Optimizing Irregular Communication with Neighborhood Collectives and Locality-Aware Parallelism

Gerald Collom
Department of Computer Science
University of New Mexico
Albuquerque, USA

Rui Peng Li
Center for Advanced Scientific
Computing Lawrence Livermore
National Laboratory
Livermore, USA

Amanda Bienz
Department of Computer Science
University of New Mexico
Albuquerque, USA

ABSTRACT

Irregular communication limits both performance and scalability of parallel applications. Typically, it is implemented as point-to-point, and optimizations are integrated into the application, lacking portability. Optimization of point-to-point messages within MPI is difficult, as the interface only provides information on a piece of overall communication. However, persistent neighbor collectives expose a suitable interface for such optimizations.

This paper presents methods for implementing existing optimizations for irregular communication within neighborhood collectives, analyzes the impact of neighborhood collectives in Hypre BoomerAMG, and shows up to a 1.38x speedup on sparse matrix-vector multiplication using optimized neighbor collectives. The authors analyze three implementations of neighborhood collectives for `Alltoallv`: an unoptimized wrapper of standard point-to-point communication, and two locality-aware aggregating methods. The second exposes a non-standard interface to perform additional optimization for an additional 0.07x speedup.

Optimizations are available open-source in MPI Advance which wraps MPI, allowing use with any MPI installation.

ACM Reference Format:
Gerald Collom, Rui Peng Li, and Amanda Bienz. 2023. Optimizing Irregular Communication with Neighborhood Collectives and Locality-Aware Parallelism. In *Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3624062.3624111

Keywords: Hypre, AMG, MPI, neighborhood collectives, locality-aware parallelism, persistent communication

1 INTRODUCTION

Parallel applications, such as simulations and iterative solvers, are often bottlenecked by irregular point-to-point communication. For instance, the performance and scalability of Hypre [1, 19], a widely-used algebraic multigrid (AMG) solver, is limited by the irregular communication that occurs throughout its numerous sparse matrix operations. While there are many optimizations for point-to-point communication, including persistent communication and locality-aware aggregation, there is no widely used library supporting these optimizations, requiring each application to optimize code by hand. This paper presents two neighborhood collective implementations that utilize existing optimizations of point-to-point communication, and the performance of these implementations are analyzed within the Hypre BoomerAMG solver.

Parallel applications typically implement their own irregular communication with calls to `MPI_Isend` and `MPI_Irecv`, or some variation of these methods. Communication optimizations are currently added within applications, and as a result not easily shared among parallel codebases. For instance, the AMG solvers Hypre, Muelu [3], and GAMG [32] each call separate implementations for point-to-point communication within sparse matrix operations, with optimizations unique to each. Furthermore, there is no easy way to add point-to-point optimizations within methods such as `MPI_Isend` and `MPI_Irecv` as these only pass information about a single message rather than the collection of all messages.

This paper addresses the point-to-point communication bottleneck through the use of MPI neighborhood collectives, which execute irregular communication while allowing for optimization within MPI. The sparse neighbor collective, e.g., `Alltoallv`, interface requires applications to provide information about all messages, allowing for optimizations within the method. While neighborhood collectives provide sufficient information for optimizations within MPI, many communication optimizations incur large initial overheads which are offset during subsequent iterations. Therefore, the addition of persistent neighborhood collectives in the MPI 4 standard allows for substantial irregular communication optimizations to be added within MPI. Adding these optimizations within MPI implementations allows for all applications to take advantage of them by simply calling the appropriate neighborhood collective.

While neighborhood collectives have potential to alleviate critical communication bottlenecks in irregular applications, they have yet to be widely adopted. While the interface has existed since the MPI 3 standard, implementations of neighborhood collective often simply wrap point-to-point communication with limited exploration of possible optimization. As few applications use these methods, there is little incentive to improve optimizations. At the same time, while the implementations contain few optimizations, there is little advantage to rewriting existing applications to utilize these methods. To alleviate this standstill, the goal of the work presented in this paper is two-fold: to present an optimized implementation of the persistent version of the neighborhood collective `MPI_Neighbor_alltoallv`, and to restructure existing parallel codebases, such as the widely used parallel multigrid solver Hypre,

SC-W 2023, November 12–17, 2023, Denver, CO, USA

ACM ISBN 979-8-4007-0785-8/23/11…$15.00
https://doi.org/10.1145/3624062.3624111

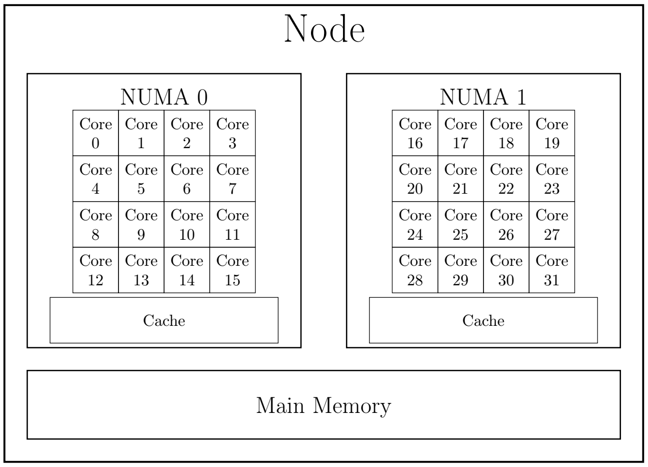


Figure 1: An example symmetric multiprocessing (SMP) node with two NUMA regions and 32 cores.

to replace point-to-point communication with persistent neighborhood collectives. All neighborhood implementations are added to a lightweight open-source library, called MPI Advance [1], which sits on top of MPI, allowing it to optimize which system version of MPI is installed. Furthermore, all neighborhood collective additions to BoomerAMG are published in the neighbor collective branches of Hypre.

The contributions of this paper include the following:

- Novel use of persistent neighborhood collectives to wrap optimizations for irregular communication in a portable format.
- An analysis of the overhead of existing neighbor collective operations, including topology communicator creation.
- A novel locality-aware optimization, which minimizes expensive message counts but not sizes, fitting within the current neighbor collective API.
- A novel extension to the current neighbor collective API, allowing for existing locality-aware aggregation techniques to be utilized within neighbor collectives.
- A performance study within the state-of-the-art solver Hypre, utilizing the portability of the neighborhood collective to optimize large existing codebases with minimal changes.

The remainder of this paper analyzes reductions to the cost of irregular communication through locality-aware neighbor collectives. Modern supercomputers contain a hierarchy of regions, with communication within a region being at different cost than between regions. For example, parallel architectures typically contain many nodes connected by a network, with each node containing many processes, as exemplified in Figure 1. x

This example symmetric multiprocessing (SMP) node contains two non-uniform memory access (NUMA) regions, each with 16 cores. All processes within a NUMA region share a level of cache, allowing intra-NUMA communication to be transferred through cache. Similarly, as all processes on the node share main memory, inter-NUMA communication within a node can be transferred through main memory. Finally, inter-node messages are injected into the network and transferred across the interconnect to the node of destination. As a result, computers achieve varying communication costs with regard to the locality of the messages. Locality-aware communication restructures point-to-point messages to reduce the most expensive messages in exchange for additional less costly communication. This paper introduces multiple novel strategies for adding locality-aware aggregation within neighborhood collectives, and presents significant associated performance improvements from replacing point-to-point communication within a widely used parallel codebase with locality-aware neighborhood collectives.

