Exploiting Common Neighborhoods to Optimize MPI Neighborhood Collectives

Seyed H. Mirsadeghi*, Jesper Larsson Träff†, Pavan Balaji‡ and Ahmad Afsahi*
*ECE Dept., Queen's University, Kingston, ON, Canada, K7L 3N6, {s.mirsadeghi, ahmad.afsahi}@queensu.ca
†Vienna University of Technology (TU Wien), Faculty of Informatics, Vienna, Austria, traff@par.tuwien.ac.at
‡Mathematics and Computer Science Division, Argonne National Laboratory, Lemont, IL, U.S., 60439, balaji@anl.gov

***Abstract*—Neighborhood collectives were added to the Message Passing Interface (MPI) to better support sparse communication patterns found in many applications. These new collectives encourage more scalable programming styles, and greatly extend the scope of MPI collectives by allowing users to define their own collective communication patterns.**

In this paper, we describe a new, distributed algorithm for computing improved communication schedules for neighborhood collectives. We show how to discover common process neighborhoods in fully general MPI distributed graph topologies, and how to exploit this information to build message-combining communication schedules for the MPI neighborhood collectives. Our experimental results show considerable performance improvements for application communication topologies of various shapes and sizes. On average, the performance gain is around 50%, but it can also be as much as 71% for topologies with larger numbers of neighbors.

***Keywords*-MPI; topology; neighborhood collective**

I. Introduction

The Message Passing Interface (MPI) [1] is the most widely used parallel programming paradigm in HPC. MPI provides different types of communication including point-to-point and collective operations. Collective operations are widely used in parallel applications because they provide a convenient, yet highly optimized way to conduct complex communications.

The standard, global MPI collectives suffer from certain limitations. They have inherent scalability issues because they entail dependencies on all processes in the given communicator. This can be especially problematic at exascale where some collective patterns (e.g., all-to-all) can become too costly to be practical. In addition, MPI collectives capture only a fixed set of predefined communication patterns (e.g., broadcast, all-to-all). Other, desired patterns must be manually implemented by the programmer or mimicked by (ab)using irregular, global collectives (alltoallw).

To address these shortcomings, the MPI-3.0 standard introduced *neighborhood collective* communication. The specific communication pattern of a neighborhood collective is defined by a virtual MPI process topology. More specifically, each process will communicate only with processes that are defined as outgoing/incoming neighbors in the virtual topology associated with the communicator. Thus, neighborhood collectives allow users to define their own communication patterns via the MPI process topology interface. Neighborhood collectives vastly extend the concept of collective communications, and are one of the most advanced features of MPI. Neighborhood collective communication can be more scalable by restricting communications to the local process neighborhoods [2]. They provide support for *sparse* communication patterns [3] found in many applications such as Nek5000 [4] and Qbox [5].

The virtual topology information associated with a communicator can be exploited to derive better communication patterns for neighborhood collectives in advance. Current MPI libraries such as MPICH [6], MVAPICH [7], and Open-MPI [8], however, use only straightforward approaches for neighborhood collective communication where processes simply issue non-blocking send and receive operations to and from its outgoing and incoming neighbors. Thus, no knowledge of the topology is used to govern the communications in order to deliver better performance.

In this paper, we show how to improve the performance of neighborhood collectives by analyzing the virtual topology and deriving non-trivial communication schedules. We make no assumptions about the structure of the communication topology, and our methods work regardless of how the topology was created. In particular, we can handle distributed graph topologies which provide the most flexible and generic specification of process topologies in MPI [2]. We show how to discover and exploit *common neighborhoods* in such topologies to improve the performance of neighborhood collectives. Specifically, we propose a distributed algorithm that extracts message-combining communication patterns and constructs schedules for neighborhood collectives. To the best of our knowledge, this is the first work that shows how common neighborhoods and message-combining can be used to optimize the performance of neighborhood collectives as found in MPI.

Our approach can be used for all of the MPI neighborhood collectives; due to space limitations we present actual results only for `MPI_Neighbor_allgather` here; results for alltoallv can be found in our technical report [9]. Our experimental results show that we can decrease the latency of allgather for topologies of various shapes and sizes. The improvements are mainly around 50% but can be as much as 71%. We also provide a detailed analysis of the overheads and discuss how they are affected by factors such as the total number of processes and the neighborhood topology.

II. Related Work

Hoefler and Träff [3] argue for the importance of adding sparse collective operations to MPI and propose three such operations, which form the basis for what was ultimately added to MPI as the neighborhood collectives [1]. Ovcharenko et al. [10] highlight the existence of sparse neighbor communications in many parallel applications and propose a library on top of MPI to support them. Hoefler et al. [2] explore the enhancements to the MPI process topology interface, in particular distributed graph topologies that significantly improve on scalability, informativeness, and user friendliness over the older topology interface. Non-blocking neighborhood collectives were used by Kandalla et al. [11] to redesign the BFS algorithm in the Combinatorial BLAS library [12].

Kumar et al. [13] show how the multisend interface of the IBM Deep Computing Message Framework (DCMF) [14] can be used to optimize applications that exhibit neighborhood collective communication patterns. The multisend interface allows multiple send operations to be bundled into a single call that can be handled by the direct memory access engine. This approach does not change the communication pattern; each process still sends a message to each of its neighbors. Our optimization is orthogonal, and can exploit the multisend benefits on our derived patterns.

Hoefler and Schneider [15] discuss general optimization principles for neighborhood collectives. They show how to use graph coloring to design communication schedules that avoid hotspots at the end processes. They also propose an algorithm for balancing communication between high-outdegree processes and those having lower outdegrees. Their optimizations are orthogonal to our proposed design. Moreover, balancing the communication tree is beneficial for unbalanced neighborhood topologies only, whereas our optimizations apply to both balanced and unbalanced neighborhoods. This is important because many real applications use balanced neighborhood topologies.

Träff et al. [16] discuss *isomorphic neighborhoods* in which all processes have the same neighborhood structure. They propose interfaces for defining such neighborhoods in MPI that impose lower overheads than the generic graph interface. In another work [17], Träff et al. develop message-combining algorithms for isomorphic neighborhoods and show that they can improve the performance of neighborhood communications. However, the proposed algorithms are limited to isomorphic neighborhoods. Furthermore, they depend on new interfaces that are not yet available in MPI. Our proposed approach works for the MPI interfaces in their full generality.

III. Overview

A. Design Principles

We focus mainly on small communication problems ($\leq$ 4KB) and aim at decreasing the *number of communication stages* in the neighborhood collectives. The following two general principles can be used to decrease the number of communication stages in a given collective operation:

1) Increase the number of senders in each stage;
2) increase the number of messages transferred in each send operation.

