A Topology- and Load-Aware Design for 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 collective communications were introduced in MPI 3.0 to enable application developers to define new communication patterns and take advantage of the sparsity in the communication patterns of applications. In this research, we propose a novel topology- and load-aware distance-halving design for neighborhood allgather. In this algorithm, each rank recursively halves the communicator and finds an agent on the opposite half to offload its outgoing neighbors. This approach limits communication with distant ranks, thereby decreasing the latency of neighborhood allgather. Our experimental study demonstrates that our proposed algorithm can outperform the default implementation of Open MPI by up to 30x and 14x speedup for Random Sparse Graph and Moore neighborhood micro-benchmarks, respectively. Furthermore, our design exhibits up to 4.92x performance gain for an SpMM Kernel.**

***Index Terms*—MPI, topology, neighborhood collective**

I. Introduction

Message passing interface (MPI) [20] is a standard programming model developed for High-Performance Computing (HPC) systems. MPI specifies portable interfaces and semantics for the message-passing model. The most basic operation in the message-passing model is data movement from the memory of one process to another, called point-to-point communication. MPI supports other communication models, including Remote Memory Access (RMA), Partitioned communication, collective communication, and neighborhood collectives. Collectives are a form of communication in which all processes in a group are involved in the communication.

In global collectives, each process communicates with all processes or with a root process within a group of processes called a communicator. In neighborhood collectives, the communication of each process is restricted to a neighborhood. MPI neighborhood collectives are defined on top of the MPI virtual topology interface in which programmers can specify each process's incoming and outgoing neighbors. Using neighborhood collectives, users can create customized communication patterns. In addition, the information provided by the user can be used for more optimization at the MPI layer.

A recent survey conducted by the U.S. Exascale Computing Project (ECP) [2] has reported that the communication of 46% of HPC applications is limited to a fixed neighborhood and 29% of HPC applications are expected to utilize neighborhood collectives in their future versions. Additionally, Neighborhood collectives are considered for use in the performance-critical sections of 9% of them. However, the state-of-the-art MPI implementations such as Open MPI [23], MPICH [21], and MVAPICH [22] support a naïve implementation of neighborhood collectives, where they use direct point-to-point send/receive operations to outgoing and incoming neighbors, regardless of the virtual topology, network topology and the underlying hardware. This approach results in poor performance and scalability issues.

In this paper, we propose a new algorithm for neighborhood allgather. We use the virtual topology graph to create a communication pattern to improve latency in neighborhood communication by limiting data exchange with distant ranks and reducing the number of messages. The communication pattern describes the details of the data exchange and memory copy operations in a neighborhood collective.

In our approach, each rank recursively halves the communicator and in each round, finds a rank on the opposite half to offload its outgoing neighbors to that rank, called the *agent* rank. The agents are selected based on a load-aware collaborative mechanism. Each rank tries to select an agent with the maximum number of shared outgoing neighbors on the agent's side. Using this mechanism, the communication with distant nodes is reduced and the load distribution becomes more balanced.

Our contributions in this paper are as follows:

- We propose a new topology- and load-aware algorithm for neighborhood allgather that reduces the communication with distant ranks. This algorithm reduces communication over the network's bottlenecks and improves the communication latency.
- We develop a performance model for the proposed algorithm to show its efficiency mathematically.
- We implemented our algorithm in Open MPI and measured its performance in different scenarios. We compare the performance and scalability of the proposed design with the default implementation of neighborhood allgather in Open MPI as well as with the Common Neighbor algorithm [8] as one of the state-of-the-art designs in the literature.
- Our proposed algorithm achieves average speedups of 1.25x to 8.31x over default Open MPI and 1.41x to 8.13x over the Common Neighbor algorithm for Random Sparse Graphs with densities from 0.05 to 0.7. In Moore

neighborhoods, the average speedup ranges from 1.81x to 8.78x over default Open MPI and 0.62x to 2.57x over the Common Neighbor algorithm. Our algorithm achieves an average speedup of 2.23x over the default Open MPI and 1.73x over the Common Neighbor algorithm for an Sparse Matrix Matrix Multiplication (SpMM) kernel across several matrices.
- We compare the result of the performance model to the experimental results and validate the accuracy of our performance model.

The rest of this paper is organized as follows: Section II provides the necessary background information. Section III reviews the work related to our research. The motivation behind this work is discussed in Section IV. The performance model is presented in Section V. The proposed algorithm is detailed in Section VI, and its evaluation is presented in Section VII. Section VIII concludes the paper and discusses the future work.

II. Background

Global collectives provide only a fixed set of communication patterns, and if a user intends to define a different communication pattern using global collectives, they have to define their mechanisms at the user level using other approaches such as point-to-point communication. This approach could lead to inefficient communications. Using point-to-point communication requires identifying diverse communication patterns within applications and optimizing communication across various systems, hardware configurations, and network topologies. This places a significant burden on programmers, demanding attention to design and development. Additionally, ensuring correctness and preventing deadlocks at the user level can be a tedious job [12].

In global collectives, all processes in the communicator are involved in the communication. This may lead to scalability issues when the number of processes in a communicator increases. The time spent on communication generally scales with $\Theta(n)$, with n the number of processes. The total number of messages scales up with $\Theta(n.log(n))$ [11].

Many HPC applications have sparse communication patterns. In such applications, each process's interactions are restricted to a limited neighborhood, regardless of the total number of processes [24]. If collective communication is used in these applications, all processes are involved in communication, which may slow the execution down because of unnecessary communication. Neighborhood collectives were introduced in MPI 3.0 [6] to overcome these challenges.

The communication pattern of the processes can be described by a graph. The nodes of the graph show the processes and the edges represent the communication. Neighborhood collectives work in conjunction with virtual topologies. Among the various interfaces provided by MPI for specifying the virtual topology graph, `MPI_Dist_graph_create_adjacent` stands out as the most scalable option in which each process defines its incoming and outgoing neighbors.

MPI defines two neighborhood operations: allgather and alltoall. In the allgather operation, every process sends a data segment to its neighboring processes and receives data from its incoming neighbors. However, in the alltoall operation, each process sends distinct data to every outgoing neighbor. The focus of this research is on allgather operation.

III. Related Work

Hoefler and Träff [12] introduced the concept of neighborhood collective communications. Their work demonstrates the advantages of neighborhood collective communications for HPC applications with sparse communication patterns. Kumar et al. [16] used a low-level interface called Multisend for Deep Computing Messaging Framework (DCMF) to implement global and neighborhood collective communications in MPI 2.2. DCMF is an open-source messaging library developed by IBM for the Blue Gene/P machines. Using InfiniBand's network-offload feature, Kandalla et al. [13] introduced scalable designs for nonblocking neighborhood collective operations for 2D Breadth First Search (BFS). They use non-blocking neighborhood collective algorithms to compose global collectives. Unlike these approaches, our algorithm is portable and does not rely on specific hardware features.

Ovcharenko et al. [24] presented a general-purpose communication package that works on top of MPI. It uses the sparsity in the communication pattern of the applications. They show the importance of reducing the number of messages and its impact on the communication time. Hoefler and Schneider [11] discussed the principles of improving the performance of neighborhood collective operations using different approaches. They propose two heuristics for neighborhood alltoall and allgather. They consider applications with a bulk synchronous parallel (BSP) pattern and assume constraints at the user level through `MPI_Info` argument, such as fixed communication channels or message size, to improve the performance of neighborhood collectives. In contrast, our algorithm is not bound to any constraints.