The remaining sections of this paper are organized as follows. Section 2 describes communication optimizations and neighborhood collectives in more detail and describes a number of related research works. Section 3 details the various neighborhood collective optimizations, and performance results associated with these implementations are presented in Section 4. Finally, conclusions and future directions are discussed in Section 5.

2 BACKGROUND

Each generation of supercomputer brings unique architectural design choices. In recent history, parallel systems have continuously increased potential compute power with additional complexity within each node. While older supercomputers such as the Blue Gene/L consisted of only a single dual-core chip per node [13], Blue Gene/Q systems such as Sequoia, were comprised of symmetric multiprocessing (SMP) nodes with 16 cores per node split across 2 CPUs [11]. More recent systems, such as Summit, contain nodes with 2 22-core CPUs [21], and emerging systems, such as Frontier, contain a single 64-core chip per node, split into 4 16-core NUMA regions [31]. The additional per-node complexity of each generation of parallel systems increases the variety in communication costs, with notable differences between intra-CPU, inter-CPU, and inter-node communication [6]. While these performance differences between locality regions vary by system, inter-CPU communication within the same node is significantly more costly than between nodes on current and emerging systems [10].

Parallel applications often fail to take full advantage of available compute power due to performance and scaling constraints associated with inter-process communication. Many simulations and numerical solvers are dominated by irregular communication, which requires each process to communicate varying amounts of data with varying subsets of other processes. Algebraic multigrid, for instance, relies on the performance of sparse matrix operations, such as the sparse matrix-matrix and sparse matrix-vector (SpMV) multiples. AMG first creates a hierarchy of increasingly dense matrices that approximate lower frequencies, with each successive matrix formed through a triple sparse matrix-matrix multiply. After the hierarchy is created, the solution is iteratively refined through numerous SpMVs on each level of the hierarchy. Sparse matrix operations require each process to receive data associated with every non-zero column held by the process. As a result, each process communicates varying amounts of data with a subset of other processes, as determined by the sparsity pattern of the given sparse matrix. As coarse levels within AMG are increasingly dense, communication requirements are often increased on levels near the middle of the hierarchy. Finally, at scale, the cost of these sparse matrix operations is dominated by the cost of irregular inter-process communication.

[1] https://github.com/mpi-advance

Example 2.1. *Assume a system has multiple regions, each containing four processes, as displayed in Figure 2. Each process within region 0 holds two unique values, represented as a circle and square. The shaded regions of these objects correspond to the processes in region 1 to which each object must be sent. For example, process $P0$ holds a circle shaded both red and green, and therefore must send this object to processes $P5$ and $P6$. Furthermore, the square held by $P0$ is shaded blue, red, and orange, and therefore must be sent to processes $P4$, $P5$, and $P7$. Throughout the remainder of this paper, the authors present multiple methods for communicating these values between regions* 0 *and* 1.

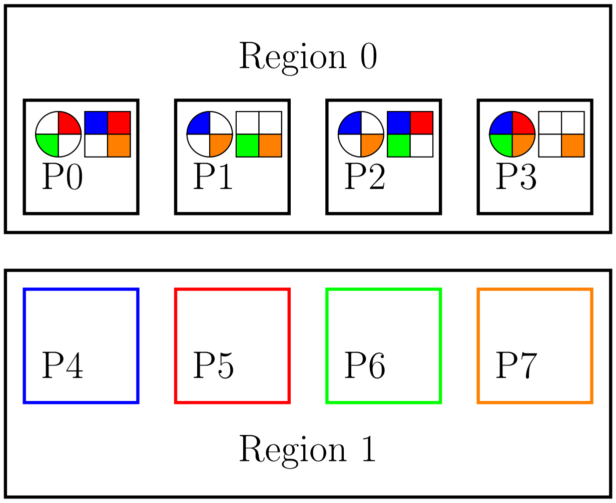


Figure 2: Visualization of the irregular communication pattern described in Example 2.1.

Example 2.1 describes a simple irregular communication pattern. This paper shows the effects of locality-aware neighborhood collectives on communication throughout the iterative solve phase of AMG, as the associated SpMVs require a large range of communication patterns. Optimized neighbor collectives, however, are not limited to AMG and can be used to reduce the cost of irregular communication within other solvers and simulations.

Standard methods of irregular communication consist of collecting all data to be sent to a process before sending it directly as a single message. This approach fails to account for the locality of the sending and receiving processes. For instance, two processes within the same CPU are able to transfer data through cache, often at a significantly faster rate than data can be transported through the interconnect. Locality-aware methods, on the other hand, aggregate data within a region of locality to minimize the number and size of inter-region messages. The locality-aware neighborhood collectives presented in Section 3 utilize three-step aggregation [9], in which every process in a region performs all communication with a unique subset of other regions. The intra-region data is initially redistributed so that each process per region holds all data to be sent to its unique subset of regions. Each process then sends a single inter-region message to each of its assigned regions. Finally, received data is redistributed within each region to transfer data to each final destination process. Note, there are many additional strategies for aggregation that could be utilized within neighborhood collectives. The authors focus on the three-step aggregation as this paper presents the effects of communication optimizations on sparse matrix-vector multiplication throughout AMG, where this method of aggregation has been shown to perform best [9]. However, other simulations and solvers may be better optimized with additional locality-aware strategies.

While neighborhood collectives provide the necessary interface for optimizing irregular communication within MPI, they do require some overhead compared to standard point-to-point communication, namely with forming the neighborhood topology. Before a neighborhood collective, such as the `MPI_Neighbor_alltoallv`, is executed, a neighborhood must first be formed. For irregular communication, a neighborhood communicator is formed at scale with the method `MPI_Dist_graph_create_adjacent`. This graph creation is passed data about each process that a given process sends to and receives from, and returns a directed neighborhood of processes with which each process communicates. There is synchronization overhead associated with this graph creation. Only a single neighborhood is needed for each required communication pattern, however, such as each unique sparse matrix within a solver. Therefore, within iterative methods, the graph creation is amortized over subsequent iterations.

Persistent neighborhood collectives allow for further amortizations of setup costs across all iterations. Persistent MPI communication consists of initializing communication once, before starting and waiting on all communication at every iteration. Persistent neighborhood collectives first set up the collective with the `MPI_Neighbor_alltoallv_init` method. Then, each iteration of communication consists of calls to `MPI_Start` and `MPI_Wait`, during which all communication is completed. Furthermore, the separate start and wait methods allow for an overlap of communication and computation, assuming the MPI implementation supports strong progress [22]. This paper utilizes the persistent neighborhood API, allowing all locality-aware setup costs, such as load balancing while determining which intra-region process communicates with each region, to be incurred once within `MPI_Neighbor_alltoallv_init`. These overheads are then quickly offset by per-iteration reductions to communication costs.

2.1 Related Work

Before costly communication can be optimized, architectures and paths of communication must be accurately benchmarked and modeled for emerging systems to pinpoint the costs of the various messages. As emerging systems increase in complexity, performance models and benchmarks are adapted to fully capture the costs of the various paths of irregular communication. While the postal models typically suffice for simple point-to-point communication [2], many extensions have been necessary to capture costs that dominate SMP architectures. For instance, the maxrate model greatly improves inter-node communication costs over the postal model by adding in measures for injection bandwidth limits [17]. The maxrate model is further optimized through locality-awareness, modeling intra-CPU, inter-CPU, and inter-node messages separately [6]. While the maxrate model accurately captures costs of inter-node communication, intra-node communication models are further improved by adding constraints for all active processes, as bandwidth varies within a node based on the number of active processes [33]. Finally, models for irregular communication, particularly for the large number of messages that occur within the coarse levels of AMG, are further improved by estimating queue search and network contention costs [6].