These two principles can be found at the core of many standard algorithms for conventional collectives [18]. However, optimizing neighborhood collectives is more challenging. Conventional collectives describe *global* communication patterns over *all* processes, and all processes locally possess this global, structural information. In contrast, the communication pattern for a neighborhood collective is described by a graph that is distributed over the processes, with only local structural information available to each process.

For a neighborhood collective, the first principle mentioned above can help improve performance when the processes have an *unbalanced* number of neighbors. More specifically, we can increase the number of simultaneous senders in each stage by delegating a portion of the communications from processes that have a high number of outgoing neighbors to those having lower outdegrees. This strategy helps avoid situations where one process is overloaded with a lot of messages to send while some other processes are idle. The tree transformation algorithm discussed by Hoefler and Schneider [15] is an example of such an optimization.

In this paper, we apply the second principle to improve the performance of the neighborhood collectives. More specifically, we are interested in designing non-trivial communication algorithms that exploit message-combining to decrease the number of communication stages of the neighborhood collectives. Although such an optimization can be used in conjunction with the neighbor-balancing approach, it becomes particularly useful in cases where neighbor balancing fails to provide any benefits, for instance, for balanced process topologies where the number of outgoing neighbors of all processes is about the same. The fact that most applications expose a balanced communication topology adds to the importance of our approach.

B. Common Neighborhoods

We aim to apply message-combining to neighborhood collectives by exploiting *common neighborhoods* that may exist in the process topology graph. We define the common neighborhood of processes p_1 and p_2 as the set of all processes that are outgoing neighbors of both p_1 and p_2. The existence of common neighborhoods in a topology graph provides an opportunity for decreasing the number of (combined) messages to be sent by each process.

To clarify, consider the sample topology graph shown in Fig. 1, where p_1 and p_2 both have the processes $n_1, n_2, \ldots, n_k$ as a part of their outgoing neighbors. In a trivial design, p_1 and p_2 each send one message to each of $n_1, n_2, \ldots, n_k$ neighbor processes, which adds up to k communication stages with two receive operations per neighbor. We can reduce the number of stages drastically by dividing the common neighborhood between p_1 and p_2. Process p_1 sends messages from itself and p_2 to half of the common neighbors, e.g., $n_1, n_2, \ldots, n_{\frac{k}{2}}$, while p_2 sends messages from itself and p_1 to the other half of

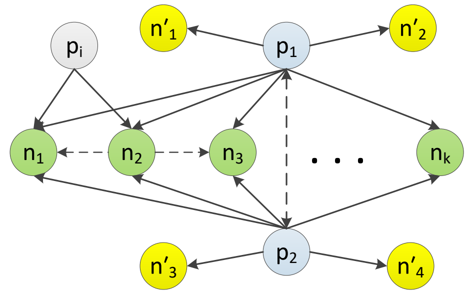


Fig. 1: Part of a general communication topology graph. Processes $n_1, \ldots, n_k$ are *common neighbors* of processes p_1 and p_2. Note that p_1 and p_2 may be neighbors of each other and there might be some edges among the common neighbors. Process p_i shares some neighbors with p_1 and p_2. Some processes that are neighbors of only p_1 or p_2 are also shown.

the common neighbors, $n_{\frac{k}{2}+1}, n_{\frac{k}{2}+2}, \ldots, n_k$. This reduces the number of stages to $\frac{k}{2}+1$, with each of the common neighbors receiving from only one process (not two). To accomplish this, p_1 and p_2 need to identify each other, and exchange half of their messages with the other process. We call processes with common neighborhoods like p_1 and p_2 *friends*.

We now present our algorithm to discover common neighborhoods among the processes and exploit them to derive more efficient communication schedules for neighborhood collectives. Our design has two phases:

1) Build a message-combining *communication pattern* from the given topology graph;
2) build an actual *communication schedule* from the derived pattern and the specific neighborhood collective.

IV. Communication Pattern Extraction

The first phase is the main part of our design, in which we build a non-trivial communication pattern among the processes based on pairwise message-combining between processes that share a common neighborhood. The resulting communication pattern describes a series of message-combining steps for each process. We note that this phase depends only on the given topology, and thus needs to be done only *once* for each new process topology. In MPI, this phase is done at the time the virtual topology is created, and adds no overhead to the actual neighborhood collective communication. Moreover, the result from this phase can be used for all of the neighborhood collectives (allgather and all-to-all etc.).

A. Distributed Design

A major challenge is that we desire a *fully distributed* solution and build communication patterns that locally describe the specific communications of each involved process. A distributed design potentially has lower overheads and better scalability. Moreover, we want to stick to the distributed representation of the topology provided by the MPI distributed graph topology functions. Thus, we want to avoid centralized solutions that require gathering the complete topology graph at one or more processes. Our approach works with both the adjacent and the non-adjacent distributed graph interface; all we need is the MPI query functions that for each process returns its in- and outgoing neighbors.

B. Message-Combining Algorithm

In order to describe our message-combining algorithm, we say that two processes are *friends* of each other if they have at least Θ outgoing neighbors in common for some chosen threshold value Θ. Note that two friends may or may not be neighbors themselves.

Alg. 1 gives the main steps of our algorithm. Each process p in the topology graph runs Alg. 1 to build its local portion of the desired communication pattern. First, we extract information about common neighborhoods of p which we store in a matrix M (Line 1). We explain this step in detail in Section IV-B1. From M, we find all friends of p with respect to the given threshold Θ (Line 2). We also initialize O_a and I_a (Line 3), which denote the list of *active* outgoing and incoming neighbors of p, respectively. Active neighbors represent a subset of neighbors that have not been dealt with yet. After that, the main loop (Lines 4 to 25) iteratively performs three main tasks: (1) pair p with one of its friends p' for the purpose of message-combining, (2) divide the corresponding common neighbors between p and its paired friend p', and (3) update topology information and the output pattern.

More specifically, p first attempts to find a friend p' to pair with (Line 6). This is a major step, and we discuss it in Section IV-B2. Having found p', we extract the set of outgoing neighbors that p and p' have in common (Line 8) and divide it evenly between p and p' (Lines 10 to 17). Let CN be the set of common neighbors between p and p'. We divide CN into two balanced subsets CN_{on} and CN_{off}, which respectively denote the neighbors assigned (onloaded) to p and the neighbors offloaded to p'. A key point here is that we have to make sure the division is consistent between p and p'. In other words, CN_{on} and CN_{off} must fulfill the following conditions:

1) $CN_{\text{on}} \cap CN_{\text{off}} = \emptyset$
2) $CN_{\text{on}} \cup CN_{\text{off}} = CN$
3) CN_{on} at p = CN_{off} at p'
4) CN_{off} at p = CN_{on} at p'

To ensure these conditions, we sort the list of common neighbors according to their MPI ranks (Line 9). We divide the list in two halves, and we decide which half to assign to p and p' based on their MPI ranks. For instance, if $p < p'$ all the neighbors in the first half of the sorted CN are assigned to p, and vice versa.