Träff et al. [28], [17], [29], [30] extended the interface of the neighborhood collectives to isomorphic communication patterns using Cartesian topologies. They use message combining to optimize those patterns; however, their approach does not apply to the general graph neighborhoods. Lübbe [18] formulated the performance expectations of neighborhood collectives and virtual topology creation functions as a group of self-consistent performance guidelines. Mirsadeghi et al. [19] and Ghazimirsaeed et al. [8] enhanced the efficiency of neighborhood allgather by employing message combining techniques for small messages. In their approach, a group of K processes with common outgoing neighbors is formed, allowing each process within the group to deliver messages on behalf of others to these neighbors.

Ghazimirsaeed et al. [9] propose a hierarchical and load-aware design to enhance the performance of neighborhood collectives with medium-sized and large messages. Collom et al. [3] present a locality-aware leader-based algorithm for persistent alltoallv that shows up to 1.32 times speedup for an algebraic multi-grid (AMG) solver.

The GPU-aware neighborhood is another direction of research. Khorasani et al. [14] develop an optimized algorithm for GPU-aware neighborhood alltoallv over AMD and Nvidia GPUs. Temuçin et al. [27] develop a mechanism for GPU-aware neighborhood allgather and allgatherv taking advantage of AMD GPUs and networking. Our algorithm is primarily designed for CPU clusters but can be adapted for GPU clusters.

IV. Algorithm Motivation and Overview

As stated earlier, MPI implementations use naïve point-to-point communication for neighborhood collectives. This approach leads to poor performance and scalability, and imbalanced communication. Significant performance enhancements can be achieved by incorporating hardware and network topology into the design of neighborhood collective algorithms.

High-end machines are typically designed based on non-uniform memory access (NUMA) architectures, where minimizing communication between cores across the sockets is an important approach for reducing intra-node communication latency. However, this approach presents significant challenges to the communication library due to the complexities of various hardware platforms. Considering these challenges and opportunities leads us to the question: **How can we design a new algorithm that considers the node architectures for neighborhood collectives?**

From the inter-node perspective, since every hop in the network increases the latency, limiting data exchange with distant nodes is of great importance. The issue is magnified in networks that employ adaptive routing, as non-minimal paths may be chosen to bypass congestion, like universal globally adaptive load-balanced routing (UGAL) in Dragonfly networks [15]. This will be a bigger issue with network topologies with larger diameters like Dragonfly+ networks [26].

Moreover, some network topologies, such as fat-tree and torus, exhibit lower bisection bandwidth relative to injection bandwidth, leading to decreased performance when congestion occurs [25]. Other network topologies suffer from similar problems. For instance, the global links of dragonfly networks are their main bottleneck, and there have been some efforts to reduce the traffic over those links through new routing algorithms [7]. The significant cost associated with network cables, particularly the long inter-cabinet connections limits the global links. This indicates that this problem extends beyond the mentioned topologies, and reducing the communication between distant nodes can reduce congestion and communication latency. This prompts the question: **How can we limit the inter-node communication with the ranks residing on distant nodes in neighborhood collectives?**

Given these considerations, we propose a topology- and load-aware algorithm for neighborhood allgather, by leveraging distance-halving techniques to minimize communication with remote ranks. In this algorithm, each rank recursively divides the communicator into halves and selects an agent in the opposite half to relay its messages. Consequently, communication with distant ranks is substantially reduced, as only one message needs to be sent to the other half. Moreover, this approach decreases the load imbalance among the ranks.

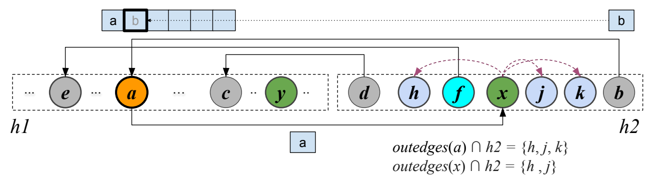
Sack and Gropp [25] have shown how distance halving can use the available bandwidth to achieve better performance compared to other algorithms for global allgather collective. Their algorithm works based on reducing congestion over the bi-section links of the torus and fat-tree networks. Their approach is designed for global allgather that always has a fixed communication pattern. There is no agent and each process exchanges its message with a fixed rank in each step and there is no intra-socket phase. On the contrary, our algorithm is dynamic and designed for graph topologies covering any arbitrary communication pattern. Our algorithm uses the virtual topology to optimize the communication pattern, and agents are selected based on the virtual topology in a distributed and collaborative mechanism.

A. Overview of the Proposed Algorithm

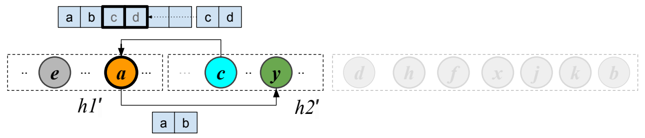
In this section, we provide an overview of the proposed algorithm to develop a performance model that could show its true potential in outperforming the naïve algorithm, for various message sizes and densities of the communication graph. Our algorithm has two phases: the halving phase (inter-socket phase) and the intra-socket phase. The halving phase includes multiple halving steps. Fig. 1 illustrates the proposed approach with a communication scenario where rank a communicates through three halving steps. Ranks x, y, and e are the agents of a in Steps 0, 1, and 2 respectively. a serves as the **agent** of b, c, and e in those steps. We say b, c, and e are the **origins** of a. In this algorithm, each rank maintains a buffer called the main buffer for managing messages exchanged with agents and origins. The variable $main\_buf$ holds a pointer to this buffer. $h1$ is the half that includes the current rank (a) and $h2$ is the opposite half.

In Step 0, a copies its message from the send buffer to the main buffer and sends it to its agent, rank x. At the same time, it receives a message from its origin, rank b, and appends it to the current data in the main buffer. After this step, a does not communicate with any rank in $h2$. Rank x will deliver a's message to a's outgoing neighbors in $h2$, ranks h, j, and k. Based on the location of h, j, and k, and whether they are on the same socket with x or not, rank x sends the message of a to them either through x's agents in the next steps, or directly after the halving phase. k is not an outgoing neighbor of x, but it receives the message of a through x.

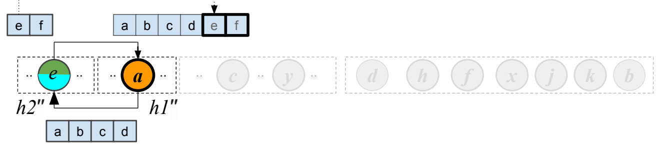
In Step 1, a forwards the content of its main buffer (the data of a and b) to the new agent, rank y, while receiving data from the new origin, rank c. After this step, a has nothing to do with $h2'$. This procedure is repeated in Step 2. Rank a sends its buffer to e which is working as its agent. At the same time, a message is received from e that contains the data of e and f. It includes the data of f since it was the origin of e in Step 0. The size of data in e's buffer does not increase in Step 1, because e does not have an origin in that step. We stop halving



(a) Step 0: x and b are the agent and origin of a, respectively. a sends its message to x relieving itself from its outgoing neighbors in $h2$. However, a must deliver the message of b to b's outgoing neighbors in $h1$.



(b) Step 1: y is the new agent of a and c is its origin.



(c) Step 2: e is the origin and agent of a

Fig. 1: The proposed Distance Halving algorithm for neighborhood allgather, showing the steps involved for process a, having outgoing neighbors h, j, and k in $h2$. Other outgoing neighbors of a are not shown.

when L ranks are left in $h1$. L shows the number of ranks per socket. In the given scenario, ranks in $h1''$ are mostly located on the same socket so there are no more halving steps.