Locality-aware communication has previously been explored extensively, both with point-to-point communication and in MPI collectives. Three-step aggregation, the focus of this paper, has shown to greatly improve instances of irregular communication in which many small messages are sent, such as in the solve phase of AMG [9]. Similarly, two-step aggregation greatly reduces the costs associated with sending numerous larger messages such as within sparse matrix-matrix multiplies [7], while ideal aggregation, which combines portions of messages ranging from two-step to three-step, optimizes the costs of medium-sized messages, such as within sparse matrix-multi-vector multiplies [25]. Similar aggregation techniques have shown large speedups within inter-GPU communication on heterogeneous architectures [20, 26].

Node-awareness is also a common technique for improving the performance of collective communication. Hierarchical communication consists of creating one or more master processes per node, and only performing steps of inter-node communication between these master processes [15, 23, 24, 34]. Multi-lane approaches have further optimized inter-node communication within large collectives by having each process per node communicate a portion of the inter-node data [35]. Locality-aware collective algorithms reduce the cost of small collectives by minimizing the number of inter-node steps, having each process per node communicate with a separate node at each inter-node step [5, 8].

Topology-awareness, or optimizing algorithms for a given interconnect, is another common approach for minimizing collective communication costs. There are two categories of topology-aware algorithms, those which remap data to cores to minimize the number of hops messages are communicated [4, 27, 28], and those that reformulate algorithms to minimize the number of steps for a given topology [29, 30]. Topology-aware neighborhood collectives have previously been shown to improve sparse matrix-matrix multiplication [14]. While topology-aware algorithms greatly improve the performance of collective algorithms, they are specific to a given interconnect, which varies with emerging architectures.

There are a number of APIs for irregular communication that exist within the MPI 4 standard, and therefore implemented within all versions of MPI, including persistent and partitioned communication. Persistent communication reduces initialization costs by having an initialization so that all overhead is only incurred once [18]. All subsequent communications then communicate data without initialization overhead. Persistent communication exists for both point-to-point communication and collective operations. Partitioned communication extends the persistent point-to-point interface, allowing multiple threads or tasks to contribute data to a single message [12, 16]. As a result, large messages that are partitioned across threads are sent in chunks rather than incurring a synchronization cost waiting for all threads to initialize corresponding communication.

3 PERSISTENT NEIGHBORHOOD COLLECTIVE IMPLEMENTATIONS

Neighborhood collectives, such as the `MPI_Neighbor_alltoallv` provide the API for irregular communication optimizations within MPI. Furthermore, the persistent version of this method (released in MPI 4) allows for further optimizations because overhead, such as load balancing, is only incurred once and amortized over all successive iterations. Persistent neighbor collectives can wrap irregular communication throughout parallel applications, replacing point-to-point communication with a single initialization step, `MPI_Neighbor_alltoallv_init`, followed by `MPI_Start` and `MPI_Wait` to begin and complete each iteration of communication, respectively. All neighborhood collectives, regardless of persistence, do require an additional step of setup beyond point-to-point communication, as the topology communicator must first be formed with a method such as `MPI_Dist_graph_create_adjacent`.

3.1 Standard Neighborhood Collectives

The standard `MPI_Neighbor_alltoallv_init` implementation consists of initializing a persistent non-blocking send and receive for all data to be sent to any process, regardless of regions of sending and receiving processes, as displayed in Algorithm 1, in which `args` are the standard `MPI_Neighbor_alltoallv_init` arguments.

Algorithm 1: `standard_init`

```
Input: args                                    {Standard method arguments}
// Send count and processes, recv count and processes, in communicator
Get $n_{\mathtt{send}}$, $p_{\mathtt{send}}$, $n_{\mathtt{recv}}$, $p_{\mathtt{recv}}$ from communicator
for $i \leftarrow 0$ to $n_{send}$ do
    MPI_Isend_init to $p_{\mathtt{send}_i}$   {Send buffer, displacements, and size in args}
for $i \leftarrow 0$ to $n_{recv}$ do
    MPI_Irecv_init from $p_{\mathtt{recv}_i}$   {Recv buffer, displacements, and size in args}
```

Similarly, during each instance of communication, all messages are started at once, as shown in Algorithm 2.

Algorithm 2: `standard_start`

```
Input: args                                    {Standard method arguments}
// Send count and processes, recv count and processes, in communicator
Get $n_{\mathtt{send}}$, $p_{\mathtt{send}}$, $n_{\mathtt{recv}}$, $p_{\mathtt{recv}}$ from communicator
for $i \leftarrow 0$ to $n_{send}$ do
    Start send $i$
for $i \leftarrow 0$ to $n_{recv}$ do
    Start recv $i$
```

The calling process then waits for all messages to complete, such as with `MPI_Waitall`, as displayed in Algorithm 3.

Standard implementations directly wrap point-to-point messages within a single API. This approach fails to optimize communication

Algorithm 3: `standard_wait`

```
Input: args                                    {Standard method arguments}
// Send count and recv count, in communicator
Get $n_{\texttt{send}}$, $n_{\texttt{recv}}$ from communicator
Wait for $n_{\texttt{send}}$ sends and $n_{\texttt{recv}}$ receives to complete
```

by, e.g., minimizing expensive communication between non-local regions. For instance, standard neighborhood collective communication of Example 2.1 consists of each process in region 0 communicating with all processes in region 1, as displayed in Figure 3. This figure displays all messages originating on process $P2$. This procedure retrieves both values represented by the circle and square, and sends them in a single messages to process $P4$, as both shapes on $P2$ have shaded blue regions, indicating $P4$ requires both values. The value represented by the square is then additionally sent multiple times, once to $P5$ and once to $P6$. Finally, the value represented by the circle is also sent to $P7$. In total, this example requires 15 messages to be sent from region 0 to region 1, and all data values with multiple indicating colors are sent in multiple inter-region messages.

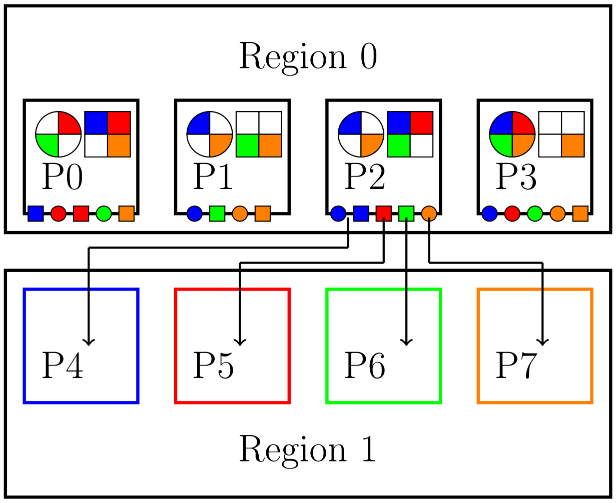


Figure 3: Standard point-to-point communication. Processes send messages directly to each destination process. E.g., P2 sends a message to P4 with both data values (circle and square), and a message to P5 with only the second value (square), etc.