At this point, we have built two stages of our target communication pattern for p (and also p'). Process p exchanges its offloaded messages with p', builds a combined message, and sends it to the processes in CN_{on}, namely, the common outgoing neighbors assigned (onloaded) to p. We save the communication steps with p' and CN_{on} in the output pattern $\mathcal{T}$ (Line 18).

Algorithm 1: Computing a distributed message-combining pattern.

```
Input  : For process $p$, set of outgoing neighbors $O$, set of incoming
         neighbors $I$, friendship threshold $\Theta$
Output: The communication pattern $\mathcal{T}$
M = Build_common_neighborhood_matrix($O$, $I$);
$F$ = Find_friends($M$, $\Theta$);
$O_a = O$, $I_a = I$;
while $|F| > 0$ do
    $CN = \emptyset$;
    $p'$ = Find_friend_to_pair($M$, $F$);
    if found $f$ then
        $CN$ = Find_common_neighbors($M$, $p'$);
        sort($CN$);
        if $p < p'$ then
            Keep (onload) the first half of $CN$;
            Offload the second half of $CN$ to $p'$;
        end
        else
            Keep (onload) the second half of $CN$;
            Offload the first half of $CN$ to $p'$;
        end
        Add $p'$ and $CN_{\mathrm{on}}$ to $\mathcal{T}$;
    end
    Notify each neighbor in $O_a$ whether it was onloaded/offloaded;
    Update $I_a$ and $\mathcal{T}$ based on notifications from neighbors in $I_a$;
    $O_a = O_a - CN$;
    Update_common_neighborhood_matrix($M$, $O_a$, $I_a$);
    $F$ = Find_friends($M$, $\Theta$);
end
Notify each neighbor in $O_a$ that $p$ is done;
Add $O_a$ to $\mathcal{T}$;
while $I_a \neq \emptyset$ do
    Update $I_a$ and $\mathcal{T}$ based on notifications from neighbors in $I_a$;
end
```

At the end of each iteration, we notify each active outgoing neighbor of p about the outcome of the current iteration. This notification (Line 20) is used for three purposes:

1) Informing offloaded neighbors that they will not receive any message from p
2) Informing onloaded neighbors that they will receive a combined message including the data from p and p'
3) Updating the list of active incoming neighbors of each process (Line 21)

We also update the list of active outgoing neighbors to remove those covered in the current iteration (Line 22). Then we update the common neighborhood matrix based on the remaining active outgoing/incoming neighbors of each process (Line 23), and recompile the list of remaining friends accordingly (Line 24). The details of how we update the common neighborhood matrix are discussed in Section IV-B3. The condition in Line 4 ensures that the main loop of the algorithm finishes when there are no more remaining friends for the process to pair with. The process then notifies its remaining active outgoing neighbors that it is done (Line 26) and adds them to $\mathcal{T}$ (Line 27). The process remains active to receive notifications sent from its active incoming neighbors (Line 29). The notifications are used to update both the pattern ($\mathcal{T}$) and the set of active incoming neighbors (I_a).

1) Finding Common Neighborhoods: The first step to building our desired pattern is finding the common neighborhoods in the topology graph. More specifically, each process has to find the set of all other processes with which it has some of its outgoing neighbors in common. Fortunately, this can be nicely done in an MPI distributed graph topology by a set of neighborhood communications. To this end, each process queries each of its *outgoing* neighbors about their *incoming* neighbors. For each outgoing neighbor n_{out} of a given process p, the incoming neighbors of n_{out} represent the set of all processes with which p has n_{out} as a common neighbor. By querying all its outgoing neighbors, p builds a common neighborhood matrix M. Each row of M corresponds to one of the outgoing neighbors of p such as n_i, and it lists the set of all incoming neighbors of n_i. Therefore, each element in row n_i represents the rank of a process with which p has n_i in common as an outgoing neighbor. By exploring all the elements in matrix M, we can find all the processes with which p has at least one neighbor in common. Also, the number of times a process rank appears within M designates the number of neighbors that p and the given process have in common (the common neighborhood size). Hence, we can find all friends of p with respect to a given friendship threshold Θ.

Each process already has the list of all its outgoing and incoming neighbors, which makes the extraction of M easy. Every process in the topology needs only to send its own list of (already known) incoming neighbors to each of its incoming neighbors and receive one such list from each of its outgoing neighbors. We note that M requires $O(\Delta^2)$ memory space, where Δ is the degree of the process topology graph.

2) Pairing Friend Processes: In each iteration of the loop in Alg. 1, each process p attempts to pair with one (and only one) of its friend processes (Line 6). Among all possible friends, we want p to choose a friend with which it has the highest number of common neighbors to reduce the number of communication stages as much as possible. Friend selection must be mutual between the two processes involved. If p chooses p' as its target friend to pair with, then p' must also have chosen p as its target friend.

The friend-pairing problem can be modeled as a *distributed maximum weighted matching problem* in the corresponding *friendship graph* of processes $G(V, E)$. For each process p, we have a vertex $v_p \in V$, and two vertices v_p, v_q are adjacent ($(v_p, v_q) \in E$) if and only if p and q are friends. The edge weight represents the number of common neighbors between p and q. Finding a maximum matching in the friendship graph G pairs each process with at most one other friend and does this pairing so that the total number of common neighbors covered is maximized. However, G is inherently distributed among the processes, since each process extracts information only about its own neighborhood and friends (Lines 1 and 2 of Alg. 1). Thus, we have a distributed maximum weighted matching (MWM) problem.