After the halving steps are done, there is an extra intra-socket phase. In this phase, each message in the main buffer is checked against a list to determine which neighbors should receive it. The messages are selectively copied to another buffer and sent to each outgoing neighbor in $h1''$. During this phase, a higher volume of messages may be involved. However, due to the communication being confined within the socket, likely through shared memory mechanisms, message transfer is significantly accelerated compared to the inter-node and inter-socket phases. To evaluate how well this algorithm works, we have developed a performance model that we will elaborate in the following section.

V. Performance Model

We use the Hockney's model [10] to develop our performance model. The Hockney's model is a well-known model and suggests that sending a message of size m between two processes takes $\alpha + \frac{m}{\beta}$ long, where α is the latency for each message, and β is the time it takes to transfer each byte.

Random sparse graphs can represent different virtual topologies. In this approach, graph $G(V, E)$ represents a communication pattern in which each vertex $v \in V$ corresponds to a rank, and each edge $e \in E$ represents an outgoing neighbor of a process. Our performance model works based on the parameter δ ($0 \leq \delta \leq 1$), the same parameter δ in Erdős–Rényi random graph generation model [5]. This parameter shows the probability of each edge being in the graph based on a Bernoulli trial, independent of other edges. We need the size and number of transferred messages to establish our model.

A. Number of Messages

In this section, we calculate the number of messages sent by each rank. Assume we have a communicator with n ranks distributed among computing nodes, each having S sockets with L ranks per socket. Each process recursively halves the ranks until equal or less than L ranks are left.

Consider a random communication topology graph with parameter density $= \delta$. The average number of outgoing neighbors of the ranks is equal to δn and on average δL of them are on the same socket as the source rank is situated. We have $\lceil log(\frac{n}{L})\rceil + 1$ steps of halving. If a rank can find an agent in all steps, it sends $\lceil log(\frac{n}{L})\rceil + 1$ messages in the halving phase. But in some steps, the current rank does not have an outgoing neighbor on the other half, particularly if the graph is too sparse and the number of off-socket outgoing neighbors of the current rank is smaller than the halving steps. Thus, we can find the expected value for the number of messages sent from a rank to out of its socket (n_{off}):

$$\mathbb{E}[n_{off}] = min\left(\lceil log(\frac{n}{L})\rceil + 1, \delta(n - L)\right) \quad (1)$$

When dealing with intra-socket messages, each rank has to deliver a subset of the messages in the main buffer to the other ranks on the local socket. Each rank undergoes $\lceil log(\frac{n}{L})\rceil + 1$ halving steps. Consider two arbitrary ranks a and b on a socket. The probability of b being the outgoing neighbor of neither a nor the origins of a is $(1-\delta)^{\lceil log(\frac{n}{L})\rceil + 2}$. Therefore, the probability of b being at the outgoing neighbor of either a or its origins is equal to $1 - (1-\delta)^{\lceil log(\frac{n}{L})\rceil + 2}$. Since there are L ranks on the local socket, the average number of intra-socket messages (n_{in}) is equal to:

$$\mathbb{E}[n_{in}] = \left(1 - (1-\delta)^{\lceil log(\frac{n}{L})\rceil + 2}\right) L \quad (2)$$

In the worst-case scenario, $\mathbb{E}[n_{in}]$ equals L. The number of messages sent in the naïve algorithm is δn. As an example, consider a cluster with 2000 processor cores, distributed among 50 nodes, each with 40 cores over two sockets. If we run a neighborhood allgather collective with a virtual topology graph with $\delta = 0.3$, each rank in the Distance Halving algorithm sends on average 23 (7 off-socket + 16 intra-socket) messages. In comparison, the naïve algorithm sends 600 messages on average. By increasing δ, the average number of messages sent in the Distance Halving algorithm will not exceed 27 messages. In contrast, the naïve algorithm can potentially send as many messages as the size of the communicator.

B. Message Size

Starting from the size of the primary messages m, in the worst-case scenario, the size of the message is doubled in each halving step. For intra-socket messages, after $\lceil log(\frac{n}{L})\rceil + 1$ halving steps, on average δL intra-socket outgoing neighbors are left; however, the current process must deliver the messages of the origins to their outgoing neighbors on the local socket. The probability of each rank on the current socket being one of the outgoing neighbors of the origins is equal to δ and independent of other ranks. The number of incoming neighbors of each rank on the local socket between the origins of the current rank follows the binomial distribution of $B(\delta, \mathbb{E}[n_{in}])$. Given that, the expected size of intra-socket messages (m_{in}) can be calculated as:

$$\mathbb{E}[m_{in}] = \delta \mathbb{E}[n_{in}] m \tag{3}$$

C. Constructing the Performance Model

We assume the network is single port, and only one message can be sent or received at a time. In the naïve algorithm, δn messages with size m are sent in a row. The number of incoming messages is also equal to δn. Based on the Hockney's Model, the expected communication time for one rank using the naïve algorithm ($t_r(\textit{naïve})$) is:

$$\mathbb{E}[t_r(\textit{naïve})] = 2\delta n \left(\alpha + \frac{m}{\beta}\right) \tag{4}$$

Under the single port assumption, the messages of the ranks on a node are serialized over the network. We do not distinguish the inter-node, intra-node, and intra-socket bandwidth for simplicity. Since there are SL ranks on a node, the expected value of the total collective time for the naïve algorithm ($t(\textit{naïve})$) is:

$$\mathbb{E}[t(\textit{naïve})] = SL\mathbb{E}[t_r(\textit{naïve})] \tag{5}$$

We find the latency for the Distance Halving algorithm. Considering the worst-case scenario, the size of the messages is doubled in each halving step. We calculate based on the worst-case scenario because the collective operation is considered completed when all ranks, including those lagging behind, have completed their work. So the expected value for the off-socket communication time for one rank ($t_{off}(\text{DH})$) is:

$$\begin{aligned}\mathbb{E}[t_{off}(\text{DH})] &= \left(\alpha + \frac{m}{\beta}\right) + \left(\alpha + \frac{2m}{\beta}\right) \\ &\quad + \dots + \left(\alpha + \frac{(2^{\mathbb{E}[n_{off}]})m}{\beta}\right) \\ &= \mathbb{E}[n_{off}]\alpha + \frac{(2^{(\mathbb{E}[n_{off}]+1)} - 1)m}{\beta}\end{aligned} \tag{6}$$

For the intra-socket messages:

$$\mathbb{E}[t_{in}(\text{DH})] = \mathbb{E}[n_{in}] \left(\alpha + \frac{\mathbb{E}[m_{in}]}{\beta}\right) \tag{7}$$

The time spent sending messages equals $\mathbb{E}[t_{off}(\text{DH})] + \mathbb{E}[t_{in}(\text{DH})]$. The time spent on receiving the messages is the same. Similar to the naïve algorithm, the messages are serialized; thus the total communication time is:

$$\mathbb{E}[t(\text{DH})] = 2SL\left(\mathbb{E}[t_{off}(\text{DH})] + \mathbb{E}[t_{in}(\text{DH})]\right) \tag{8}$$

We compared the performance of these two algorithms using the performance model shown in Fig. 2. This figure highlights the potential of the new algorithm and its ability to outperform

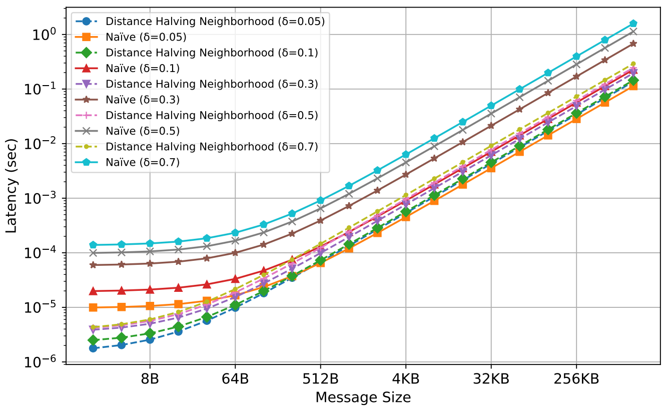


Fig. 2: Performance modeling of the Distance Halving algorithm vs. the naïve algorithm

the naïve algorithm across various message sizes and densities in Random Sparse Graphs. The calculations are based on parameters obtained from ping-pong tests conducted on the Niagara cluster. Additional details about the Niagara cluster are provided in Section VII.

