive $wait_any$ is employed, and upon receiving data, it atomically updates the $readers$ count in the corresponding frame to $cp.max_readers[r]$ and assigns the current round number, $cp.round$, to the frame, signaling that the data is now available for reader processes to copy.

The intra-node communication is handled in Lines 33-38. The current rank initiates non-blocking receives from local incoming neighbors $cp.I^L$, and non-blocking sends to

local outgoing neighbors $cp.O^L$. Afterward, in Lines 40-51, the algorithm iterates over the shared memory frames of other local processes containing data from remote incoming neighbors. For each rank $r \in cp.indirect_I^R$, the algorithm checks if the data has not been copied before, as indicated by the flag $rcv_flags[r]$. If the frame's $round$ matches the current round, the data is ready to be copied to the final receive buffer. The $readers$ count is then atomically decremented, and the flag $rcv_flags[r]$ is set to indicate that data is copied.

Finally, the algorithm ensures that all communication operations have completed by invoking $wait_all$ in Line 52. This approach minimizes redundant data transfers using shared memory, allowing multiple local ranks to access the same remote data efficiently.

F. Neighborhood Alltoall

Neighborhood alltoall algorithm is shown in Algorithm 4, which is similar to neighborhood allgather. In the neighborhood alltoall algorithm, each rank sends the data for all remote processes located on the same remote node as a single message to that node's leader, which is shown in Line 22 of Algorithm 4. In Line 45, when copying data from the shared memory region, the offset corresponding to the current process is used to locate its data within the shared memory frame.

V. Evaluation and Analysis

In this section, we evaluate the performance of the proposed neighborhood allgather and alltoall algorithms using random sparse graphs and Moore microbenchmarks, representing different communication patterns. For neighborhood allgather, performance is additionally evaluated using a sparse matrix-matrix multiplication kernel.

We implemented our algorithms in Open MPI v5.0.0r12 and compiled them with UCX v1.17.0. The experiments were conducted on Niagara cluster [36] at the Digital Research Alliance of Canada. Niagara is designed for large-scale workloads and consists of 2024 nodes, each equipped with two Intel Skylake or Cascade Lake CPUs, totaling 40 cores per node, and 202 GB of RAM. The nodes are interconnected via a 100 Gb/s EDR InfiniBand network configured in a Dragonfly+ [37] topology with adaptive routing. We allocated 36 cores on each node to the MPI jobs, leaving 4 cores for the OS and other system services. Each experiment is conducted a minimum of three times, and we present the mean, maximum, and minimum values obtained. We classify message sizes into three categories: small ($\leq$ 4 KB), medium (> 4 KB and $\leq$ 256 KB), and large (> 256 KB), to examine the results in greater detail. Vertical lines in the figures indicate the boundaries between these categories.

For neighborhood allgather, we compare our algorithm to the default Open MPI and the state-of-the-art Distance-halving neighborhood algorithm [11]. For neighborhood alltoall, we compare only against the default Open MPI, as no publicly available source code exists for neighborhood alltoall. In this section, we use LA-SHM to refer to our load-aware shared memory neighborhood algorithms.