Various algorithms have been proposed for the distributed MWM problem [19]–[22]. Our algorithm similar to that of Hoepman [20]. A recent study [23] shows that his algorithm outperforms other major alternatives for distributed MWM. Alg. 2 gives the details of our algorithm for distributed

Algorithm 2: Distributed friend pairing/matching.

```
   Input  : For process $p$, common neighborhood matrix $M$, Set of friends $F$
   Output: A friend process $p'$ to pair with for message-combining
1  paired = terminal = False;
2  while !(paired OR terminal) do
3      if first iteration then
4          $p' = p'_{\text{old}}$ = friend with the maximum number of common neighbors;
5      end
6      else
7          if $p'_{\text{old}}$ is not terminal AND is not paired AND $p < p'_{\text{old}}$ then
8              $p' = p'_{\text{old}}$; // Retry the previous choice
9          end
10         else
11             $p' = g \in F : g < p$ and $g$ chose $p$;
12             if not found $g$ then
13                 $F = F - p'_{\text{old}}$ ;
14                 if $F \neq \emptyset$ then
15                     terminal = True ; // failed to pair with anyone
16                 end
17                 else
18                     $p'$ = friend with maximum number of common neighbors;
19                 end
20             end
21         end
22     end
23     Notify all friends in $F$ about $p'$;
24     if $p$ is not terminal AND $p'$ chose $p$ then
25         paired = True ; // Successfully paired with $p'$
26     end
27     Notify all friends in $F$ about paired state;
28     Remove from $F$ all the paired and terminal friends;
29 end
```

matching of friend processes. The main differences between Alg. 2 and Hoepman's algorithm are in the way we choose the candidates in each iteration and the information communicated among the processes. We note that Alg. 2 is one step (Line 6) of our main design outlined in Alg. 1.

Alg. 2 is an iterative algorithm where each process first chooses a potential friend to pair with and then checks to see whether its choice is mutual. If it is, then we have a successful pairing, and we are done. Otherwise, the process goes to the next iteration to try another (or the same) friend again. A given process p runs the main loop in Lines 2 to 29 until it reaches either a *paired* or *terminal* state. The former represents the case where p can successfully find a mutual friend, whereas the latter represents the case where p fails to find a mutual friend and gives up the search. Friend selections happen in Lines 3 to 22. The remaining steps are used for communicating the choices to check their mutuality.

In the first iteration of the loop, each process p greedily chooses the friend with which it has the maximum number of common neighbors. In all other iterations, we consider two main cases: (1) p sticks to its previously chosen friend p'_{old} (Lines 7 to 9), or (2) p tries one of its other still-active friends (Lines 10 to 21). As shown by the condition in Line 7, p sticks to its previously chosen friend p'_{old} only if it finds that p'_{old} has also failed to pair with its previously chosen friend. The rationale is that in such a case, p'_{old} will still be looking for a friend to pair with and it might choose p this time. Thus, it makes sense for p to stick to p'_{old}. However, p does so only if (1) p'_{old} has not become terminal and hence is still looking for a friend, and (2) the rank of p is lower than the rank of p'_{old}. This second condition is needed to avoid deadlock situations where the set of friends chosen by a group of processes creates a circular dependency. Otherwise, such processes will all keep sticking to their previously chosen friend, causing them all to fail indefinitely.

In choosing a new friend, we first look for a friend g such that rank of g is lower than the rank of p, and g has chosen p as its potential friend (Line 11). The reason is that we know from the first case above that g will choose p again as its potential friend and, hence, p and g could successfully pair. If g does not exist, we attempt to choose a new friend with which p has the highest number of common neighbors (Line 18). However, we first discard the previously chosen friend from the list of friends in Line 13, and we put the process in the *terminal* state if there are no more friends left to try. Doing so guarantees a bounded number of iterations when a process fails to mutually pair with a friend.

At the end of each iteration, we have two communication steps. In the first one (Line 23), p sends (receives) the value of p' to (from) all its remaining friends. In the second one (Line 27), p informs its friends about its success/failure in mutual pairing based on the information received in the first communication step. Next, we update the list of friends to remove all the paired and terminal ones. Such friends will not be active in the next iteration of the algorithm.

3) Updating Neighborhood Information: At the end of each iteration of Alg. 1, we update the neighborhood information before moving to the next iteration. This step is necessary because the neighbor offloading/onloading changes the effective neighborhoods of a process. More specifically, when a process such as p offloads some of its outgoing neighbors to a friend process p', it no longer is communicating (directly) with those neighbors, thus effectively canceling those processes as outgoing neighbors of p. In our current implementation, each outgoing neighbor of a process is considered for at most one message-combining stage; that is, the onloaded neighbors of a process are not considered for message-combining with other friends in further iterations of the algorithm. Thus, before choosing another friend for message-combining, we first need to update the common neighborhood matrix with respect to such changes in effective outgoing neighbors.

We can do this update in a way similar to that discussed in Section IV-B1 for building the common neighborhood matrix in the first place. Each process queries each of its outgoing neighbors about their remaining active incoming neighbors. To this end, each process sends its own list of active incoming neighbors (I_a) to each of its active incoming neighbors and receives one such list from each of its active outgoing neighbors. Each received list is compared with its corresponding row in the common neighborhood matrix to mask those elements that are no longer in the list.

4) Complexities: The complexity of Alg. 2 can be given by $\mathcal{O}(rF)$, where r denotes the number of iterations of the algorithm and F the maximum number of friends of a process (degree of the friendship graph). This leads to a *worst-case* complexity of $\mathcal{O}(p^2)$ for Alg. 2, where p is number of processes. The reason is that in the worst case, $r = p - 1$ due to Lines 7 to 9 of Alg. 2, and $F = p - 1$ for a complete friendship graph.

The complexity of Alg. 1 can be given by $\mathcal{O}(t(rf + \Delta^2))$, where t denotes the number of iterations of the algorithm and Δ the degree of the topology graph. The Δ^2 term is for traversing the neighborhood matrix to find the friends of a process. Assuming an upper bound of F for t (we do not have a formal proof for it), the *worst-case* complexity of Alg. 1 will be $\mathcal{O}(p(p^2 + p^2)) = \mathcal{O}(p^3)$ with a complete topology graph.

We note that these are worst-case complexities. In Section VI-C, we show that in practical use cases, Alg. 1 overhead increases with $\sqrt{F}(F + \Delta^2)$. Moreover, Δ can be expected to be much lower (constant) than p for the topology graphs used with neighborhood collectives.

V. Communication Schedule Construction

So far, we have explained how to extract an improved, *message-combining communication pattern* from the underlying process topology graph. We now explain how to build a *communication schedule* from the extracted pattern for each given neighborhood collective function. The communication schedule precisely specifies the send, receive, and memory copying operations that a process should perform for the derived communication pattern. Since buffer addresses, counts and datatypes are available only with the actual neighborhood collective call, unlike the pattern, the schedule *cannot* be built in advance. It is therefore important that schedule construction is fast, given the computed pattern information from the first phase.

Alg. 3 shows the steps involved in building the communication schedule. Again, each process p runs the algorithm to build the desired schedule. As the input, it takes the message-combining pattern $\mathcal{T}$ and all the parameters corresponding to a specific neighborhood collective call, including send/receive buffer addresses and message sizes (counts). The output is a communication schedule $\mathcal{S}$ to be executed by the MPI runtime. Each step of the schedule captures a series of nonblocking send/receive operations to be issued by each process.