3.2 Aggregating Messages

The `MPI_Neighbor_alltoallv_init` interface provides the necessary information for locality-aware optimizations, such as aggregation of data within local regions, to be performed within MPI libraries. The method arguments include information on all processes with which to send or receive data, and the amount of data to be sent to each. This is sufficient information for all processes within a region to determine the regions to which they send to and receive from and the inter-region data sizes. Methods of aggregation, such as locality-aware strategies, partition the communication across all processes per region so that each sends a minimal portion of messages for small data sizes, or an equal portion of data when sizes are large.

Aggregation within the persistent neighborhood collectives is shown in Algorithm 4. The black text, excluding red text, describes aggregation within the standard `MPI_Neighbor_alltoallv_init` API.

Algorithm 4: `aggregated_init`

```
Input: args                                    {Standard method arguments}
       send_idx                                {Unique send indices}
       recv_idx                                {Unique recv indices}
// Send count and processes, recv count and processes, in communicator
Get $n_{\texttt{send}}$, $p_{\texttt{send}}$, $n_{\texttt{recv}}$, $p_{\texttt{recv}}$ from communicator
// Form aggregated communication args
// $l$ : fully local communication, $s$ : initial local redistribution
// $g$ : global, inter-region communication, $r$ : final redistribution of received data
setup_aggregation($l$, $s$, $g$, $r$, send_idx, recv_idx)
standard_init($l$)
standard_init($s$)
standard_init($g$)
standard_init($r$)
```

The method `setup_aggregation` creates the path of aggregation, assigning a portion of the inter-region communication to each process within a region. Both aggregating implementations presented in this paper use three-step aggregation [9], but this could be replaced by any aggregation strategy. The aggregated communication consists of fully local communication and three aggregation steps:

- ℓ : fully local communication, with source and destination process located within the same region
- s : initial redistribution of data within a region
- g : inter-region communication
- r : final redistribution of received data within a region

Fully local communication is executed in parallel with the three aggregation steps. Persistent communication is initialized for each of these four steps.

During each instance of communication, all fully local ℓ and inter-region g communication is started within the method `start`, as described in Algorithm 5.

The initial redistribution of data within the region must be fully completed before inter-region communication can begin. Therefore, this method consists of both starting and completing the initial redistribution s, before starting the inter-region communication g. Because of this, starting synchronizes within regions but not globally.

Finally, each instance of communication is completed within the `wait` method, described in Algorithm 6.

The inter-region communication g must complete before the final intra-region redistribution of data can be performed. Therefore, this method consists of completing the inter-region step g, before both starting and completing the final intra-region redistribution r.

This approach greatly reduces the number of inter-region messages. For instance, the inter-region communication required within

Algorithm 5: `aggregated_start`

```
Input: $l_{args}$          {fully local communication arguments}
       $s_{args}$          {initial local redistribution arguments}
       $g_{args}$          {global, inter-region communication arguments}
       $r_{args}$          {final local redistribution arguments}
// Start fully local communication
standard_start($l_{args}$)
// Start and complete initial redistribution
standard_start($s_{args}$)
standard_wait($s_{args}$)
// Start inter-region communication
standard_start($g_{args}$)
```

Algorithm 6: `aggregated_wait`

```
Input: $l_{args}$          {fully local communication arguments}
       $s_{args}$          {initial local redistribution arguments}
       $g_{args}$          {global, inter-region communication arguments}
       $r_{args}$          {final local redistribution arguments}
// Complete fully local communication
standard_wait($l_{args}$)
// Complete inter-region communication
standard_wait($g_{args}$)
// Start and complete final redistribution
standard_start($r_{args}$)
standard_wait($r_{args}$)
```

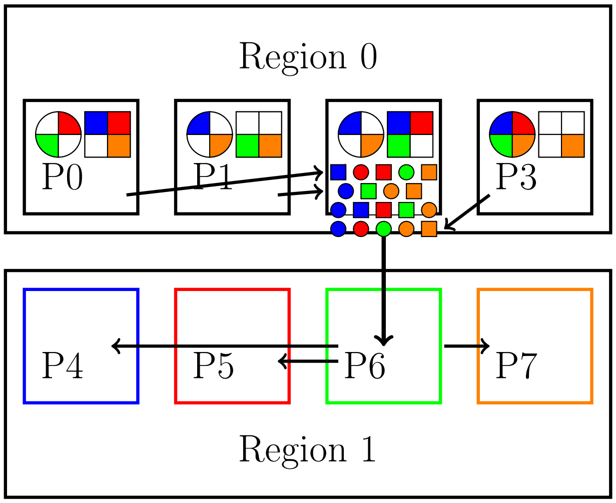


Figure 4: Aggregated neighborhood communication. For each destination region, processes in a region send messages to a single process, e.g., P2 which then sends a single message to a process, e.g., P6 in the destination region. Then, the receiving process (P6) sends messages to their final destination processes within the same region.

Example 2.1 is performed in three steps, as shown in Figure 4. Initially, all inter-region messages are redistributed locally so that each process holds all data to be sent to a unique subset of regions. For example, in Figure 4, all data to be communicated to region 1 is first sent locally to process $P2$. Each row of values on $P2$ represent values from processes $0-3$ respectively. Then, process $P2$ sends a single inter-region message to $P6$. Finally, process $P6$ redistributes the received values locally.

3.3 Extensions for Duplicate Data

By default, duplicate values are communicated between regions when a process in one region needs to communicate a certain value to multiple destination processes co-located in a second region. For instance, Example 2.1 displays a scenario in which P0 sends the same data value represented by a circle to both P5 and P6. This situation could arise in, for example, sparse matrix-vector multiplication, when the vector is partitioned across processes. The SpMV operation requires values in each partition to be communicated to any processes (possibly co-located) containing non-zero values in the corresponding columns of the matrix. In previous approaches, these values are communicated between regions in duplicate, once per destination. As the current `Alltoallv` interface does not allow providing unique identifiers of values being communicated, there is no simple way for users to provide the information necessary to remove these duplicates.

While the persistent neighborhood collective API provides sufficient information to aggregate messages within each region, it fails to include necessary information for removal of duplicate values. A small extension to the API, requiring uniquely identifying indices associated with each data value to be communicated, allows for minimization of inter-region message sizes on top of doing aggregation. This extension to `MPI_Neighbor_alltoallv_init` is displayed as red text throughout Algorithm 4. The uniquely identifying indices are used while setting up aggregation to remove duplicate values from intra-region aggregation and inter-region communication. Note, besides three-step aggregation, any aggregation technique can use this information to minimize inter-region data sizes.

Figure 5 displays the deduplicating approach for communicating the values in Example 2.1. As discussed, each value from each source process now has only one copy sent both within the two regions for aggregation and between the two regions. Each row of values (circle and square) on $P2$ represent the unique aggregated values from processes 0 to 3 respectively.

4 EXPERIMENTAL RESULTS

The performance of the neighborhood collective implementations presented in Section 3 were analyzed throughout the sparse matrix-vector multiplies of the solve phase of Hypre's BoomerAMG.