A. Random Sparse Graphs

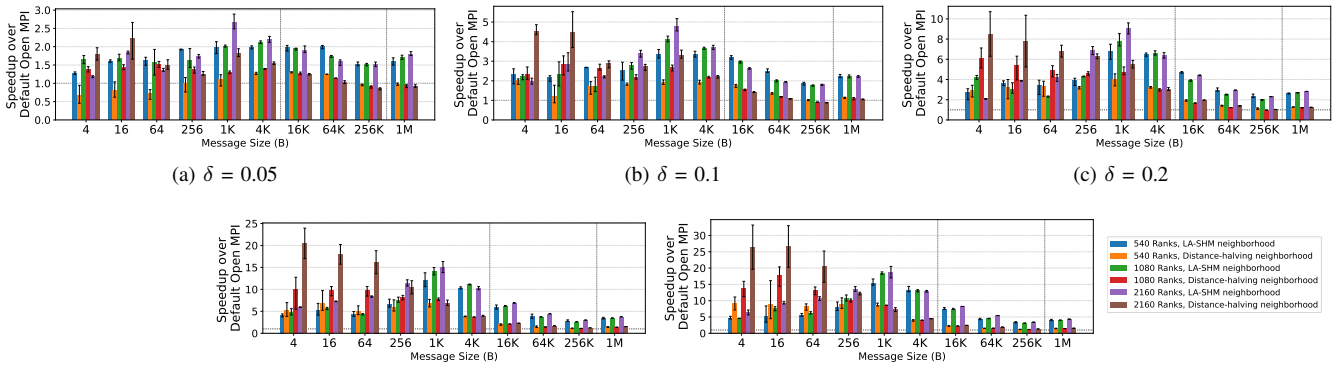
1) Neighborhood Allgather: To evaluate our algorithm across different virtual topologies, we use Erdős–Rényi model [38] to generate random sparse graphs with various sizes and densities. In this model, the parameter δ shows the probability of one rank being chosen as the outgoing neighbor of the other, for all ordered pairs of ranks. Each possible edge is included independently according to a Bernoulli trial with success probability δ.

Figure 4 compares the speedup achieved by the proposed algorithm for neighborhood allgather against the Distance-halving neighborhood over the default Open MPI. The experiment covers random graphs with densities ranging from 0.05 to 0.6 and includes sizes of 540, 1080, and 2160 ranks. The speedup of the proposed algorithm in all cases is greater than one, which means it is faster than the default algorithm.

For very small messages up to 256B, the proposed algorithm outperforms the default Open MPI, whereas the Distance-halving neighborhood outperforms the proposed algorithm in most cases. This speedup is because of the message combining used in the Distance-halving, which considerably increases its performance for small messages. However, for messages larger than 256B, our algorithm outperforms both Open MPI and the Distance-halving neighborhood in most cases.

As we increase the number of ranks, the speedup of the proposed algorithm remains steady or improves in most cases, which shows the scalability of the proposed algorithm. Also, as we increase the density of the graph, the proposed algorithm demonstrates an increase in speedup. This suggests that the algorithm successfully harnesses the increased opportunities for optimization present in denser graphs. Overall, for large messages, the speedup of the proposed algorithm is between 1.50x and 5.10x over Open MPI, with a mean of 2.70x, while the Distance-halving algorithm shows speedup between 0.85x and 1.57x over Open MPI, with an average of 1.17x. For medium-sized messages, the speedup of the proposed algorithm is between 1.51x and 10.71x, with a mean of 4.04x. In contrast, the Distance-halving neighborhood shows 0.89x up to 3.57x with a mean of 1.65x. For small messages, the speedup of the proposed algorithm is between 1.16x and 18.74x, with a mean of 5.27x. However, the Distance-halving algorithm shows speedups ranging from 0.57x to 27.62x, with a mean of 5.88x.

