

Optimizing Neighbor Collectives with Topology Objects

1st Gerald Collom

*Department of Computer Science
University of New Mexico
Albuquerque, USA
geraldc@unm.edu*

2nd Derek Schafer

*Department of Computer Science
University of New Mexico
Albuquerque, USA
dschafer1@unm.edu*

3rd Amanda Bienz

*Department of Computer Science
University of New Mexico
Albuquerque, USA
bienz@unm.edu*

4th Patrick Bridges

*Department of Computer Science
University of New Mexico
Albuquerque, USA
patrickb@unm.edu*

5th Galen Shipman

*Department of Computer Science
Los Alamos National Laboratory
Los Alamos, USA
gshipman@lanl.gov*

Abstract—Many HPC applications implement non-cartesian neighbor data exchanges using MPI point-to-point operations rather than utilizing native MPI neighbor collective methods. Each application must therefore implement their own communication optimizations, rather than leveraging any optimizations that could be provided by MPI. While an interface for such optimizations is provided within MPI through neighborhood collectives, applications avoid these methods due to the lack of performance optimizations within them along with large costs associated with graph communicator formation. This paper presents a novel approach for creating local, non-cartesian topology objects that provides finer control over the aforementioned setup costs. Any additional setup costs, such as initializing per-iteration optimizations, can then be deferred until additional information is available, such as within persistent initialization calls. This paper describes our implementation within an MPI extension library and demonstrates the effectiveness of our approach in simple benchmarks and real-world applications.

Index Terms—MPI, high performance computing, neighbor collectives

I. INTRODUCTION

MPI non-cartesian neighbor collectives and graph communicators were designed to enable and optimize irregular communication performance in parallel scientific applications [1]. To use these primitives, an application programmer must first create a communicator with an associated topology for the desired irregular communication pattern before using neighbor collective calls (e.g. `MPI_Neighbor_alltoallv`) to exchange data. A wide range of neighbor collective optimizations have been proposed to improve the performance of sparse data exchanges [2], [3], utilizing communication topology and volumes to minimize associated costs.

Unfortunately, the design of these abstractions in MPI 4.0 is inefficient if the communication topology changes frequently. In particular, MPI communicator creation can be costly, often needing multiple collective operations between participating processes, regardless of the MPI topology pattern used [4]. This imposes a large overhead on changing communication

topologies, something that occurs in widely-used numerical methods that rely on irregular communication, including algebraic multigrid (AMG), adaptive mesh refinement (AMR), and n-body particle codes. As a result, neighbor collectives and graph communicators are generally not used in these codes, despite existing irregular communication bottlenecks.

This paper presents the design and evaluation of a novel *topology* abstraction for MPI as well as corresponding MPI neighbor collective and communicator operations. The overall goal of this abstraction is to make MPI neighbor collectives and the optimizations previously developed for them effective in real high performance computing (HPC) applications. To achieve this goal, this paper describes the following contributions after describing essential background information:

- An analysis of the requirements for a useful, efficient topology abstraction resulting from the examination and benchmarking of multiple production applications that rely on irregular communication (Section III);
- The semantics and specification of a new first class MPI Communication Topology abstraction and related portions of the MPI API that meet these requirements (Sections IV-A and IV-B);
- An initial implementation of the resulting API in the MPI Advance [5] prototyping library (Section IV-C);
- An evaluation of the performance of a sparse matrix-vector multiplication benchmark utilizing neighborhood collectives, comparing this novel topology creation to MPI distributed graph communicators (Section V-B); and
- An assessment of the impact of this new abstraction on the xRAGE and *hypre* production codes (Section V-C).

While the proposed approach addresses the primary problems that motivated its design, this alternative approach and our current evaluation of it do have some limitations:

- The new topology abstraction increases performance by avoiding the communications associated with communica-

cator creation, resulting in somewhat different ordering semantics from the previous approach. Despite this limitation, we believe that our proposed approach is preferable to the alternative, as it lets the programmer chose when and if to pay the cost of the stronger semantics associated with communicator creation.