Throughout the presented results, the following four communication protocols are analyzed :

- Standard Hypre : persistent point-to-point communication as implemented in release 2.28 of Hypre
- Unoptimized neighborhood collectives : standard communication within a persistent neighborhood collective, as described in Section 3.1.
- Aggregating neighborhood collectives : locality-aware aggregation within a persistent neighborhood collective, as described in Section 3.2

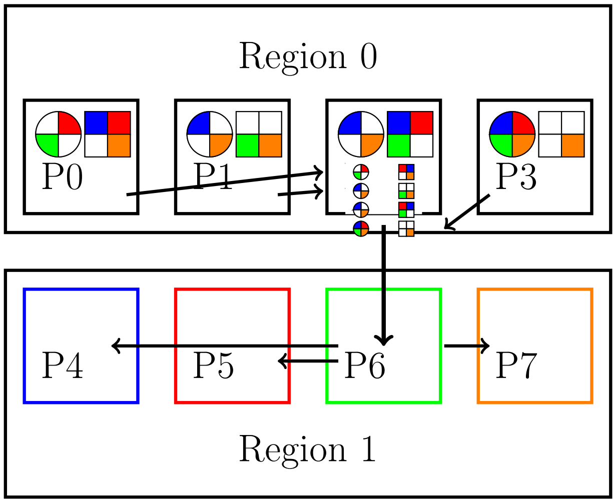


Figure 5: Deduplicating neighborhood communication. Based on the previous aggregating neighborhood communication, only unique identifiers for values are used to avoid communicating duplicate values. E.g., multiple processes in region 1 require P0's data values, but only one copy is sent both to P2 for aggregation and to P6 for inter-region communication.

- Deduplicating neighborhood collectives : locality-aware aggregation with additional removal of duplicate values, as described in Section 3.3

4.1 Experimental Setup

All neighborhood collective implementations are implemented within a lightweight open-source library, MPI Advance, that is then linked within Hypre. Implementations within MPI Advance then call necessary instances of point-to-point communication using the system install of MPI. All experiments are performed on a 7-point rotated anisotropic diffusion system, with rotated of 45 degrees and anisotropy of 0.001. All experiments are run on the CPU cores of Lassen, a Power9 system at Lawrence Livermore National Laboratory, using the system install of Spectrum MPI. Each node of Lassen contains two 22-core CPUs. While intra-CPU communication outperforms inter-node, inter-CPU communication within a node requires over twice the cost of inter-node for large messages [10]. Therefore, all presented results use only 16 cores per node on a single CPU to avoid inter-CPU expenses.

In an effort to achieve performance reproducibility, each performance result presented in this section acquires the time required to perform 1000 calls to `MPI_Start` and `MPI_Wait`, and then finds the average cost of a single instance of those 1000 steps of communication. Each test is run three separate times and the minimum of the three resulting averages is taken, in order to show the performance with minimal impact of other jobs running concurrently on the system.

4.2 Overhead Costs

There is an overhead to using neighborhood collectives over point-to-point communication, namely in creating the topology communicator. Neighborhood collectives require creating the topology communicator only once, amortizing this cost over all iterations of communication. For irregular communication, this communicator is formed with the method `MPI_Dist_graph_create_adjacent`. The cost of this method was evaluated for two MPI implementations available on the test system, Spectrum MPI and MVAPICH2, over a range of process counts in Figure 6. As shown in Figure 6, the method can be called with minimal overhead, but the choice of MPI implementation is important. For the problem tested, MVAPICH2 performs the method 8.6x as fast as Spectrum MPI at the scale of 2048 cores. The cost with MVAPICH2 also demonstrates improved strong scaling.

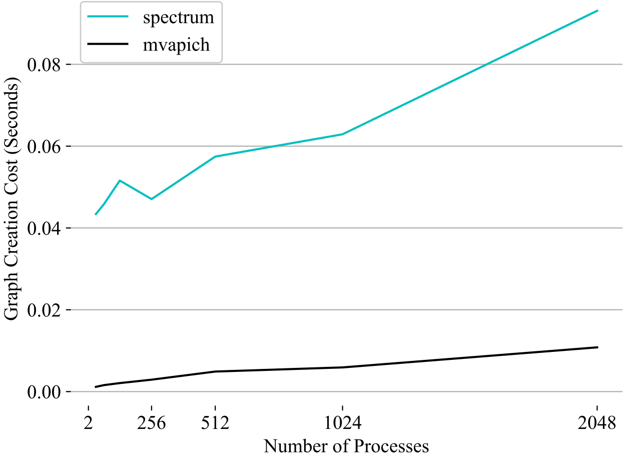


Figure 6: Cost of calling `MPI_Dist_graph_create_adjacent` once per level of the AMG hierarchy at a variety of process counts. This problem is strongly scaled, with each rotated anisotropic diffusion system containing 524 288 rows.

Persistent neighborhood collectives incur all setup costs only once during the initialization method. As a result, costly setup of optimizations, such as load balancing inter-region communication across all processes within a region, are amortized over all iterations of communication. Figure 7 displays the costs associated with initializing each of the neighborhood collectives for a rotated anisotropic diffusion system containing 524 288 rows run on 2048 cores. The figure shows the cost of communication for a number of iterations added to the initialization cost across a range of iteration counts. The diamond markers at iteration count 0 denote the initialization costs of the different methods. Intersections, denoted by dotted vertical lines, indicate the number of iterations at which the higher initialization cost is outweighed by a lower per-iteration communication cost. The crossover points found are 40 iterations for the aggregating implementation, and 22 iterations for the deduplicating implementation. There is minimal overhead associated with the standard neighborhood implementation, as this method simply wraps point-to-point communication. The aggregating implementation demonstrates a higher initialization cost