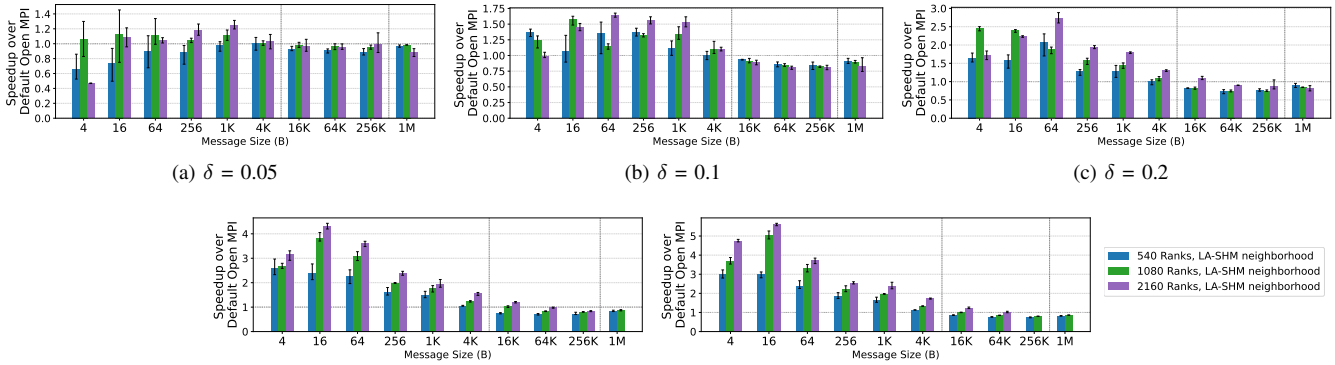
2) Neighborhood Alltoall: We repeated the experiments for neighborhood alltoall with the same number of ranks and graph densities. The results are shown in Fig. 5. Since the data differs for each outgoing neighbor in neighborhood alltoall, a larger volume of messages must be transferred compared to the neighborhood allgather. This results in higher communication time for the alltoall operation. The performance of the proposed algorithm is modest for the most sparse density. The maximum speedup for $\delta = 0.05$ is 1.25x and reaches up to 5.61x for $\delta = 0.6$. The average speedup for $\delta = 0.05$ is 0.95x, which shows a small degradation against the default Open MPI, while it rises to 2.94x for $\delta = 0.6$. For small



(a) δ = 0.05 (b) δ = 0.1 (c) δ = 0.2

(d) δ = 0.4 (e) δ = 0.6

Fig. 4. Speedup of the proposed algorithm compared to the Distance-halving neighborhood over the default Open MPI for neighborhood allgather on random sparse graphs with various densities (δ) and message sizes.



(a) δ = 0.05 (b) δ = 0.1 (c) δ = 0.2

(d) δ = 0.4 (e) δ = 0.6

Fig. 5. Speedup of the proposed algorithm over the default Open MPI for neighborhood alltoall on random sparse graphs with various densities (δ) and message sizes.

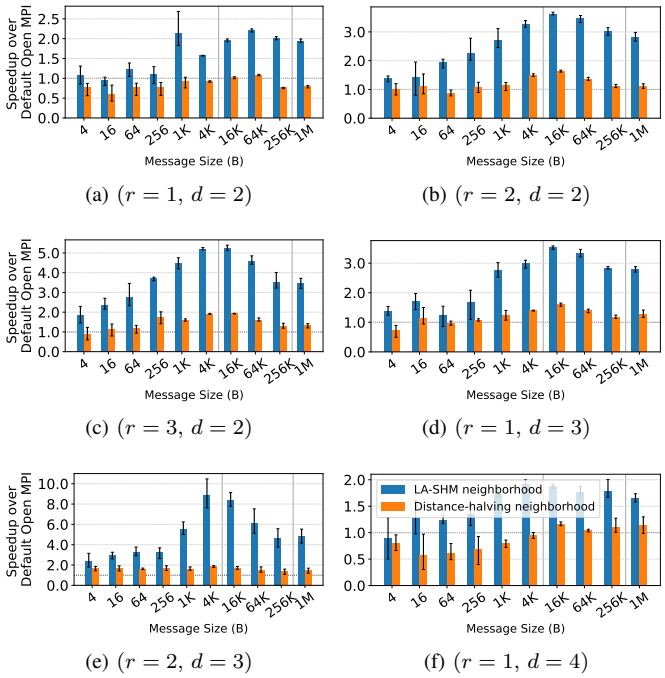
to medium message sizes, the proposed algorithm achieves a mean speedup of 1.57x and a maximum of 5.61x. However, this speedup is from 0.86x to 1.03x for larger message sizes.

Based on the popular Hockney's Model, the limiting factor for small messages is α, while for large messages, β becomes the dominant factor. For small messages, our algorithm shows a better performance as it combines messages of the outgoing neighbors of each rank on a remote node into a single message. In contrast, for large messages, these parameters become negligible, and the benefit of aggregating messages into a single one diminishes. Therefore, the network term $\frac{m}{\beta}$ becomes the determining factor. This term is dependent on the total volume of the messages, not the frequency of them, resulting in a performance drop for large messages. Nonetheless, the probability of congestion is higher in denser graphs, and our algorithm can effectively reduce congestion by combining intra-node messages from multiple outgoing neighbors into a single message, particularly for small messages.