Accordingly, the loop in Lines 1 to 19 processes the pattern $\mathcal{T}$ one step at a time. Each step of $\mathcal{T}$ contains the results of the corresponding iterations of Alg. 1, which can include two items: (1) A friend process p' to combine messages with and a corresponding set of onloaded outgoing neighbors, and (2) a set of incoming neighbors to receive messages from. If $\mathcal{T}$ contains a message-combining friend p' at step t, we add to the schedule a send (receive) operation to (from) p' for exchanging messages between p and p' (Lines 3 and 4). Next, we add a hint to the schedule (Line 5) to note that the succeeding operations should not be started until all the previous ones are completed. Then, we add an operation to

Algorithm 3: Communication schedule extraction from the message-combining pattern.

```
Input : For process $p$, communication pattern $\mathcal{T}$, send/recv buffers,
        send/recv sizes, send/recv datatypes, MPI communicator
Output: Communication schedule $\mathcal{S}$
1  for each step $t$ in $\mathcal{T}$ do
2      if have a combining friend $p'$ then
3          $\mathcal{S} \leftarrow$ send operation to $p'$;
4          $\mathcal{S} \leftarrow$ recv operation from $p'$;
5          $\mathcal{S} \leftarrow$ wait on issued operations;
6          $\mathcal{S} \leftarrow$ build the combined message;
7          for each onloaded neighbor $n_{\mathrm{on}}$ do
8              $\mathcal{S} \leftarrow$ send operation to $n_{\mathrm{on}}$ (combined message);
9          end
10     end
11     for each incoming neighbor $n_i$ tagged with $t$ do
12         $\mathcal{S} \leftarrow$ recv operation from $n_i$ to get a message (possibly
           combined);
13     end
14     $\mathcal{S} \leftarrow$ wait on issued operations;
15     for each incoming neighbor $n_i$ tagged with $t$ do
16         $\mathcal{S} \leftarrow$ copy operation to move the message contents to the
           final buffers;
17     end
18     $\mathcal{S} \leftarrow$ wait on issued operations;
19 end
20 for each remaining outgoing neighbor $n_o$ do
21     $\mathcal{S} \leftarrow$ send operation to $n_o$;
22 end
23 Repeat Lines 11 to 17 for the remaining incoming neighbors;
```

build the desired combined message, which in our current implementation translates into a memory copy operation. After that, we add a send operation (Lines 7 to 9) to transfer the combined message from p to each of its onloaded outgoing neighbors.

Lines 11 to 17 of Alg. 3 process the list of incoming neighbors from which p should receive a message at step t. For each of such incoming neighbors, we add a receive operation to $\mathcal{S}$ to receive a message into an intermediate buffer (Line 12). In Line 16 we add a memory copy operation to $\mathcal{S}$ to move data into the final application buffers. First, however, we add a hint to $\mathcal{S}$ (Line 14) to make sure that such data movements are not started before the messages arrive in the intermediate buffers. Before processing the next step t, we add another hint to $\mathcal{S}$ (Line 18) to block any further operations until the current step completes. Finally, we add a send (recv) operation to $\mathcal{S}$ for each remaining outgoing (incoming) neighbor that has not been covered in the previous steps of $\mathcal{T}$ (Lines 20 to 23). These operations constitute a final step of communications and correspond to Lines 27 to 30 of Alg. 1. Alg. 3 has a complexity of $\mathcal{O}(\Delta)$, where Δ is the degree of the topology graph.

VI. Experimental Results

In this section, we evaluate the performance of our design for various neighborhood topologies. We use a microbenchmark to measure the latency of neighbor allgather with and without our message-combining approach. In our microbenchmark, we first build a desired process topology graph and then call `MPI_Neighbor_allgather`. The allgather function is

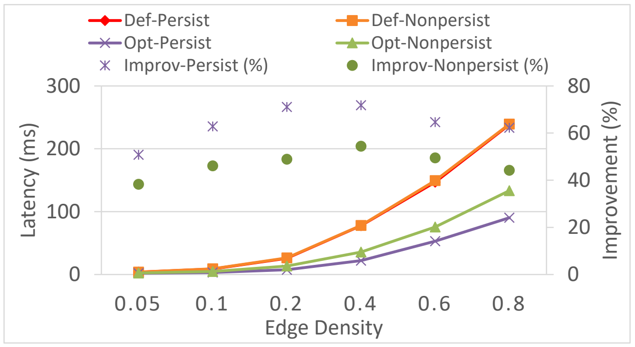


Fig. 2: Neighbor allgather average latencies for the random sparse graph topology—4K processes, 4B message size.

called 1,000 times and the average latency per call is reported as the output. A friendship threshold of $\Theta = 4$ is used in all the experiments as it represents the minimum number of common neighbors that could potentially lead to performance improvement through message-combining. The experiments are conducted on the GPC cluster at the SciNet HPC Consortium [24]. GPC consists of 3780 nodes, each having two quad-core Intel Xeon sockets operating at 2.53GHz, and a total of 16GB memory per node. The nodes run CentOS 6.4 and we use MVAPICH2-2.2. Approximately one quarter of the cluster is connected with DDR InfiniBand while the rest uses QDR InfiniBand. For our experiments, we only use the QDR partition. Due to the limited availability of the cluster, we could only run our experiments with up to 8K cores.

We evaluate the performance for two flavors of our design and the default naive approach: *Persistent* and *nonpersistent*. The persistent approach is useful in cases where the neighborhood collective function parameters (such as buffer addresses, counts, etc.) are known to remain unchanged. Thus, in the persistent approach, the neighborhood collective schedule is built *once* in the first call to the collective (or in an "`init`" API) and is reused for all succeeding calls. In the nonpersistent approach, the schedule is built every time that the neighborhood collective is called. Note that the communication pattern is built only once in both cases. We report the results for four cases:

1) Def-Nonpersist: The naive approach in MVAPICH
2) Opt-Nonpersist: Our proposed design
3) Def-Persist: The persistent version of the naive approach
4) Opt-Persist: The persistent version of our design

A. Random Sparse Graph

In our first set of experiments, the neighborhood topology is on an Erdös-Renyi random sparse digraph $G(V, E)$. The set of vertices V represents the set of MPI processes, and an edge $(i, j) \in E$ represents an outgoing neighbor from process i to process j. The edges of the graph are created randomly with density factor $\delta, 0 < \delta \leq 1$ which determines the number of outgoing neighbors per process (higher δ means denser graph). The same type of random sparse graphs were also used by Hoefler and Schneider [15].