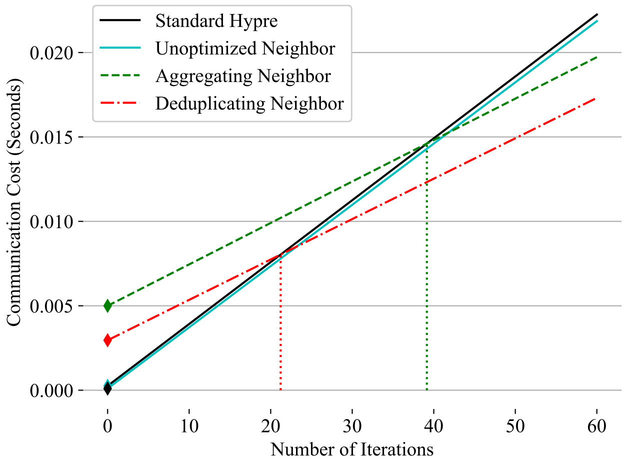


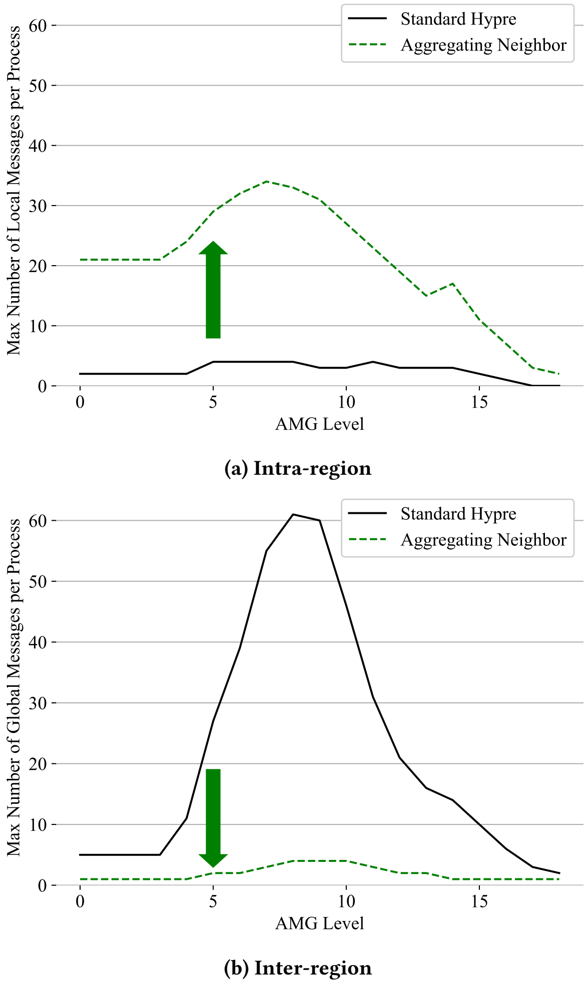
Figure 7: Cost of communication for a range of iteration counts. At each iteration count, cost is calling `MPI_Neighbor_alltoallv_init` once per level of the AMG hierarchy, plus calling `MPI_Start` and `MPI_Wait` once per AMG level per iteration. The diamond markers indicate the initialization cost of each method, and the vertical dotted lines indicate intersections. This problem is a rotated anisotropic diffusion system containing 524 288 rows run on 2048 cores

than the deduplication implementation because the former simply wraps the latter. The aggregating initialization time could be further reduced by implementing it directly. The overheads associated with aggregated communication techniques are due to forming the aggregated path of communication and load balancing. Note, as this cost is only incurred once per communication pattern, more significant initialization overheads are acceptable for higher iteration counts. For communication with fewer iterations, however, simpler aggregation techniques would be necessary to reduce initialization overheads.

4.3 Per-Level Analysis

Algebraic multigrid requires sparse matrix operations to be performed across a hierarchy of levels, with each level decreasing in dimension but often increasing in density. As a result, the communication pattern and created graph topology are unique to each level and communication dominates coarse levels near the middle of the hierarchy. Locality-aware neighbor collectives reduce the inter-region message count and sizes in exchange for additional intra-region communication. This section analyzes the impact of locality-aware neighborhood collectives on each level of a rotated anisotropic diffusion hierarchy. The fine-level system contains 524 288 split across 2048 cores. Test runs of hypre were performed on this problem that included printing of message metadata such as number of intra-region and inter-region messages per process, as well as the sizes of inter-region messages.

Figure 8a displays the maximum number of intra-region messages sent by any process on each level of the hierarchy. Locality-aware neighbor collectives greatly increase the intra-region communication requirements, as both initial and received data is redistributed among processes within each region. Figure 8b displays the maximum number of inter-region messages sent by any process on each level of the hierarchy. While locality-aware aggregation greatly increased intra-region message counts, it results in a similar decrease in the more costly inter-region communication for both optimized implementations.



(a) Intra-region

(b) Inter-region

Figure 8: Per-level message counts when performing a SpMV on each level of the hierarchy.

As discussed in Section 3.3, standard and aggregating communication techniques result in data values being communicated multiple times between regions. Deduplicating neighbor collectives eliminate values from being communicated more than once between any region pair. For each AMG level, Figure 9 displays the largest global message size for the aggregating versus deduplicating neighbor collectives. As shown, locality-aware deduplication results in up to a 35% reduction of the maximum size of global messages per process for level 4 of the AMG hierarchy for the same 524 288 row problem tested on 2048 processes.

Figure 10 displays the cost of communication within a sparse matrix-vector multiply on each level of the AMG hierarchy. Fine levels incur minimal communication overheads, as they are relatively

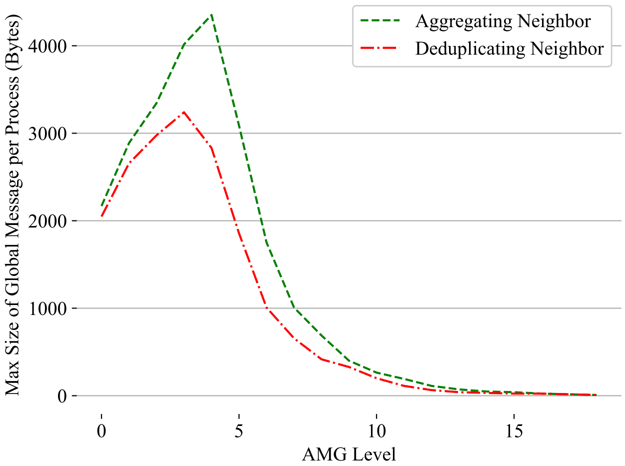


Figure 9: Per-level inter-region message sizes when performing a SpMV on each level of the hierarchy.

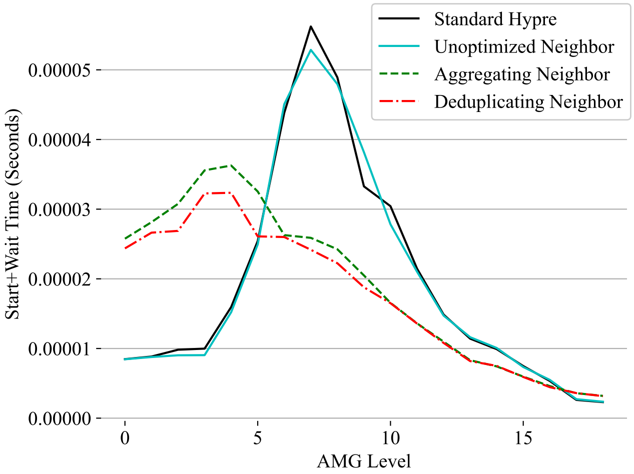
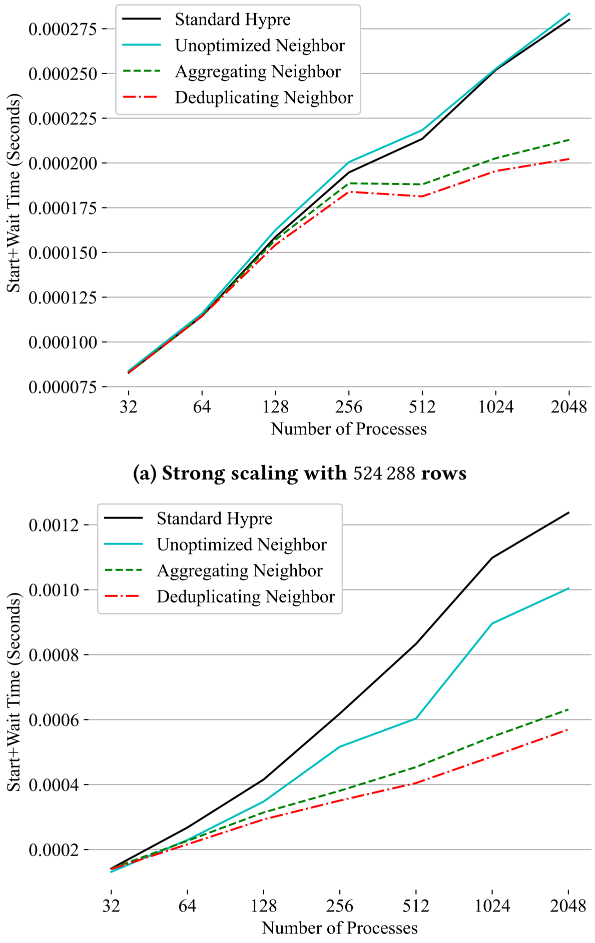


Figure 10: The cost of communicating data during a SpMV on each level of a rotated anisotropic diffusion hierarchy. The fine-level system contains 524 288 **rows split across** 2048 **cores.**



(a) Strong scaling with 524 288 **rows**

(b) Weak scaling with 16 384 **rows per process**

Figure 11: Cost of communication within SpMV for a rotated anisotropic diffusion system.

sparse. Overheads incurred during local redistributions of data increases the cost of locality-aware neighborhood collectives over standard point-to-point communication on these levels. However, as per-level costs increase on the coarse levels, locality-aware aggregation techniques pay off, with optimized neighborhood collectives greatly outperforming standard communication near the middle of the hierarchy. Finally, there are additional benefits to removing values from being communicated multiple times between a single set of regions. Note, the coarsest levels are small enough in dimension that few processes participate in communication, resulting in minimal differences between communication strategies.