B. Moore Neighborhoods

Moore neighborhoods represent an important communication pattern in HPC applications. A Moore neighborhood is defined by two parameters, r and d. Processes are arranged on a virtual d-dimensional grid, and each process communicates with all $(2r+1)^d - 1$ ranks located within a Chebyshev distance of r. We conducted a series of experiments utilizing different Moore neighborhoods to assess the performance of the proposed algorithm for both neighborhood allgather and alltoall operations. Moore neighborhoods are used to study the performance of neighborhood collectives in previous research [19], [26], [32], [34].

1) Neighborhood Allgather: Fig. 6 compares the speedup of the proposed algorithm against Distance-halving neighborhood over the default Open MPI neighborhood allgather for various Moore neighborhoods over 2160 ranks. The proposed algorithm achieves a maximum of 8.66x speedup against the default Open MPI at 4KB messages for the largest Moore neighborhood with $r = 2$, $d = 3$. The average speedup of the proposed algorithm in these experiments is 2.81x. In comparison, the Distance-halving neighborhood reaches a max speedup of 1.92x and an average of 1.2x. The minimum speedup of the proposed algorithm is 0.90x against 0.57x for the Distance-halving neighborhood, both at 16B messages in Fig. 6a.

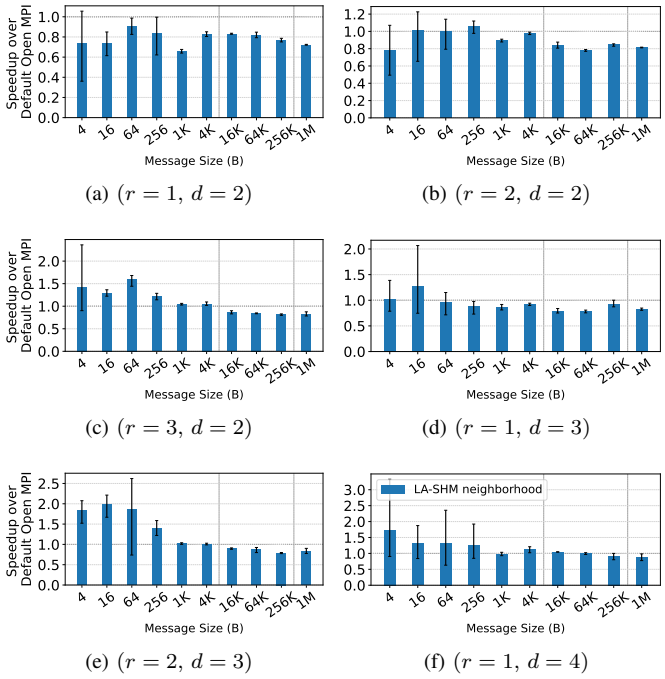


(a) $(r = 1,\ d = 2)$ (b) $(r = 2,\ d = 2)$

(c) $(r = 3,\ d = 2)$ (d) $(r = 1,\ d = 3)$

(e) $(r = 2,\ d = 3)$ (f) $(r = 1,\ d = 4)$

Fig. 6. Speedup of the proposed algorithm compared to the Distance-halving neighborhood over the default Open MPI for neighborhood allgather with 2160 ranks and various Moore neighborhoods and different message sizes.



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

(c) $(r = 3,\ d = 2)$ (d) $(r = 1,\ d = 3)$

(e) $(r = 2,\ d = 3)$ (f) $(r = 1,\ d = 4)$

Fig. 7. Speedup of the proposed algorithm over the default Open MPI for neighborhood alltoall on various Moore neighborhoods with 2160 ranks and different message sizes.