- Our current topology object interface has not been expanded to provide the full generality of topologies provided by communicator process topologies, currently focusing only on distributed graph topologies needed by neighbor communicators.
- Limitations of the vendor MPI implementation on some test systems prevented us from providing apples-to-apples performance comparisons in *hypre* between the use of traditional MPI communicator process topologies and the enhanced communication topology abstraction we describe, resulting in a comparison only with *hypre*'s hand-optimized point-to-point sparse data exchange.

II. BACKGROUND AND RELATED WORK

A wide range of approaches for performing irregular, dynamic communication have been implemented throughout high performance computing programs. Many applications and application-focused libraries implement custom application-specific primitives for this. Because of the prevalence of such exchanges, more general communication libraries and standards, for example MPI and NCCL, have also added support for such exchanges. In the remainder of this section, we provide background on these widely-used communication primitives and implementations and discuss their relationship with the research described in this paper.

A. Application-Defined Sparse Data Exchanges

Algorithm 1: p2p

```
Input: rank           {ID of a given process}
send_info    {Processes rank sends to, and what to send to each}
recv_info    {Processes rank receives from, and what to receive}
comm         {MPI communicator}

// Post all receives
for  $i \leftarrow 0$  to  $recv\_info.n$  do
  MPI_Irecv recv_info.counts[i] values at
  recv_info.buf[recv_info.displs[i]]
  from recv_info.procs[i]

// Post all sends
for  $i \leftarrow 0$  to  $send\_info.n$  do
  MPI_Isend send_info.counts[i] values at
  send_info.buf[send_info.displs[i]]
  to send_info.procs[i]

// Wait for all communication to complete
MPI_Waitall
```

Most applications that perform dynamic, irregular sparse data exchanges, such as those that require sparse matrix

operations, implement these exchanges through point-to-point messages. These implementations generally use a variant of the simple implementation shown in Algorithm 1, and in some cases overlap message unpacking with waiting for message arrival. For instance, the widely-used sparse linear solvers *hypre* BoomerAMG [6], Trilinos [7], and PETSc [8], as well as the xRAGE [9] and Parthenon [10] adaptive mesh refinement codes, each use this approach.

Algorithm 2: persistent_p2p

```
Input: rank           {ID of a given process}
send_info    {Processes rank sends to, and what to send to each}
recv_info    {Processes rank receives from, and what to receive}
comm         {MPI communicator}

// Initialize all receives
for  $i \leftarrow 0$  to  $recv\_info.n$  do
  MPI_Recv_init recv_info.counts[i]
  values at
  recv_info.buf[recv_info.displs[i]]
  from recv_info.procs[i]

// Initialize all sends
for  $i \leftarrow 0$  to  $send\_info.n$  do
  MPI_Send_init send_info.counts[i]
  values at
  send_info.buf[send_info.displs[i]]
  to send_info.procs[i]

// Start all communication
MPI_Startall

// Wait for all communication to complete
MPI_Waitall
```

More recently, some applications have also leveraged support for persistent communication in libraries like MPI to potentially reduce communication overhead; the result, now included in some of the previously mentioned solvers, is described in Algorithm 2. Importantly, however, these algorithms are all implemented *above* the underlying communication library and this cannot easily take advantage of underlying communication topology information. Communication library abstractions for performing these exchanges, including both the one described in this paper and in the following subsections, avoid such limitations.

B. MPI-Based Sparse Data Exchanges

Because of the prevalence of such data exchanges, MPI added support for sparse data exchanges through the introduction of *neighbor collectives* in MPI 3.0. These primitives are largely based on an original proposal that leverages MPI process topologies [11], allowing the process topology to optionally be associated with an MPI communicator and used to determine communication partners for data exchanges. Providing sparse data exchanges via the MPI standard enables optimizations to be added within MPI implementations rather than relying on application programmers to each write their

own. It also, however, ties applications that use these implementations to the design choices of the implementation and their performance implications.