VI. Proposed Algorithm

The proposed algorithm consists of the communication pattern creation routines and the neighborhood collective operation. The communication pattern illustrates how messages are transferred when the neighborhood collective is called. This pattern is built after the creation of the virtual topology graph and is attached to the communicator along with the topology graph. The communication pattern creation routines are executed once, whereas the routines for the neighborhood collective operation are triggered every time `MPI_Neighbor_allgather` is called.

A. Terminology

A summary of the symbols and variables is provided in Table I to facilitate understanding of this section. The parameters $h1$ and $h2$, introduced in Section IV-A, are defined as follows: $h1$ denotes the half that includes the current rank, while $h2$ represents the opposite half. The parameters n and L, discussed in Section V, are defined as: n represents the size of the communicator, and L indicates the number of ranks per socket. The variable p denotes the rank of the current process. The set I represents the incoming neighbors, with $indegree$ indicating its size. The set O includes the outgoing neighbors, with $outdegree$ representing its size.

C denotes the list of candidate agents/origins, where a candidate is a rank sharing at least one outgoing neighbor with the current rank. The shared outgoing neighbors of the candidates and the current rank are documented in matrix A, similar to matrices used in [8]. The matrix structure is illustrated in Fig.3, where $A[i][j] = 1$ indicates that $O[i]$ is an outgoing neighbor of $C[j]$. O_{on} and O_{off} are the outgoing neighbors loaded onto and offloaded from the current rank, respectively. I_{on} includes the incoming neighbors who

$$A = \begin{array}{cccccc} O[0] & O[1] & & \dots & O[outdegree\text{-}1] & \\ 1 & 0 & 1 & \dots & 0 & \rightarrow C[0] \\ 1 & 1 & 0 & \dots & 1 & \rightarrow C[1] \\ & & \vdots & & & \vdots \\ 0 & 1 & 0 & \cdots & 1 & \end{array}$$

Fig. 3: Matrix A. Each rank has a different matrix. Each row represents a candidate agent/origin, and each column corresponds to the outgoing neighbors of the current rank. If $A[i][j]$ equals 1, it shows that outgoing neighbor j ($O[j]$) is also an outgoing neighbor of $candidates[i]$.

have not offloaded the current rank and the agents of other incoming neighbors. O_{org} are the outgoing neighbors of an origin to whom the current rank delivers messages. D is a descriptor assisting agents in message delivery. This descriptor is represented as a key-value map where the keys are the new origin and the new origin's previous origins. For each key, the value is a list of its outgoing neighbors that are in $h1$. The variable O_{org} has a structure similar to D.

TABLE I: Description of symbols in the algorithm

symbol	Description
n	size of $comm$
L	number of ranks on the local socket
$h1$	the half that includes p
$h2$	the opposite half ($p \notin$h2)
p	rank of current process
I	incoming neighbors of the current rank(inedges)
O	outgoing neighbors of the current rank(outedges)
$outdegree$	size of O
$inedegree$	size of I
C	candidate agents/origins
A	Matrix A
O_{on}	onloaded outgoing neighbors
O_{off}	outgoing neighbors offloaded to an agent
I_{on}	agents of my inedges $\cup$ remaining inedges
O_{org}	outgoing neighbors of origins
D	a descriptor to guide agents
$sbuf$	send buffer
$rbuf$	receive buffer
m	size of the data in $sbuf$

Algorithm 1: building the communication pattern

```
1  **Input:** $comm$, $I$, $indegree$, $O$, $outdegree$, $L$, $p$, $n$
2  **Output:** $communication\_pattern\ cp$
3  **Function** $build\_cp$:
4      $A, C \leftarrow$ calculate_A($comm$,$I$, $indegree$, $O$, $outdegree$)
5      $h1 \leftarrow [0, n-1]$
6      $h2 \leftarrow \phi$
7      $O_{on} \leftarrow O$
8      $O_{off} \leftarrow \phi$
9      $I_{on} \leftarrow I$
10     $t \leftarrow 0$
11     **while** $t \leq \lceil log(\frac{n}{L}) \rceil + 1$ **do**
12         $step \leftarrow new\ step()$
13         $mid\_rank \leftarrow \frac{(h1.start + h1.end)}{2}$
14         **if** $p \leq mid\_rank$ **then**
15             $h1 \leftarrow [h1.start\ ...\ mid\_rank]$
16             $h2 \leftarrow [mid\_rank + 1\ ...\ h1.end]$
17             $step.agent \leftarrow find\_agent(comm,\ A,\ h1,\ h2,\ C)$
18             $step.origin \leftarrow find\_origin(comm,\ A,\ h1,\ h2,\ C)$
19         **else**
20             $h2 \leftarrow [h1.start\ ...\ mid\_rank]$
21             $h1 \leftarrow [mid\_rank + 1\ ...\ h1.end]$
22             $step.origin \leftarrow find\_origin(comm,\ A,\ h1,\ h2,\ C)$
23             $step.agent \leftarrow find\_agent(comm,\ A,\ h1,\ h2,\ C)$
24         **end**
25         $O_{off} \leftarrow \phi$
26         **if** $agent\ found$ **then**
27             $O_{off} \leftarrow O_{on} \cap h2$
28             $O_{on} \leftarrow O_{on} - h2$
29         **end**
30         send/receive notification to/from $I$ and $O$
31         $D \leftarrow \phi$
32         $D[p] \leftarrow O_{off}$
33         **if** $t > 0$ **then**
34             **for** $origin\ o \in cp.steps[t-1].origins$ **do**
35                 **if** $agent\ found$ **then**
36                     add $o$ to $step.origins$
37                     $list1 \leftarrow (cp.steps[t-1].O_{org}[o]) \cap h2$
38                     $list2 \leftarrow (cp.steps[t-1].O_{org}[o]) - list1$
39                     $D[o] \leftarrow list1$
40                     $step.O_{org}[o] \leftarrow list2$
41                 **else**
42                     $step.O_{org}[o] \leftarrow cp.steps[t-1].O_{org}[o]$
43                 **end**
44             **end**
45         **end**
46         **if** $agent/origin$ found **then**
47             send/receive $D$ to/from $step.agent/step.origin$
48         **if** $origin$ found **then**
49             add received $D$ to $step.O_{org}$
50         add $step$ to $cp.steps$
51         $t$++
52     **end**
53     **return** cp
54 **end**
```

B. Communication Pattern

The communication pattern creation routines consists of two sections. The main function that creates the communication pattern and the joint mechanism for choosing the agents.