2) Neighborhood Alltoall: Fig. 7 illustrates the speedup of the proposed alltoall algorithm over the default Open MPI for various Moore neighborhoods on 2160 ranks. The proposed algorithm shows the best performance on denser neighborhoods, with modest performance on sparser neighborhoods. The lower speedups in this experiment compared to allgather are related to the different messages transferred in alltoall. In a sparser neighborhood, the memory copy and synchronization overhead in the proposed algorithm pushes the speedup below 1.0. However, when the neighborhood becomes denser, our algorithm begins to show its benefit.

Since the data sent to each outgoing neighbor is different in alltoall and we only reduce the number of messages, the deciding factor in the Hockney's Model is α. As mentioned earlier, this parameter is important for small messages, and so this explains the increased performance for small messages. On the one hand, the impact of α is small for large messages. On the other hand, memory copy overhead is higher for large messages. These factors reduce the speedup for the shared memory neighborhood alltoall for large messages. However, in denser topologies, congestion turns into a critical factor, and the proposed algorithm demonstrates enhanced performance by effectively alleviating congestion.

Another factor that impacts the speedup in these experiments is the locality of the data. In Moore neighborhoods, increasing the number of ranks tends to reduce neighborhood locality, even when topology-aware mapping techniques are applied. The proposed algorithm is specifically optimized to improve non-local communications. Consequently, as the rank count increases, particularly in sparse graph scenarios where communication patterns are more dispersed, the algorithm demonstrates improved speedup due to its ability to efficiently handle non-local data exchanges.

The cluster used in our experiments is fully connected and has a 1:1 blocking factor between the nodes. In such topologies, the naïve algorithm benefits significantly from the abundance of available paths, naturally achieving higher performance. As a result, the relative speedup of our algorithm appears lower, particularly for sparser graphs. This happens not due to inefficiencies in its design, but because the baseline is already strong under these network conditions, and this is the worst-case scenario for our algorithm. Overall, the proposed algorithm reaches a maximum of 1.99x speedup with an average of 1.01x for neighborhood alltoall on Moore neighborhoods.

C. Sparse Matrix-Matrix Multiplication

Sparse matrix-matrix multiplication is one of the most important operations in HPC and AI applications. We used a sparse matrix-matrix multiplication kernel that uses neighborhood allgather to evaluate the proposed algorithm. In this kernel, which is used in previous research [25], [34], the product of two matrices $X_{i\times j}$ and $Y_{j\times k}$ is calculated to produce matrix $Z_{i\times k} = X \times Y$. This kernel uses the sparsity of matrix X to improve performance. Matrix Y can be either sparse or dense. The rows of these matrices are partitioned into n strips, including $\frac{i}{n}$ and $\frac{j}{n}$ rows of those matrices. Before the operation, each process holds a strip of matrices X and Y. Each strip includes n consecutive virtual blocks

TABLE II
LIST OF MATRICES

Matrix	Size	Non-zero Elements
Journals	124 × 124	12068
dwt_193	193 × 193	3493
cegb2802	2802 × 2802	277362
heart1	3557 × 3557	1385317
msc01440	1440 × 1440	44998
mycielskian12	3071 × 3071	407200
bcsstk13	2003 × 2003	83883
nasa2910	2910 × 2910	174296

along the columns. The virtual topology is created based on the blocks of X, with at least one non-zero element. The function `MPI_Neighbor_allgather` is employed to gather the columns of Y to each process. We launched this kernel with multiple matrices with various sizes and sparsity levels from the SuiteSparse Matrix collection [39] to study the performance of the proposed design in various scenarios. These matrices are listed in Table II.

The proposed algorithm shows its best performance, achieving a 28.3x speedup on mycielskian12, which is the second largest matrix with moderate density. The second best speedup is achieved on nasa2910, which is the third-largest matrix. The minimum speedup is related to msc01440 matrix with 1.19x, which shows that our algorithm outperforms the default Open MPI algorithm in all experiments. The performance of the Distance-halving algorithm is better for three matrices: cegb2802, heart1, and msc01440. The Distance-halving neighborhood demonstrates optimal performance when applied to small messages and dense matrices.