4.4 Scaling Analysis

The cost of communication, and performance of the various neighborhood collective implementations, varies with problem scale. This section analyzes the cost of communicating within a SpMV on every level of the AMG hierarchy at various scales. At each scale, the timing is a sum of the times required to perform SpMV communication on each level of the hierarchy at the given scale. The aggregating and deduplicating neighborhood results use the standard communication strategy on finer levels when it outperforms the locality-aware optimizations, summing up the least expensive of standard communication and the given optimized neighbor collective at each step. This demonstrates the maximum possible improvement over standard communication techniques. To achieve this performance, however, a selection strategy, such as a simple performance model, is needed to dynamically choose the optimal neighborhood collective implementation for a given communication pattern.

Figure 11a presents a strong scaling study of communication costs for a rotated anisotropic diffusion system with 524 288 rows split across process counts ranging from 32 to 2048.

The unoptimized neighborhood collective performs similarly to the standard point-to-point communication within Hypre, with both strategies communicating equivalent message counts and sizes. The aggregating neighbor collective significantly improves the scalability of this communication, achieving a speedup of 1.32x over standard communication at 2048 processes. The deduplicating neighbor collectives achieve an additional 0.07x speedup by reducing the size of inter-region communication. As the problem is strongly scaled, the impact of the locality-aware neighborhood collectives increases, indicating that the optimized neighbor collective have increasingly large impacts and message counts increase, and per-message sizes decrease.

Figure 11b presents weak scaling results for the communication cost of a rotated anisotropic diffusion hierarchy with 16 384 rows per process, scaling from 32 to 2048 processes. The weak scaling study shows that the impact of locality-aware aggregation increases with process count. As a larger number of processes are performing communication, there is an additional benefit to reducing duplicate messages between processes. For the weakly scaled problem at 2048 cores, locality-aware aggregation results in a speedup of 1.96x, while reducing duplicate messages provides an additional 0.21x speedup.

5 CONCLUSIONS AND FUTURE DIRECTIONS

Persistent neighborhood collectives provide the interface for locality-aware optimizations to be efficiently implemented within MPI. Standard point-to-point communication can be efficiently replaced with neighbor collectives, incurring only an additional overhead associated with graph creation. However, while forming the topology communicator has a significant cost at scale, this cost is only incurred once and then amortized over all iterations of communication. Methods are in development to avoid this cost with an alternate, more light-weight topology creation interface. Similarly, initialization of locality-aware aggregation techniques can incur large overheads. The persistent neighborhood collective, however, only requires this initialization once per communication pattern, before also being amortized over all iterations of communication.

Locality-aware neighbor collectives, implemented in MPI Advance, significantly improve the performance of irregular communication throughout the coarse levels of Hypre, in which communication requirements are the largest. Furthermore, the optimized neighbor collectives improve both weak and strong scalability of the solver when the appropriate communication strategy is selected at each level of the hierarchy. Eliminating values from being communicated multiple times between a single set of regions further increases these improvements. Finally, as the neighborhood collective implementations exist within MPI Advance, they are accessible to all applications that are limited by irregular communication, requiring the application only to replace point-to-point communication with neighborhood collective and link with MPI Advance.

While locality-aware neighbor collectives have the potential to greatly improve performance, they also are capable of greatly increasing communication costs, particularly for patterns with fewer communication requirements. As a result, a simple performance measure is needed within the neighborhood collective to dynamically select the optimal communication strategy. Furthermore, there are many existing aggregation techniques for locality-aware communication not discussed in this paper. Additional aggregation strategies should be added into MPI Advance, allowing for the dynamic selection not only of locality-aware aggregation, but also of the optimal type of aggregation. Finally, other optimizations should also be added within the implementations of neighborhood collectives. For instance, large messages have been optimized separately with both locality-aware methods and partitioned communication [12]. The combination of these optimizations, partitioning locality-aware messages, can have an even larger impact on communication requirements.

Currently, neighborhood collective implementations optimize only inter-CPU communication. State-of-the-art computers such as Lassen, however, consist of heterogeneous nodes with multiple GPUs per node. Many applications, such as Hypre, achieve full performance through acceleration on these GPUs, relying on inter-GPU communication. Neighborhood collective strategies can be extended to optimize inter-GPU communication, not only dynamically selecting the optimal locality-aware strategy, but also determining whether to communicate data directly between GPUs, to first copy to CPUs, or to copy a portion of the data to each available CPU core, allowing each to communicate with a smaller subset of regions.

ACKNOWLEDGMENTS

This work was performed with partial support from the National Science Foundation under Grant No. CCF-2151022, the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966, and under the auspices of the US Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-854677).

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation and the U.S. Department of Energy's National Nuclear Security Administration.

REFERENCES

[1] [n. d.]. HYPRE: High performance preconditioners. http://www.llnl.gov/CASC/hypre/.

[2] Amotz Bar-Noy and Shlomo Kipnis. 1992. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*. 13–22.

[3] Luc Berger-Vergiat, Christian A. Glusa, Jonathan J. Hu, Matthias Mayr, Andrey Prokopenko, Christopher M. Siefert, Raymond S. Tuminaro, and Tobias A. Wiesner. 2019. *MueLu User's Guide*. Technical Report SAND2019-0537. Sandia National Laboratories.

[4] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. 2012. Mapping Applications with Collectives over Sub-communicators on Torus Networks. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Salt Lake City, Utah) *(SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 97, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389128

[5] Amanda Bienz, Shreeman Gautam, and Amun Kharel. 2022. A Locality-Aware Bruck Allgather. In *Proceedings of the 29th European MPI Users' Group Meeting* (Chattanooga, TN, USA) *(EuroMPI/USA'22)*. Association for Computing Machinery, New York, NY, USA, 18–26. https://doi.org/10.1145/3555819.3555825

[6] Amanda Bienz, William D. Gropp, and Luke N. Olson. 2018. Improving Performance Models for Irregular Point-to-Point Communication. In *Proceedings of the 25th European MPI Users' Group Meeting, Barcelona, Spain, September 23-26, 2018*. 7:1–7:8. https://doi.org/10.1145/3236367.3236368

[7] Amanda Bienz, William D. Gropp, and Luke N. Olson. 2020. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 547–561. https://doi.org/10.1177/1094342020925535

arXiv:https://doi.org/10.1177/1094342020925535

[8] Amanda Bienz, Luke N. Olson, and William D. Gropp. 2019. Node-Aware Improvements to Allreduce. In *Proceedings of ExaMPI 2019*. IEEE, United States, 19–28. https://doi.org/10.1109/ExaMPI49596.2019.00008