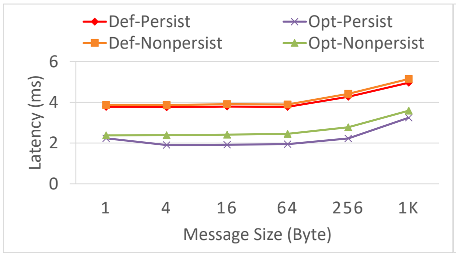
We conduct experiments for different values of δ. This allows us to evaluate our design with neighborhood topologies of random shapes and different sizes. For each concrete value of δ, we run our microbenchmark 5 times to account for the randomness of the topology graph, and report the average.

Fig. 2 shows the results with 4K MPI processes and 4 B message size. It shows the absolute latency values (left axis) as well as the improvement percentages (right axis). We can see that our proposed design results in lower latencies compared with those with the naive approach used in MVAPICH. In general, we see greater improvements for denser graphs, as expected because denser graphs provide a larger number of neighbors per process, which in turn provide more room for improvements through our proposed approach. But as Fig. 2 shows, even at a low edge density of 0.05, we can still achieve about 50% improvement. The improvements can be as high as 70%. Another observation is that while the persistent feature does not impact the default approach noticeably, it can improve the performance of our design. This is because building a schedule out of the naive pattern is much easier (with almost no overhead) than building it from the message-combining pattern. Thus the persistent feature has higher impacts for the latter. However, as we discuss in Section VI-C, the schedule overhead is still low for our design, and hence both the persistent and nonpersistent versions of our design outperform the default approach.

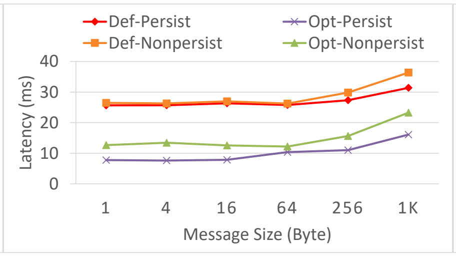
In Fig. 3, we show the results across various message sizes and for three edge densities. We can see lower latencies achieved by our proposed design across all message sizes up to 1 KB. The improvements vary from 36% (for $\delta = 0.05$ and 1 KB message size) to 71% (for $\delta = 0.2$ and 4 B message size), but in most cases we can achieve about 50% reduction in latency. The improvement gap starts to shrink after 1 KB message size because of bandwidth effects. For clarity, we do not show the results for message sizes above 1 KB here. Those results show that all four approaches converge more or less to the same latency at around a 4 KB message size. The exact convergence point varies a little for different edge densities and happens at larger messages ($\approx$ 16 KB) for denser graphs. After that, the naive approach leads to lower latencies. This result is expected because message-combining does not improve the bandwidth terms of the underlying communication pattern. Thus, it does not have much benefit for large messages, which are sensitive mainly to bandwidth/congestion characteristics of communication patterns. We emphasize that our goal from the outset was to improve the performance for small messages; large messages require a different design as they have a different optimization objective. Another observation is that for our design, the persistent approach provides lower latencies across all message sizes. Even the nonpersistent version, however, still outperforms the naive approach.

B. Moore Neighborhoods

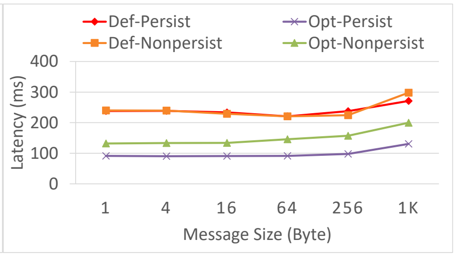
For our second set of experiments, we use a set of Moore neighborhood as the process topology graph. A Moore neighborhood is defined by two parameters: Dimension D and



(a) Edge density $\delta = 0.05$



(b) Edge density $\delta = 0.2$



(c) Edge density $\delta = 0.8$

Fig. 3: Neighbor allgather average latencies for the random sparse graph topology—4K processes, various message sizes.

radius R. The former represents the number of the grid dimensions that the nodes (MPI processes) are organized into, and the latter represents the absolute value of the maximum relative (Manhattan) distance at which other nodes are considered a neighbor of a given node. Thus, for each pair of D and R values, the number of neighbors of each node is $(2R+1)^D - 1$. Moore neighborhoods are symmetric in the sense that each neighbor is both an outgoing and an incoming neighbor of a given MPI process.

Moore neighborhoods allow us to study the benefits of our design with more regular types of (balanced) topologies. They can be seen as as a generalizations of stencil patterns such as 2D 9-point or 3D 27-point. Moore neighborhoods have a high number of neighbors per process. They allow modeling of a wider variety of neighborhood shapes and sizes through different values of D and R. Moore (and von Neumann) neighborhoods were used by Träff et al. [16], [17] as examples of isomorphic neighborhoods in MPI.

We conduct our experiments for different values of D, R. Fig. 4 shows the results for 8K MPI processes and 4 B message size. In almost all cases, our approach successfully decreases the latency for all values of D and R. The improvements are at least 43% and can reach as high as 67% for $D = 4, R = 3$. The only exception is the case with $D = 2, R = 1$ in Fig. 4(a), where we can see a slight increase in the latency. The reason is twofold. First, in this case, each process has a very small number of neighbors (namely 8), for which the naive approach will be good enough. Second, the number of common neighbors between any two processes is at most 4. Considering the extra communication required to exchange messages between any two friend processes, we cannot achieve any considerable decrease in the number of communications with only 4 common neighbors.

Another observation is that we get greater improvements with the increase in either D or R. The reason is that higher values of D and/or R result in a higher number of neighbors per process, providing more opportunity for optimizations through message-combining. Note that we do not have any results for $D = 4, R = 4$. The reason is that with 8K processes and $D = 4$, we will have 8 processes along some of the dimensions, which is not large enough to support a neighborhood radius of 4 without duplicate neighbors. Moreover, we mostly see similar results for both the persistent and nonpersistent versions of our design, implying a low overhead in Phase 2 of our design. This is confirmed by the results shown later in Section VI-C.