Algorithm 3: neighbor

```

Input: rank           {ID of a given process}
send_info    {Processes rank sends to, and what to send to each}
recv_info    {Processes rank receives from, and what to receive}
comm         {MPI communicator}

// Declare variables
MPI_Comm xcomm
int reorder           {allocate to 0 or 1}
MPI_Info info        {allocate or use MPI_INFO_NULL}

// Create neighborhood communicator
MPI_Dist_graph_create_adjacent(comm,
    recv_info.n, recv_info.procs,
    recv_info.counts, send_info.n,
    send_info.procs, send_info.counts,
    info, reorder, &xcomm)

// Perform data exchange
MPI_Neighbor_alltoallv(send_info.buf,
    send_info.counts, send_info.displs,
    send_info.type, recv_info.buf,
    recv_info.counts, recv_info.displs,
    recv_info.type, xcomm)

// Free neighbor communicator
MPI_Comm_free(&xcomm)

```

Neighborhood collectives within MPI fundamentally provide an interface that wraps and schedules point-to-point communication between processes on a communicator; that is, each process communicates with a subset of other processes. Any directed graph-based communication, as required within sparse matrix operations, can be implemented with neighborhood collectives, as demonstrated in Algorithm 3.

Persistent versions of neighbor collectives, described in Algorithm 4, were added to MPI 4.0. Similar to persistent point-to-point operations, persistent neighbor collectives allow for further optimizations and error checking within MPI, incurring an initial setup overhead before performing a neighborhood of data exchanges persistently. The operations within Algorithm 4 can likewise be interleaved throughout application code with `MPI_Start` and `MPI_Wait` called many times.

Unlike point-to-point operations, MPI neighbor collectives lack user-provided tags; this eases their usage but also requires all processes to issue neighbor collectives in identical orders. In contrast, the application-built neighbor exchanges described above leverage the power of point-to-point communication tags. The alternative API described in this paper preserves a tag-free API; the implementation can use a reserved portion of the tag space to avoid conflicting with user tags.

Finally, the opportunity to detect programmer error in communicator creation and persistent collective initialization routines is theoretically strong but in practice very

Algorithm 4: persistent_neighbor

```

Input: rank           {ID of a given process}
send_info    {Processes rank sends to, and what to send to each}
recv_info    {Processes rank receives from, and what to receive}
comm         {MPI communicator}

// Declare variables
MPI_Comm xcomm
MPI_Request request
int reorder           {allocate to 0 or 1}
MPI_Info info        {allocate or use MPI_INFO_NULL}

// Create neighborhood communicator
MPI_Dist_graph_create_adjacent(comm,
    recv_info.n, recv_info.procs,
    recv_info.counts, send_info.n,
    send_info.procs, send_info.counts,
    info, reorder, &xcomm)

// Initialize data exchange
MPI_Neighbor_alltoallv_init(
    send_info.buf, send_info.counts,
    send_info.displs, send_info.type,
    recv_info.buf, recv_info.counts,
    recv_info.displs, recv_info.type,
    xcomm, info, &request)

// Start data exchange
MPI_Start(&request)

// Complete data exchange
MPI_Wait(&request)

// Free persistent request
MPI_Request_free(&request)

// Free neighbor communicator
MPI_Comm_free(&xcomm)

```

limited. The collective nature of communicator creation and persistent collective creation allows, for example, `MPI_Dist_graph_create_adjacent` to check that if process p sends to process q , process q also receives from process p . Modern MPI implementations such as OpenMPI and MPICH, however, do not perform such checks. The alternative approach to topology specification described in this paper likewise defers error checking to the user.

C. Limitations of MPI Neighbor Collectives

Existing HPC applications rarely use neighborhood or persistent neighborhood collectives, despite the fact that they wrap and potentially optimize common point-to-point communication patterns. The main limitation of neighbor collectives is that they require the use of a communicator with an associated process topology, such as that constructed through the use of `MPI_Dist_graph_create_adjacent`. While the use of communicators for encapsulating communication topologies provides MPI with optimization options, as noted by Hoeffler, et. al [11], it also incurs significant communication overhead due to the fact that communicator creation is a

distributed, collective MPI operation. As a result, applications that must periodically *change* communication topologies may not be able to adequately amortize these overheads, even with improved sparse communication performance.