1) Main function

The main function for creating the communication pattern is shown in Algorithm 1. This function is executed when `MPI_Dist_Graph_create_adjacent` is called. The function takes I, O, $indegree$, $outdegree$, L, and $comm$ as inputs and returns the communication pattern. All ranks within the communicator invoke this function to build their communication patterns cooperatively.

First, all ranks generate matrix A and initialize other variables. The $while$ loop in Line 11 shows the main part of the algorithm. Each iteration of this loop adds a new step to the communication pattern. t shows the step number. In Lines 13 to 24, $h1$ and $h2$ are updated. The current $h1$ is recursively divided into two halves and assigned to $h1$ and $h2$.

If the current rank is in the lower half (the half that has smaller ranks), it calls the $find\_agent$ function (Line 17) when its candidate origins on the upper half are calling $find\_origin$ in Line 22. After that, all the ranks in the upper half call the $find\_agents$ function in Line 23. At the same time, all ranks in the lower half call the $find\_origin$ function in Line 18. $find\_agent$ and $find\_origin$ are two dual functions used to find the agents/origins. These functions are explained in Section VI-B2.

The communication pattern must be updated when the agent and origins are determined. As mentioned earlier, finding an agent may fail. In Line 26, we check whether a new agent is found or not. If an agent is found, O_{off} is updated to the outgoing neighbors in $h2$. These outgoing neighbors are removed from O_{on} in Line 28. Then we have to notify our outgoing neighbors about the selected agent. We send the agent to those ranks. At the same time, we receive notifications from all incoming neighbors in $h2$ to see whether they offloaded the current rank to another rank or not.

So far, the outgoing neighbors in $h2$ are notified about the selected agent, however, the agent does not know the outgoing neighbors of the current rank. So we send O_{off} which includes the outgoing neighbors of the current rank in $h2$ to it. However, the current rank has more responsibilities concerning the ranks in $h2$. It may need to deliver the messages of the origins of the previous steps to some ranks in $h2$. We offload all of those ranks to the new agent. Therefore, we have to notify the agent about the previous origins and their outgoing neighbors. So we create object D, pack all this information in it, and send it to the agent. This procedure is shown in Lines 31 to 49. For each origin o in the origins of the previous step, we save the outgoing neighbors of o which are in $h1$ in O_{org} in the communication pattern.

In each step, we remove the outgoing neighbors of origins that are offloaded to the new agent ($list1$ in Line 37) and add them to D which is going to be sent to the new agent. Then the outgoing neighbors of those origins are updated ($list2$ in Line 38) and copied to O_{org} in the new step in Line 40. If no agent was found we copy O_{org} of the previous to the new step in Line 42. Finally, the current step is added to the communication pattern Line 50. We exit the loop when less than or equal to L ranks are left in $h1$. This means all of the ranks in $h1$ are located on the current socket and we don't need to continue halving. After the loop, the communication pattern is ready and is returned in Line 53.

2) *Agent Selection*

When a rank aims to select another rank as its agent, the target rank must accept to act as the agent since we do not want to put too much load on the agent. The agents are selected in a joint selection mechanism containing two steps. In the first step, all ranks in the first half call $find\_agent$ while the ranks in the other half call $find\_origin$ function to respond to the messages sent by the first half. In the next step, all ranks in the second half call $find\_agent$ while the first half runs $find\_origin$. These functions are shown in Algorithms 2 and 3 respectively.

Assume rank a halves the ranks into two subsets, $h1$ and $h2$ while $a \in h1$. a chooses a rank that shares the highest number of common outgoing neighbors in $h2$. All ranks in $h2$ with at least one common outgoing neighbor with a in $h2$ are considered candidates. If a chooses b in a step, it is not

Algorithm 2: Agent Selection

```
1  Input: $comm$, $A[][]$, $h1$, $h2$, $C$
2  Output: $selected\_agent$
3  Function $find\_agent$:
4      $find\_new\_agent \leftarrow true$
5      $agent \leftarrow -1$
6      $c_r \leftarrow 0$ // Received Signals
7      $c_s \leftarrow 0$ // Sent Signals
8      $c_t$ // Total number of Signals
9      $status\_agents[\,]$ , $signal$, $sender$
10     update $status\_agents$
11     $c_t \leftarrow 2 \times$ number of ACTIVE agents
12     while $c_r + c_s < c_t$ do
13         if $find\_new\_agent$ then
14             $agent \leftarrow get\_best\_agent(A, p, h2, status\_agents)$
15             if $agent\ found$ then
16                 send REQ to agent
17                 $c_s$++
18             end
19             $find\_new\_agent \leftarrow false$
20         end
21         if $c_s + c_r == c_t$ then break
22         recv($signal$, MPI_ANY_SOURCE, $comm$, $status$)
23         $c_r$++
24         $sender \leftarrow status$.MPI_SOURCE
25         if $signal$ == ACCEPT then
26             $find\_new\_agent \leftarrow$ false
27             $selected\_agent \leftarrow agent$
28             $status\_agents[sender] \leftarrow$ SELECTED
29             send EXIT to all ACTIVE agents
30         else if $signal$ == DROP then
31             if $sender$ == $agent$ then
32                 $find\_new\_agent \leftarrow$ true
33             else
34                 send EXIT to sender
35                 $c_s$++
36             end
37             $status\_agents[sender] \leftarrow$ INACTIVE
38         end
39     end
40     return $selected\_agent$
41 end
```

guaranteed that b chooses a in that step. We do this to achieve better performance in imbalanced communication patterns.

$status\_agents$ in Algorithm 2 keeps the status of the candidates. In Line 10 the status of the candidates that do not share any outgoing neighbor with the current rank in $h2$ is set to INACTIVE, as the current rank does not need to communicate with those ranks. The while loop in Lines 12 to 39 represents the main operation of the function. In this loop, the current rank communicates with all active candidates. It has to send REQ or EXIT signal to all candidates in $h2$ and receive an ACCEPT or DROP signal in reply. c_t and c_r track the number of sent and received signals respectively. These variables are updated every time a message is sent or received.

The $agent$ variable shows the rank of the current best agent that we are communicating with. $find\_best\_agent$ function returns the best candidate which is the active candidate that has the maximum number of common outgoing neighbors with the current rank in $h2$. If no agent is found, the output of that

Algorithm 3: Origin Selection

```
1  Input: $comm$, $A[][]$, $h2$, $h1$, $C[]$
2  Output: $selected\_origin$
3  Function $find\_origin$:
4      $update\_best\_origin \leftarrow true$
5      $c_r \leftarrow 0$ // Received Signals
6      $c_s \leftarrow 0$ // Sent Signals
7      $c_t$ // Total number of Signals
8      $status\_origins[\,]$, $signal$, $sender$
9      $selected\_origin \leftarrow -1$
10     update $status\_origins$
11     $c_t \leftarrow 2 \times$ number of ACTIVE origins
12     while $c_r + c_s < c_t$ do
13         if $update\_best\_origin$ then
14             $best\_origin \leftarrow get\_best\_origin(A$,p $h1$, $h2$,
                 $status\_origins)$
15             if $best\_origin \neq -1$ and
                 $status\_origins[best\_origin]$ == WAITING then
16                 send ACCEPT to $best\_origin$
17                 $c_s$++
18                 $status\_origins[best\_origin] \leftarrow$ SELECTED
19                 $selected\_origin \leftarrow best\_origin$
20                 send DROP to all ACTIVE/WAITING origins
21                 update $c_s$
22             end
23             $update\_best\_origin \leftarrow false$
24         end
25         if $c_s + c_r$ == $c_t$ then break
26         recv($signal$, MPI_ANY_SOURCE, $comm$, $status$)
27         $c_r$++
28         $sender \leftarrow status$.MPI_SOURCE
29         if $signal$ == REQ and $selected\_origin \neq -1$ then
30             if $sender$ == $best\_origin$ then
31                 send ACCEPT to $best\_origin$
32                 $c_s$++
33                 $selected\_origin \leftarrow best\_origin$
34                 $status\_origins[sender] \leftarrow$ SELECTED
35                 send DROP to all ACTIVE/WAITING origins
36                 update $c_s$
37                 $update\_best\_origin \leftarrow$ false
38             else
39                 $status\_origins[sender] \leftarrow$ WAITING
40             end
41         else if $signal$ == EXIT then
42             if $status\_origins[sender]$ == ACTIVE and
                 $selected\_origin \neq -1$ then
43                 send DROP to $sender$
44                 $c_s$++
45                 if $sender$ == $best\_origin$ then
46                     $update\_best\_origin \leftarrow$ true
47                 end
48             end
49             $status\_origins[sender] \leftarrow$ INACTIVE
50         end
51     end
52     return $selected\_origin$
53 end
```