TABLE I: Random sparse graph. Time spent in Phase 1 and Phase 2—4K processes

Edge Density	$\delta = 0.05$	$\delta = 0.1$	$\delta = 0.2$	$\delta = 0.4$	$\delta = 0.8$
Phase 1	13.54 s	17.36 s	21.58 s	24.71 s	30.56 s
Phase 2	0.0005 s	0.001 s	0.005 s	0.019 s	0.07 s

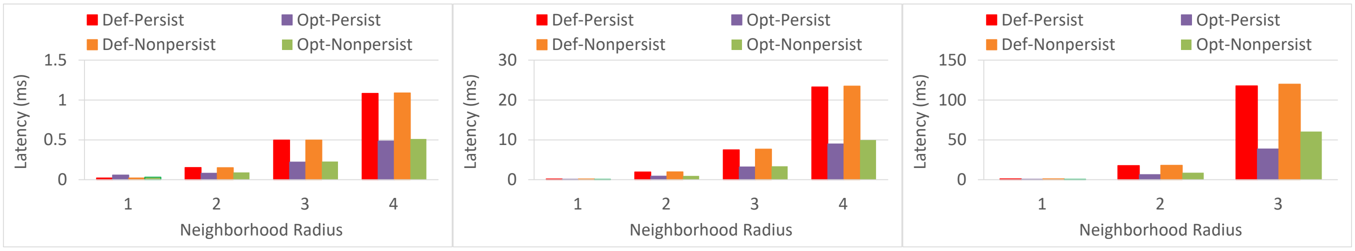
TABLE II: Moore neighborhood. Time spent in Phase 1 and Phase 2—8K processes

(D, R)	$(2, 2)$	$(2, 3)$	$(2, 4)$
Phase 1	0.31 s	0.62 s	0.96 s
Phase 2	0.000008 s	0.000017 s	0.000047 s

Fig. 5 shows the results across different message sizes and for three values of D, R. We see a consistently lower latency achieved by our approach across all message sizes up to 1 KB. The improvements are mostly around 40% for $D = 2, R = 2$ and increase to more than 65% for higher values of D. Other results (not given here) show that all approaches have almost the same performance at 4 KB message size (the exact point depends on D, R), after which the naive approach outperforms message-combining. Also, we do not see a considerable difference between the persistent and nonpersistent approaches with lower values of D, R. The impacts become more visible starting from $D = 4, R = 2$.

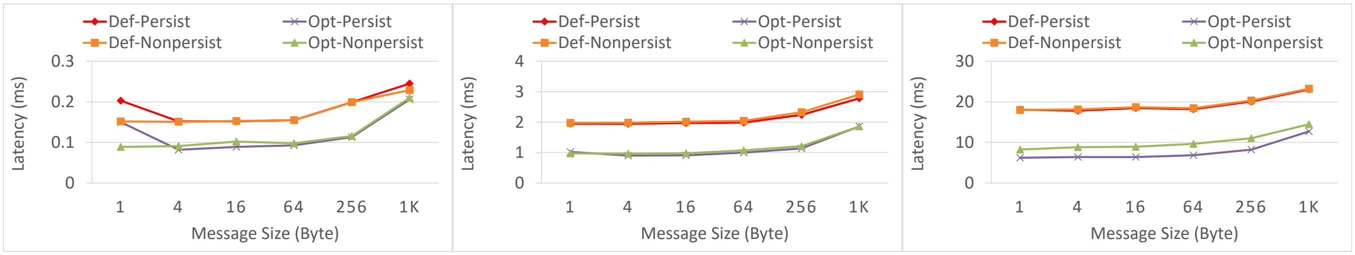
C. Overhead Analysis

In this section we analyze the overheads associated with our design. Table I and Table II, respectively, show the time spent in each phase for the sparse random graph and the Moore neighborhood. As we can see, the main overhead is due to Phase 1, where we build the message-combining pattern (Alg. 1). The overheads of Phase 2 (Alg. 3) are 4 orders of magnitude lower and remain well below 1 second in all cases. Therefore, we do not see significant differences between the persistent and nonpersistent cases in the results shown in Section VI-A and VI-B. This is particularly true with the Moore neighborhood which has very low Phase 2 overheads in Table II.



(a) $D = 2,\ 128 \times 64$ processes (b) $D = 3,\ 32 \times 16 \times 16$ processes (c) $D = 4,\ 16 \times 8 \times 8 \times 8$ processes

Fig. 4: Neighbor allgather average latencies for the Moore neighborhood topology—8K processes, 4B message size.



(a) $D = 2, R = 2$ 128×64 processes (b) $D = 3, R = 2$ $32 \times 16 \times 16$ processes (c) $D = 4, R = 2$ $16 \times 8 \times 8 \times 8$ processes

Fig. 5: Neighbor allgather average latencies for the Moore neighborhood topology—8K processes, various message sizes.

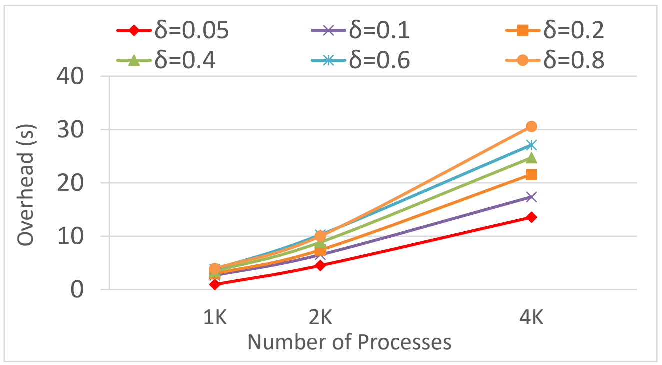


Fig. 6: Phase 1 overheads for the sparse random graph topology across different number of processes.

Next, we evaluate how the overheads are affected by the total number of processes as well as the number of neighbors per process. Due to lack of space, we only discuss Phase 1. Fig. 6 shows the overhead of Phase 1 for the sparse random graph topology across various numbers of processes and edge densities. The overhead increases with the total number of processes as well as edge densities (i.e., number of neighbors). An important observation is that the increase in the total number of processes has a much higher impact on overhead than does the increase in the number of neighbors. In fact, Fig. 6 shows a super-linear increase in the overhead with the total number of processes, whereas we see a sub-linear trend with the edge densities.

This behavior has its roots in the fact that the overhead of Phase 1 is dominated by the point-to-point communications used in Alg. 2 to notify the friend processes (Lines 23 and 27 of Alg. 2). On the other hand, other results (not presented here) show that for the random sparse graph topology, the number of iterations of Alg. 1 increases with the square root of F, the (maximum) number of friends per process (i.e., $t \propto \sqrt{F}$). This leads to an overhead complexity of $\mathcal{O}(\sqrt{F}F)$ for Phase 1. Moreover, in the random sparse graph topology, the number of friends per process increases linearly with the number of processes, but remains unchanged with increase in the edge densities. The reason is that in the random sparse graph topology (and $\Theta = 4$), each process tends to be a friend with almost every other process, even at low edge densities. In fact, starting from 0.1 edge density, each process becomes a friend with all other processes. Thus, Phase 1 overhead increases with $\mathcal{O}(p^{1.5})$ for the random sparse graph topology.