Another challenge to the uptake of neighbor collectives is that many applications need to perform bidirectional data exchanges. For instance, many applications require both a data exchange implemented as: `neighbor(rank, send_info, recv_info, comm);` as well as a reverse data exchange implemented as `neighbor(rank, recv_info, send_info, comm);`. Unfortunately, the current neighbor collective API does not support communication in this inverse direction. As a result, applications seeking to use neighbor collectives for bidirectional sparse data exchanges must create *two* communicators, incurring communicator construction overheads twice. In contrast, this paper presents a novel approach for avoiding this overhead and allowing applications to efficiently utilize neighbor collectives.

III. REQUIREMENT AND DESIGN TRADE-OFFS

While neighbor communications offer a wide range of potential optimization opportunities, their current design makes them prohibitively expensive for use in a wide range of modern applications. To address this issue, we considered multiple requirements and potential approaches to redesigning support for sparse data exchanges in MPI to better support applications with dynamic communication topologies.

A. Requirements Analysis

Based on our analysis of multiple applications and libraries, along with the literature optimizing neighbor collectives, we identified two key requirements of any solution to the performance issues that limit the usability of neighbor collectives in applications with dynamic, sparse data exchanges:

- 1) The ability to modify communication topology without incurring the communication and synchronization overheads of communicator construction or collective communication to optimize cases where a communication pattern will not be reused often; and
- 2) The ability to optimize neighbor communication patterns that are used repeatedly, including optimizations that take into account the distributed structure of the communication pattern.

These two requirements were initially in conflict when neighbor collectives were originally designed. This is because communicators were the only persistent object in MPI-3.0 that encapsulated distributed knowledge of process and communication topology. As a result, the original design of neighbor collectives [11], which noted the performance advantages of constructing distributed schedules for neighbor collectives, had little choice but to tie topologies to communicators.

The introduction of persistent collectives in MPI-4.0, however, provides an opportunity to avoid the trade-off that originally motivated the usage of communicators for sparse

communication. Persistent collectives are designed to be created when a code expects a distributed communication pattern to be used repeatedly. As such, the use of these primitives both informs the MPI implementation of the opportunity to amortize away optimization overheads and provides access to the distributed information needed to perform optimizations such as scheduling [11] or aggregating [3] sparse data exchanges.

B. Design Choices

Large overheads associated with the per-topology communicator construction can be eliminated through either of two high-level design alternatives, *changeable process topologies* and *first class communication topologies*. Changeable process topologies, or editing the topology after it is associated with an active communicator, would reduce costs associated with communicator construction while limiting the scope of changes to only the process topology portion of the MPI standard. As a result, the neighbor collective API would remain unchanged. However, the impact of changing process topologies on an active communicator whose rank mapping may have been optimized based on that process topology is unclear. In addition, it is unclear if changing a process topology can be restricted to being a local or non-collective operation. However, this analysis identifies a novel insight: *A key shortcoming of the MPI-3.0 neighbor collective design is that it conflates practical communication topologies with logical process topologies.*

Due to the shortcomings associated with changeable topology objects, this paper is focused on first-class communication topology objects for MPI that are distinct from the process topologies associated with MPI communicators. We additionally chose to have all topology operations be *local* operations. In other words, creating and destroying communication topologies *requires no communication*. Because topology creation and destruction is thus inexpensive, this approach also avoided the need to create a large number of operations for changing the distributed information stored in a topology object.

This decision has two key implications besides the direct performance improvements to topology optimizations:

- 1) It allows the runtime to defer topology optimization decisions to when neighbor collective operations, either persistent or not, are created. This is important because additional information, such as the amount of data being communicated and if the operation is expected to be executed multiple times, is available at this point.
- 2) It prevents topologies from checking for consistent distributed topology states. For example, creating inconsistent topology objects between processes, such as a topology in which process *p* will send to process *q*, but *q* does not receive from *p*, is *erroneous*. Note, however, that the MPI standard [12] does not require this check when creating distributed graph communicators, and state-of-the-art MPI implementations such as OpenMPI [13] and mpich [14] do not check for this inconsistency.