The speedups for heart1, which is the largest matrix, are too close to 2x, and the Distance-halving neighborhood algorithm is slightly faster because this matrix is dense and we used a large number of ranks, which is the most suitable scenario for the Distance-halving neighborhood algorithm. cegb2802 is a large matrix, but we used a smaller number of ranks for this matrix, which increases the density of the virtual topology. On the other hand, msc01440 is a small and sparse matrix and does not leave too much opportunity for a performance boost.

Overall, the proposed algorithm achieves speedups ranging from 1.19x to 28.13x, with an average speedup of 5.72x across all matrices. In contrast, the Distance-halving neighborhood demonstrates speedups ranging from 1.04x to 2.47x, and an average of 1.73x.

D. Overhead Analysis

This section discusses the communication pattern creation overhead. Figure 9 compares the proposed algorithm's overhead with that of the Distance-halving neighborhood for the random sparse-graph experiments described in Section V-A. In every scenario, the overhead incurred by the proposed algorithm is lower and remains nearly constant as graph density increases, while the Distance-halving approach exhibits rising overhead at higher densities. Since the virtual topology is constructed once and then reused across multiple neighborhood collective operations, this one-time cost is rapidly amortized, leaving communication latency to dominate overall performance.

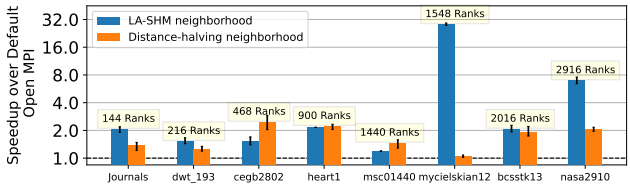


Fig. 8. Speedup of the proposed algorithm and the Distance-halving neighborhood over the default Open MPI implementation for the matrix-matrix multiplication kernel based on neighborhood allgather, using the matrices listed in Table II

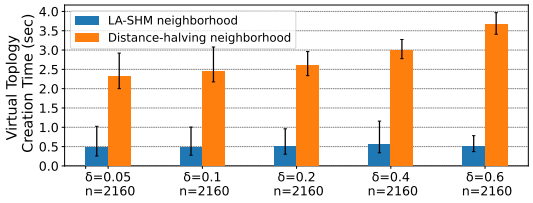


Fig. 9. Virtual topology creation overhead of the proposed algorithm against the Distance-halving neighborhood for 2160 ranks

VI. Conclusion and Future Work

This work introduces new algorithms for neighborhood collectives that reduce inter-node communication by leveraging virtual topology information. We create a load-aware communication pattern that eliminates extra data movement in neighborhood allgather and message combining in neighborhood alltoall. Our algorithm uses shared memory and allows remote ranks to write directly into shared memory regions, enabling the local ranks to efficiently retrieve data. The proposed approach improves performance across various communication patterns. Implemented within Open MPI and evaluated using two micro-benchmarks and a sparse matrix-matrix multiplication kernel, the algorithms demonstrated substantial performance gains, achieving up to 18.75x speedup over the default Open MPI implementation for neighborhood allgather and up to 5.61x for neighborhood alltoall. In future work, we intend to extend this approach by making it NUMA-aware, adapting dynamically to communication patterns, and enabling support for GPUs. We also plan to contribute our implementation to the Open MPI codebase.

VII. Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful comments, which helped improve the quality of this paper. This research was supported in part by the Digital Research Alliance of Canada. Computations were performed on the Niagara cluster [36], with support from the SciNet HPC Consortium.

REFERENCES

[1] 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*, pp. 1–16, 2020.

[2] T. D. Kühne, M. Iannuzzi, M. Del Ben, V. V. Rybkin, P. Seewald, F. Stein, T. Laino, R. Z. Khaliullin, O. Schütt, F. Schiffmann *et al.*, "CP2K: An electronic structure and molecular dynamics software package - Quickstep: Efficient and accurate electronic structure calculations," *The Journal of Chemical Physics*, pp. 1–95, 2020.