Fig. 7 shows the overhead trend of Phase 1 for the Moore neighborhood topology. An important observation is that we no longer see a super-linear increase in the overhead. In fact, the overhead remains mostly unchanged as we increase the number of processes, and it increases only with R. The reason is that Moore neighborhoods model a more structured and localized neighborhood topology compared with that of random sparse graphs. Therefore, the number of friends per process is not affected by the increase in the total number of processes. It increases linearly only with the number of neighbors per process. In addition, other results (not presented here) show that for the Moore neighborhood, the number of iterations of Alg. 1 is independent of F. Moreover, the number of neighbors per process (Δ) is independent of the

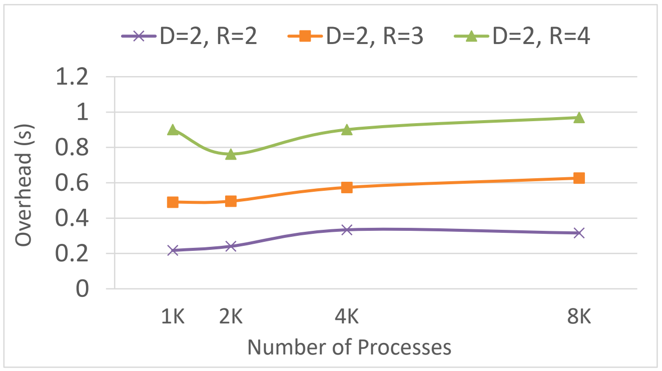


Fig. 7: Phase 1 overheads for the Moore neighborhood topology across different number of processes.

total number of processes, resulting in an overhead complexity of $\mathcal{O}(\Delta)$. This is encouraging as many real applications tend to use such structured and localized neighborhood topologies.

VII. Conclusion and Future Work

We presented a new approach to build optimized communication schedules for the MPI neighborhood collectives, and gave distributed algorithms to find and exploit potential common neighborhoods in distributed graph topologies for building message-combining communication schedules. Our experimental results show that significant improvements of up to 71% can be achieved for neighborhood topologies of various shapes and sizes.

We plan to evaluate the benefits of our design for real applications. A major challenge is with porting applications' source code to neighborhood collective function calls; an effort that requires collaboration with application developers. Moreover, we intend to enhance our design with *cumulative combining*, where each process reconsiders the set of its onloaded neighbors for further message combining with other friend processes. We also plan to reduce the overheads by utilizing MPI one-sided communication. This can help to avoid a large fraction of the point-to-point communications that are performed among the friend processes. We are also interested in evaluating the impact of using other distributed maximum matching algorithms. Exploiting MPI datatypes to avoid memory copies and implement zero-copy schedules is another work planned for future.

Acknowledgments

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada Grant RGPIN/05389-2016. Computations were performed on the GPC supercomputer at the SciNet HPC Consortium. SciNet is funded by: the Canada Foundation for Innovation under the auspices of Compute Canada; the Government of Ontario; Ontario Research Fund - Research Excellence; and the University of Toronto.

References

[1] "Message Passing Interface Forum, http://www.mpi-forum.org/, last accessed 2017/06/20."

[2] T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, and J. L. Träff, "The scalable process topology interface of MPI 2.2," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 4, pp. 293–310, 2011.

[3] T. Hoefler and J. L. Traff, "Sparse collective operations for MPI," in *Workshop on High-level Parallel Programming Models and Supportive Environments (HIPS, IPDPS)*, 2009, pp. 1–8.

[4] "Nek5000, https://nek5000.mcs.anl.gov/, last accessed 2017/06/20."

[5] F. Gygi, "Large-scale first-principles molecular dynamics: moving from terascale to petascale computing," *Journal of Physics: Conference Series*, vol. 46, no. 1, pp. 268–277, 2006.

[6] "MPICH: High-performance and widely portable MPI implementation, http://http://www.mpich.org/, last accessed 2017/06/20."

[7] "MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE, http://mvapich.cse.ohio-state.edu/, last accessed 2017/06/20."

[8] "Open MPI: Open source high performance computing, http://www.open-mpi.org/, last accessed 2017/06/20."

[9] S. H. Mirsadeghi, J. L. Träff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize MPI neighborhood collectives," ECE0630. Technical Report, ECE Dept., Queen's University, Kingston, ON, Canada, June 2017.

[10] 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.

[11] 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 *Proc. International Conference on Cluster Computing Workshops*, 2012, pp. 222–230.

[12] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[13] 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 *Proc. International Symposium on Parallel & Distributed Processing (IPDPS)*, 2010, pp. 1–11.

[14] S. Kumar, G. Dozsa, G. Almasi, P. Heidelberger, D. Chen, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith, and C. J. Archer, "The Deep Computing Messaging Framework: Generalized scalable message passing on the Blue Gene/P supercomputer," in *Proc. International Conference on Supercomputing (ICS)*, 2008, pp. 94–103.

[15] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 98:1–98:10.

[16] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold, "Isomorphic, sparse MPI-like collective communication operations for parallel stencil computations," in *Proc. European MPI Users' Group Meeting (EuroMPI)*, 2015, pp. 10:1–10:10.

[17] J. L. Träff, A. Carpen-Amarie, S. Hunold, and A. Rougier, "Message-combining algorithms for isomorphic, sparse collective communication," arXiv:1606.07676, 2016.

[18] E. Chan, M. Heimlich, A. Purkayastha, and R. van de Geijn, "Collective communication: theory, practice, and experience," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 13, pp. 1749–1783, 2007.

[19] M. Wattenhofer and R. Wattenhofer, "Distributed weighted matching," in *Proc. International Symposium on Distributed Computing (DISC)*, 2004, pp. 335–348.

[20] J.-H. Hoepman, "Simple distributed weighted matchings," *arXiv preprint cs/0410047*, 2004.

[21] Z. Lotker, B. Patt-Shamir, and S. Pettie, "Improved distributed approximate matching," *J. ACM*, vol. 62, no. 5, pp. 38:1–38:17, 2015.

[22] C. Koufogiannakis and N. E. Young, "Distributed algorithms for covering, packing and maximum weighted matching," *Distributed Computing*, vol. 24, no. 1, pp. 45–63, 2011.

[23] C. U. Ileri and O. Dagdeviren, "Performance evaluation of distributed maximum weighted matching algorithms," in *Proc. International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, 2016, pp. 103–108.

[24] C. Loken *et al.*, "SciNet: Lessons learned from building a power-efficient Top-20 system and data centre," *Journal of Physics: Conference Series*, vol. 256, no. 1, 2010.