function is -1 which shows there is no active candidate left and finding an agent has failed. Otherwise, a request signal is sent (Line 16) and we have to wait to receive a reply from the target rank who is running the $find\_origin$ function. The reply is either an ACCEPT or DROP signal. These signals are received through the receive request posted in Line 22. Since we are not aware of the sender of the next signal, MPI_ANY_SOURCE is used.

A received ACCEPT signal indicates that the target rank has agreed to be the agent of the current rank in this step. In this case, we notify the other ranks in $h2$ that they do not receive a REQ signal from the current rank by sending an EXIT signal in Line 29. DROP signal means that the agent has denied the request and we have to find another agent, therefore we change the status of that agent to INACTIVE and set $find\_new\_agent$ to $true$ and the negotiation to a new candidate is started in the next iteration of the loop.

The $find\_origin$ function shown in Algorithm 3 is similar to $find\_agent$. A process communicates with all active origins in its opposite half while running this function. The main part of the function is the loop in Lines 12 to 51. In Line 14, the rank of the best origin is found using $find\_best\_origin$, similar to the $find\_best\_agent$ function in Line 14 of Algorithm 2. The best origin is a rank in $h2$, having the maximum number of common outgoing neighbors with the current rank in $h1$.

If the best origin is WAITING, an ACCEPT signal is sent to notify it that the current rank accepts to operate as its agent (Line 16). Then a DROP signal is sent to all ACTIVE and WAITING origins. If the best origin is not WAITING, the current rank has to wait for a signal from it. A request is posted in Line 26 to receive signals from other ranks. REQ signals are handled in Lines 29 to 40. If the sender process is the best origin, it means that we received the signal we were waiting for and we reply with an ACCEPT signal in Line 31. Otherwise, we change the sender's status to WAITING in Line 39 and wait for the next incoming signal.

EXIT signals, handled in Lines 41 to 50, reveal that the sender never chooses the current rank as its agent since it has already selected another rank. In this case, the sender's status changes to INACTIVE. If the sender process is the best origin, we were awaiting a signal from, the best origin is updated by changing $update\_best\_origin$ to $true$ in Line 46. The best origin is updated in the next iteration of the loop.

C. Neighborhood Operation

Section VI-B shows how a communication pattern is created. In this section, we show how the communication pattern is used to perform the neighborhood communication. Algorithm 4 shows this operation. The inputs are $sbuf$, $rbuf$, m, and $comm$, representing the send buffer, receive buffer, size of the message in $sbuf$, and the communicator, respectively. This algorithm has two parts: the halving phase and the intra-socket phase. Lines 3 to 18 show the halving phase, where $sbuf$ is initially copied to $main\_buf$ to be sent to the agents. The loop starting at Line 5 shows the actual halving steps. In each iteration, $main\_buf$ is sent to the agent while simultaneously receiving and merging the $main\_buf$ from the origin into the local $main\_buf$. If the incoming message consists of a message from the incoming neighbors of the current rank, it is copied to the $rubf$ in Line 16.

The intra-socket phase, spanning Lines 19 to 33, requires

Algorithm 4: MPI_Neighbor_allgather

```
1  Input: $sbuf$, $rbuf$, $m$, $comm$
2  communication_pattern $cp \leftarrow comm.cp$
3  copy $sbuf$ to $main\_sbuf$
4  $d \leftarrow m$
5  foreach $step$ in $cp.steps$ do
6      $d_{old} \leftarrow d$
7      foreach origin $o$ in $step.origins$ do
8          irecv from $o$ in $d$-th byte of $main\_buf$
9          $d \leftarrow d$ + sizeof($incoming\ message$)
10     end
11     if $step$ has $agent$ then
12         isend first $d_{old}$ bytes of $main\_buf$ to $step.agent$
13     end
14     wait_all
15     foreach origin $o \in step.origins \cap I$ do
16         copy data of $o$ from $main\_buf$ to $rbuf$
17     end
18 end
19 $step \leftarrow cp.last\_step$
20 $i \leftarrow 0$
21 foreach $outgoing\ rank\ r$ in $step.O_{on}$ do
22     foreach origin $o$ in $step.origins$ do
23         if $r \in step.O_{org}[o]$ then
24             copy data of $o$ in $main\_buf$ to $buf_{temp}[i]$
25         end
26     end
27     isend $buf_{temp}[i{+}{+}]$ to $r$
28 end
29 foreach incoming rank $r$ in $step.I_{on}$ do
30     irecv from $r$ in $buf_{temp}[i{+}{+}]$;
31 end
32 wait_all
33 copy received messages to $rbuf$
```

the information of the last step of the communication pattern. In this phase, we have the data of multiple origin ranks stored in $main\_buf$ and O_{org} indicates the destinations of each message. We copy the message from the $main\_buf$ to a temporary buffer and send it to the destination ranks, shown in Lines 21 to 28. At the same time, we receive the messages from other ranks on the socket and copy them to the receive buffer of the current rank (Lines 29 to 31).

VII. Performance Evaluation and Analysis

We implemented our proposed Distance Halving neighborhood algorithm in Open MPI v5.0.0rc12 and compiled it with UCX v1.15.0. We compare the performance of our algorithm against the default Open MPI implementation and the Common Neighbor algorithm [8], using the Random Sparse Graphs and Moore neighborhood microbenchmarks, and a neighborhood Sparse Matrix Matrix Multiplication (SpMM) kernel. The Common Neighbor algorithm is one of the state of the art algorithms in literature. We launched the Common Neighbor algorithm with various values of K. We report the best results in this study.

We ran our experimental studies on the Niagara cluster, in the Digital Research Alliance of Canada, consisting of 2024 nodes. Each node features 40 CPU cores distributed over two sockets, running CentOS 7. The cores are either Intel Skylake at 2.4 GHz or Cascade Lake at 2.5 GHz. The cluster utilizes a DragonFly+ topology with Adaptive Routing over EDR InfiniBand. Each node has either 188 GB or 202 GB of RAM.