[9] Amanda Bienz, Luke N. Olson, and William D. Gropp. 2019. Node aware sparse matrix-vector multiplication. *J. Parallel and Distrib. Comput.* 130 (2019), 166 – 178. https://doi.org/10.1016/j.jpdc.2019.03.016

[10] Amanda Bienz, Luke N. Olson, William D. Gropp, and Shelby Lockhart. 2021. Modeling Data Movement Performance on Heterogeneous Architectures. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622742

[11] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Burkhard Steinmacher-Burow, Anamitra Choudhury, Yogish Sabharwal, Swati Singhal, and Jeffrey J. Parker. 2012. Looking under the hood of the IBM Blue Gene/Q network. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–12. https://doi.org/10.1109/SC.2012.72

[12] Matthew G.F. Dosanjh, Andrew Worley, Derek Schafer, Prema Soundararajan, Sheikh Ghafoor, Anthony Skjellum, Purushotham V. Bangalore, and Ryan E. Grant. 2021. Implementation and evaluation of MPI 4.0 partitioned communication libraries. *Parallel Comput.* 108 (2021), 102827. https://doi.org/10.1016/j.parco.2021.102827

[13] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. 2005. Overview of the Blue Gene/L system architecture. *IBM Journal of Research and Development* 49, 2.3 (2005), 195–212. https://doi.org/10.1147/rd.492.0195

[14] S. Mahdieh Ghazimirsaeed, Qinghua Zhou, Amit Ruhela, Mohammadreza Bayatpour, Hari Subramoni, and Dhabaleswar K. DK Panda. 2020. A Hierarchical and Load-Aware Design for Large Message Neighborhood Collectives. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41405.2020.00038

[15] Richard Graham, Manjunath Gorentla Venkata, Joshua Ladd, Pavel Shamis, Ishai Rabinovitz, Vasily Filipov, and Gilad Shainer. 2011. Cheetah: A Framework for Scalable Hierarchical Collective Operations. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 73–83. https://doi.org/10.1109/CCGrid.2011.42

[16] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer International Publishing, Cham, 330–350.

[17] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. Modeling MPI Communication Performance on SMP Nodes: Is It Time to Retire the Ping Pong Test. In *Proceedings of the 23rd European MPI Users' Group Meeting* (Edinburgh, United Kingdom) *(EuroMPI 2016)*. Association for Computing Machinery, New York, NY, USA, 41–50. https://doi.org/10.1145/2966884.2966919

[18] Masayuki Hatanaka, Atsushi Hori, and Yutaka Ishikawa. 2013. Optimization of MPI Persistent Communication. In *Proceedings of the 20th European MPI Users' Group Meeting* (Madrid, Spain) *(EuroMPI '13)*. Association for Computing Machinery, New York, NY, USA, 79–84. https://doi.org/10.1145/2488551.2488566

[19] Van Emden Henson and Ulrike Meier Yang. 2002. BoomerAMG: A Parallel Algebraic Multigrid Solver and Preconditioner. *Appl. Numer. Math.* 41, 1 (April 2002), 155–177. https://doi.org/10.1016/S0168-9274(01)00115-5

[20] Mert Hidayetoğlu, Tekin Bicer, Simon Garcia de Gonzalo, Bin Ren, Vincent De Andrade, Doga Gursoy, Raj Kettimuthu, Ian T. Foster, and Wen-mei W. Hwu. 2020. Petascale XCT: 3D Image Reconstruction with Hierarchical Communications on Multi-GPU Nodes. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13. https://doi.org/10.1109/SC41405.2020.00041

[21] Jonathan Hines. 2018. Stepping up to Summit. *Computing in Science & Engineering* 20, 2 (2018), 78–82. https://doi.org/10.1109/MCSE.2018.021651341

[22] Daniel J. Holmes, Anthony Skjellum, and Derek Schafer. 2020. Why is MPI (Perceived to Be) so Complex? Part 1—Does Strong Progress Simplify MPI?. In *Proceedings of the 27th European MPI Users' Group Meeting* (Austin, TX, USA) *(EuroMPI/USA '20)*. Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3416315.3416318

[23] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. 2009. Designing multi-leader-based Allgather algorithms for multi-core clusters. In *2009 IEEE International Symposium on Parallel & Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2009.5160896

[24] N.T. Karonis, B.R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. 2000. Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*. 377–384. https://doi.org/10.1109/IPDPS.2000.846009

[25] Shelby Lockhart, Amanda Bienz, William Gropp, and Luke Olson. 2023. Performance Analysis and Optimal Node-Aware Communication for Enlarged Conjugate Gradient Methods. *ACM Trans. Parallel Comput.* 10, 1, Article 2 (mar 2023), 25 pages. https://doi.org/10.1145/3580003

[26] Shelby Lockhart, Amanda Bienz, William D. Gropp, and Luke N. Olson. 2023. Characterizing the performance of node-aware strategies for irregular point-to-point communication on heterogeneous architectures. *Parallel Comput.* 116 (2023), 103021. https://doi.org/10.1016/j.parco.2023.103021

[27] Teng Ma, George Bosilca, Aurelien Bouteiller, and Jack Dongarra. 2012. HierKNEM: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 970–982.

[28] Seyed H. Mirsadeghi and Ahmad Afsahi. 2016. Topology-Aware Rank Reordering for MPI Collectives. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1759–1768. https://doi.org/10.1109/IPDPSW.2016.139

[29] Shuo Ouyang, Dezun Dong, Yemao Xu, and Liquan Xiao. 2021. Communication optimization strategies for distributed deep neural network training: A survey. *J. Parallel and Distrib. Comput.* 149 (2021), 52–65. https://doi.org/10.1016/j.jpdc.2020.11.005

[30] P. Patarasuk and X. Yuan. 2007. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370405

[31] David Schneider. 2022. The Exascale Era is Upon Us: The Frontier supercomputer may be the first to reach 1,000,000,000,000,000,000 operations per second. *IEEE Spectrum* 59, 1 (2022), 34–35. https://doi.org/10.1109/MSPEC.2022.9676353

[32] Barry Smith. 2011. *PETSc (Portable, Extensible Toolkit for Scientific Computation)*. Springer US, Boston, MA, 1530–1539. https://doi.org/10.1007/978-0-387-09766-4_87

[33] Andreas Thune, Sven-Arne Reinemo, Tor Skeie, and Xing Cai. 2023. Detailed Modeling of Heterogeneous and Contention-Constrained Point-to-Point MPI Communication. *IEEE Transactions on Parallel and Distributed Systems* 34, 5 (2023), 1580–1593. https://doi.org/10.1109/TPDS.2023.3253881

[34] Jesper Larsson Träff. 2006. Efficient Allgather for Regular SMP-Clusters. In *Proceedings of the 13th European PVM/MPI User's Group Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Bonn, Germany) *(EuroPVM/MPI'06)*. Springer-Verlag, Berlin, Heidelberg, 58–65. https://doi.org/10.1007/11846802_16

[35] Jesper Larsson Träff and Sascha Hunold. 2020. Decomposing MPI Collectives for Exploiting Multi-lane Communication. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. 270–280. https://doi.org/10.1109/CLUSTER49012.2020.00037