[3] 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. [Online]. Available: https://doi.org/10.5065/1dfh-6p97

[4] V. E. Henson and U. M. Yang, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, pp. 155–177, 2002.

[5] "MPI Forum," 2026. [Online]. Available: https://www.mpi-forum.org/

[6] MPI Forum, "MPI: A Message-Passing Interface Standard, Version 3.0," University of Tennessee, Knoxville, TN, USA, MPI Forum Document MPI-3.0, 2012. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[7] 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)*, pp. 1–16, 2020.

[8] "Open MPI," 2026. [Online]. Available: https://www.open-mpi.org/

[9] "MPICH," 2026. [Online]. Available: https://www.mpich.org/

[10] "MVAPICH," 2026. [Online]. Available: https://mvapich.cse.ohio-state.edu/

[11] H. Sharifian, A. Sojoodi, and A. Afsahi, "A topology- and load-aware design for neighborhood allgather," in *Proceedings of the 26th IEEE International Conference on Cluster Computing (CLUSTER)*, 2024, pp. 131–142.

[12] A. Bienz, S. Gautam, and A. Kharel, "A locality-aware Bruck allgather," in *Proceedings of the 29th European MPI Users' Group Meeting*, ser. EuroMPI/USA '22. Association for Computing Machinery, 2022, p. 18–26.

[13] J. Peng, J. Fang, J. Liu, M. Xie, Y. Dai, B. Yang, S. Li, and Z. Wang, "Optimizing MPI collectives on shared memory multi-cores," *SC23: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2023.

[14] G. Katevenis, M. Ploumidis, and M. Marazakis, "A framework for hierarchical single-copy MPI collectives on multicore nodes," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, 2022, pp. 94–105.

[15] S. Li, T. Hoefler, and M. Snir, "NUMA-aware shared-memory collective communication for MPI," in *Proceedings of the 22nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '13. Association for Computing Machinery, 2013, p. 85–96.

[16] 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.

[17] 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*, pp. 140–156, 2012.

[18] 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.

[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*, 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*, 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] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize MPI neighborhood collectives," in *IEEE 24th international conference on high performance computing (HiPC)*. IEEE, 2017, pp. 348–357.

[25] 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.

[26] 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.

[27] MPI Forum, "MPI: A Message-Passing Interface Standard, Version 4.0," University of Tennessee, Knoxville, TN, USA, MPI Forum Document MPI-4.0, 2021. [Online]. Available: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[28] 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*, 2023, p. 427–437.

[29] 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.

[30] 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.

[31] 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.

[32] 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*, 2024, pp. 1–12.

[33] H. Huang, Y. Jin, and W. Xue, "BoostN: Optimizing imbalanced neighborhood communication on homogeneous many-core system," in *Proceedings of the 53rd International Conference on Parallel Processing*, ser. ICPP '24. Association for Computing Machinery, 2024, p. 262–272.

[34] H. Sharifian, A. Sojoodi, and A. Afsahi, "Cascade: a collaborative algorithm for scalable and efficient neighborhood allgather," in *Proceedings of the 27th IEEE International Conference on Cluster Computing (CLUSTER)*, 2025, pp. 1–13.

[35] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing*, pp. 389–398, 1994.

[36] M. Ponce, R. van Zon, S. Northrup, D. Gruner, J. Chen, F. Ertinaz, A. Fedoseev, L. Groer, F. Mao, B. C. Mundim, M. Nolta, J. Pinto, M. Saldarriaga, V. Slavnic, E. Spence, C.-H. Yu, and W. R. Peltier, "Deploying a top-100 supercomputer for large parallel workloads: The Niagara supercomputer," in *Practice and Experience in Advanced Research Computing 2019: Rise of the Machines (Learning)*, ser. PEARC '19. Association for Computing Machinery, 2019, pp. 1–8.

[37] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, and E. Zahavi, "Dragonfly+: Low cost topology for scaling datacenters," in *IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB)*, 2017, pp. 1–8.

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

[39] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, pp. 1–25, 2011.