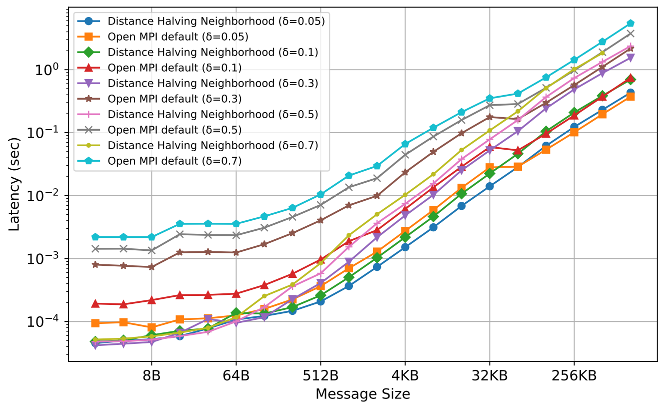


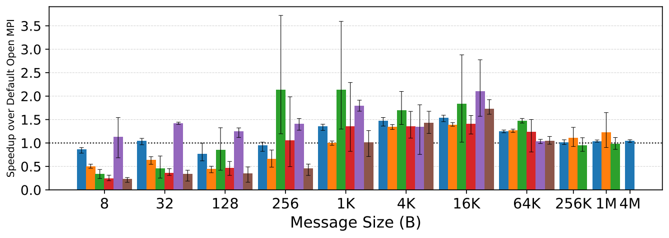
Fig. 4: Latency of the proposed algorithm for Random Sparse Graphs with various densities and message sizes against the default Open MPI algorithm for 2160 ranks over 60 nodes

A. Random Sparse Graph

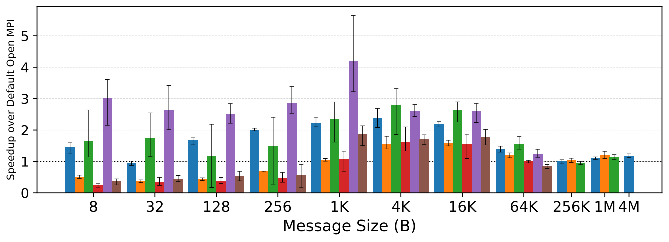
We use the Erdős–Rényi model to generate Random Sparse Graphs. This model is also used in [8], [9], and [27]. Fig. 4 compares the latency of our algorithm against the default Open MPI for various message sizes and graph densities. Each node was configured with 36 ranks, reserving 4 cores per node for the operating system to minimize system noise. The results confirm the validity of our performance model in Section V. The difference between the absolute values can be due to system noise, the accuracy of the Hockney's model, and congestion. In most cases for messages smaller than 64KB, our algorithm experiences a lower latency, particularly for dense graphs. The performance is on par with Open MPI for 64KB messages and above, and in some cases outperforms it.

Fig. 5 illustrates the speedup of the proposed algorithm compared to the Common Neighbor algorithm and the default implementation of Open MPI, across various densities and message sizes ranging from 8 bytes to 4 megabytes. The experiments were conducted with 2160, 1080, and 540 ranks distributed over 60, 30, and 15 nodes, respectively. The results demonstrate that the proposed algorithm outperforms the Common Neighbor algorithm and the Open MPI in many scenarios, particularly with higher graph densities, such as $\delta = 0.5$ and $\delta = 0.7$, where the speedups peak at nearly 30x over the default Open MPI for 32B messages with $\delta = 0.7$. These highlight the capability of the proposed algorithm to efficiently leverage denser graphs to minimize the communication time between distant ranks.

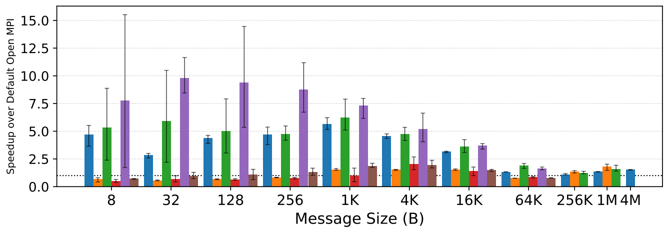
In lower densities and smaller numbers of ranks, communication is rather limited, offering limited opportunity for enhancement. Therefore, the speedups are more modest compared to denser graphs. The proposed algorithm, nevertheless, demonstrates up to 3.5 times speedup at lower graph densities of 0.05 and 0.1, much higher than what the Common Neighbor algorithm can achieve. For large messages exceeding 256KB,



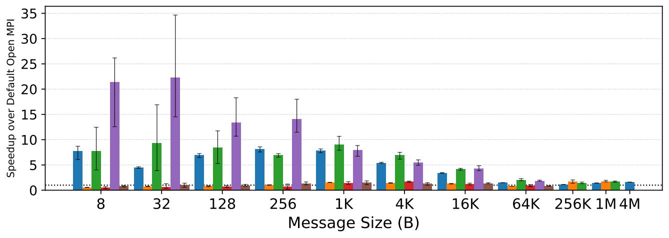
(a) δ = 0.05



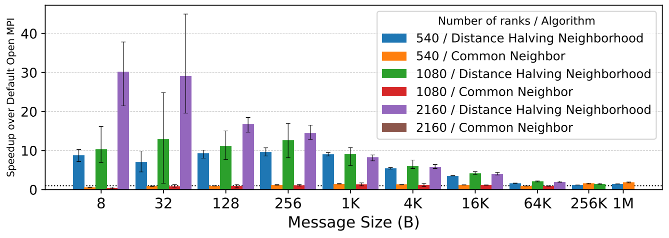
(b) δ = 0.1



(c) δ = 0.3



(d) δ = 0.5



(e) δ = 0.7

Fig. 5: The scaling of the speedup of the proposed algorithm compared to the Common Neighbor algorithm over the default Open MPI for Random Sparse graphs with various densities (δ) and message sizes.

the performance of the algorithm declines, possibly due to the overhead from additional memory copy operations and increased network congestion. This problem is amplified in Random Sparse Graphs because the outgoing neighbors are uniformly distributed in the communicator and the chance of the message size being doubled in each step is high. Our study shows an 80% average success rate across all ranks in finding an agent with $\delta = 0.05$. This shows, on average, the message size is doubled in 80% of steps. This reveals that the load is balanced. In other words, the Random Sparse Graph shows the worst-case scenario for our algorithm. However, in the Moore neighborhood study in Section VII-B, we show that our algorithm achieves better performance for larger messages, since the outgoing neighbors are not uniformly distributed, and the proposed algorithm helps in limiting communication with distant nodes and balancing the communication.

Overall, the average speedup of the proposed algorithm across all message sizes is between 1.25x and 8.31x over default Open MPI, and from 1.41x to 8.13x over the Common Neighbor algorithm for Random Sparse Graphs with densities ranging from 0.05 to 0.7.

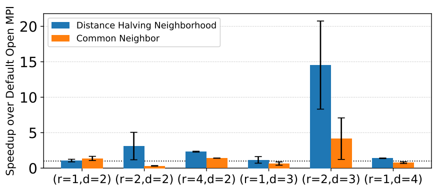
B. *Moore Neighborhood*

Moore neighborhood is a form of neighborhood defined by two parameters: r and d. The processes are located on a d dimensional virtual grid. Each rank communicates with all ranks within a maximum distance of r in the grid. The number of neighbors in a Moore neighborhood is $(2r+1)^d - 1$. Moore neighborhoods offer structured and balanced topologies, presenting regular patterns.

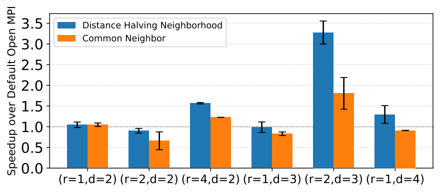
We evaluated the speedup of the Distance Halving and Common Neighbor algorithms over the default Open MPI. Fig. 6 shows the speedup for 2048 processes over various Moore neighborhoods and small, medium, and large message sizes. The experiments were performed on configurations of 64 nodes with 32 ranks per node. As shown in Fig. 6a, the proposed algorithm can reach up to 14x speedup over the default algorithm. The results show our algorithm outperforms other algorithms by up to 3x speedup for medium-sized messages with more dense neighborhoods. For small messages, our algorithm shows a high level of speedup when the neighborhoods are more dense and there is room for improvement. The high variance shown in some cases is due to the high variance in the communication time of the default algorithm, and its impact can also be seen in the Common Neighbor algorithm. The experiments were repeated multiple times, and each time different nodes are assigned to the job. This reveals that the default algorithm is sensitive to the distance of the nodes and experiences varying communication times across different configurations. In contrast, our algorithm is considerably more stable in communication time.

C. *Sparse Matrix Matrix Multiplication Kernel*

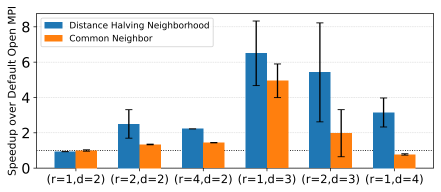
Sparse Matrix Matrix Multiplication Kernel (SpMM) is an important kernel in computational linear algebra, big data analytics, and graph algorithms [1]. This kernel calculates the multiplication of two matrices X and Y to produce matrix $Z = X \times Y$. Matrices X and Y are distributed among processes in a block-stripped row-wise and column-wise fashion. `MPI_Neighbor_allgather` is used to gather matrix Y to each process. At the end of the computation, each process has a block-stripped row-wise portion of matrix Z. We used various matrices from The SuiteSparse Matrix Collection (formerly the University of Florida Sparse Matrix Collection) [4] with different sizes and sparsity, shown in Table II.



(a) 4KB



(b) 256KB



(c) 4MB

Fig. 6: The speedup of the proposed algorithm against common neighbor algorithm over default Open MPI with 2048 ranks for various Moore neighborhoods and small (4KB), medium (256KB), and large (4MB) message sizes

TABLE II: Sparse matrices and their sizes

Matrix	Size	Non-zero Elements
dwt__193	193 × 193	1843
Journals	128 × 128	6096
Heart1	3600 × 3600	1387773
ash292	292 × 292	2208
bcsstk13	2003 × 2003	83883
cegb2802	2802 × 2802	277362
comsol	1500 × 1500	97645

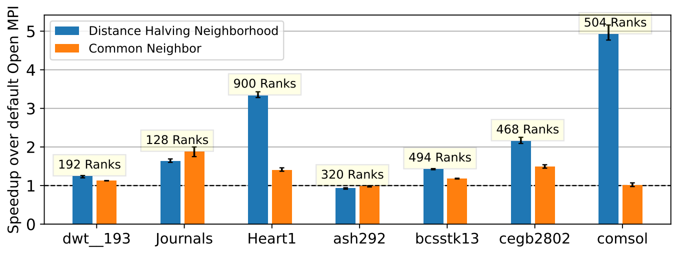


Fig. 7: The speedup of the proposed algorithm and Common Neighbor algorithm over default Open MPI for SpMM for input matrices in Table II.

Fig. 7 shows that the performance of our algorithm is superior over the Common Neighbor approach in most cases. Our algorithm achieves up to 3.33x and 4.92x speedup against the default algorithm for Heart1 and comsol, respectively. cegb2802 and bcsstk13 are large matrices, but they are more sparse compared to Heart1 and comsol. ash292 is a small matrix and too sparse that does not leave too much room for improvement, even for the Common Neighbor algorithm. Overall, the proposed algorithm shows 0.93x to 4.92x speedup.

D. *Overhead Analysis*

In this section, we analyze the overhead of the proposed algorithm by measuring the communication pattern creation time for the Distance Halving and Common Neighbor algorithms for Random Sparse Graphs for 2160 ranks with various densities. As shown in Fig. 8, the overhead of the proposed algorithm is around 20%-50% more than the Common Neighbor algorithm. However, this is a one-time overhead, and considering the speedup achieved by the new algorithm, this overhead is amortized after multiple neighborhood calls. The major part of the overhead comes from the agent selection routines. In the worst-case scenario, each rank attempts to select every other rank on other sockets as its agent, and all

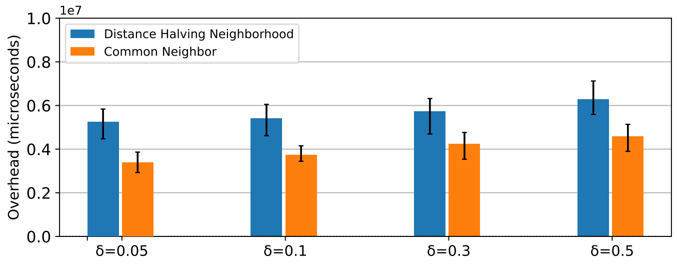


Fig. 8: The overhead of the proposed algorithm against the Common Neighbor algorithm in Random Sparse Graphs with various densities (δ) and 2160 ranks.

ranks try to choose that rank as their agent. Considering one request and one response message on each side, each pair of ranks, located on different sockets communicate 4 messages. The total number of messages exchanged in the agent selection routines is equal to $\frac{4n(n-L)}{2} \in O(n^2)$ messages.

VIII. Conclusion and Future Work

Our research presents a new approach to enhancing communication performance in MPI by proposing a new neighborhood allgather algorithm. The proposed algorithm efficiently reduces the communication latency by limiting interactions with distant ranks. Our performance model demonstrates the performance benefits of the proposed design. Our experimental study showed the performance and scalability of our algorithm, achieving up to 30x, 14x, and 4.93x speedup over Open MPI, for Random Sparse Graph, Moore neighborhood, and SpMM, respectively. These findings highlight the practical relevance and effectiveness of our approach in optimizing communication performance for modern HPC clusters.

As for future work, we intend to focus on improving the agent selection process to enhance the performance of sparse virtual topologies with large message sizes. We would also like to extend our approach to alltoall and other variants of these neighborhood collectives.

IX. Acknowledgement

This research was supported in part by the Digital Research Alliance of Canada. Computations were performed on the Niagara cluster at the SciNet HPC Consortium. SciNet is funded by the Canada Foundation for Innovation, the Government of Ontario, Ontario Research Fund - Research Excellence; and the University of Toronto.

References

[1] S. Acer, O. Selvitopi, and C. Aykanat, "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems," *Parallel Computing*, vol. 59, pp. 71–96, 2016, theory and Practice of Irregular Applications.

[2] 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.

[3] 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.

[4] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, dec 2011.

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

[6] M. 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] M. García, E. Vallejo, R. Beivide, M. Odriozola, and M. Valero, "Efficient routing mechanisms for Dragonfly networks," in *42nd International Conference on Parallel Processing*, 2013, pp. 582–592.

[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] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Computing*, vol. 20, pp. 389–398, 1994.

[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] 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.

[13] 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.

[14] 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.

[15] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-Driven, Highly-Scalable Dragonfly Topology," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 3, pp. 77–88, 2008.

[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] 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.

[18] 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.

[19] 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.

[20] "MPI Forum," 2024. [Online]. Available: https://www.mpi-forum.org/

[21] "MPICH," 2024. [Online]. Available: https://www.mpich.org/

[22] "MVAPICH," 2024. [Online]. Available: https://mvapich.cse.ohio-state.edu/

[23] "Open MPI," 2024. [Online]. Available: https://www.open-mpi.org/

[24] 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.

[25] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 45–54.

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

[27] 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.

[28] 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.

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

[30] 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.