IV. THE COMMUNICATION TOPOLOGY OBJECT

The improved topology management presented in this paper makes communication topologies (hereafter, *topology objects*) first-class objects in MPI. Unlike existing communicator-based process topologies, the creation and destruction of topology objects are *local* MPI operations, requiring no communication. In the remainder of this section, we describe the semantics and performance implications of replacing topology communicators with topology objects, the API for performing neighbor collectives with topology objects, and an implementation of this API as an MPI overlay library.

A. Performance and Semantics Implications

Because topology object creation and destruction are local operations, our proposed design significantly reduces the minimum overhead required to create, manipulate, and destroy these objects. The overheads of frequent collective usage in MPI applications, even for optimized, scalable collectives, are well-studied [15], [16]. Eliminating these overheads provides applications potentially significant performance gains, as displayed through the results presented in Section V.

This optimization is not without cost. Removing synchronization and communication from communication topology creation imposes a non-trivial requirement on applications to write error-free code. In particular, if processes have inconsistent topology information, they may send to or try to receive from processes that are not expecting to do so, resulting in mismatched communication at best and deadlock at worst.

Based on our experience with real applications, however, we do not view this as a significant limitation. Applications often already build or explicitly synchronize this information before creating neighbor communication patterns using either collectives or custom neighbor exchanges. In addition, applications that need to verify topology correctness for debugging purposes (e.g., because a neighbor exchange hangs), can use existing MPI collectives such as `MPI_Alltoall` to exchange topology connectivity and easily verify topology information. In contrast, a primitive that required MPI to verify the correctness of a distributed topology would *mandate* that this cost be paid on every topology manipulation.

Another important semantic difference between our proposed approach and the original communicator-based approach is a change in communication ordering constraints. In the previous approach, neighbor collectives with different communication topologies were *required* to be performed on distinct communicators. As a result, such communications were forced to be on independently-ordered communicators. In contrast, our proposed approach allows the application to decide whether communications with different topologies of the same processes should or should not be on the same communicator. Applications that seek to preserve the independence of communications required in the previous approach (e.g. barriers on `MPI_COMM_WORLD` being on distinct communicators from neighbor collectives) can maintain this behavior

by creating additional communicators; applications that do not require this separation are no longer required to do so.

B. API Design

Currently, MPI forms all topology information on top of a new communicator, requiring the initial communicator to be available at the time the application calls `MPI_Dist_graph_create_adjacent`. The design of the presented API focuses on creating and using a concrete topology object instead of an additional communicator. This change significantly impacts the reliance on MPI communicators, by altering the first instance that an MPI communicator is needed. The novel topology objects can be utilized within existing parallel application by simply replacing the topology communicator with the equivalent `MPIX_Topo` topology object.

The topology object creation call mirrors the MPI distributed graph creation call, accepting all of the same parameters except for the input and output communicators. Instead, this local function now returns only a topology construct that contains copies of the provided inputs; while this operation is local, it does require memory allocation and data copies proportional to the number of number of processes involved. Accessing the members of this construct can be done using API calls that mirror their MPI counterparts. For example, we provide a variation of `MPI_Dist_graph_neighbors` that uses the `MPIX_Topo` object to return the provided source and destination processes. The topology object is also required to be provided as an input parameter to any neighborhood collective operations, along with a communicator to use for communication. These parameters allow MPI to apply the topology pattern on the given communicator. As with any MPI construct, a corresponding freeing API is also provided. The full set of APIs provided by our library is shown in Table I, along with any changes from their MPI counterparts.

C. Implementation in MPI Advance

All proposed changes exist within an external *MPI eXtension* library and are a part of the MPI Advance collection of MPI optimizations [5]. As a result, the library can be used alongside any MPI implementation that supports basic point-to-point operations. Similarly, the proposed persistent neighborhood collectives can be used with any MPI implementation that also supports persistent point-to-point communication.

However, the lack of full integration with an MPI implementation requires additional changes to applications in order to leverage the proposed APIs. Namely, there is no portable way to create an abstract `MPI_Request` for custom communication¹. Therefore, our non-blocking and persistent neighborhood collectives APIs return an `MPIX_Request` object. This object cannot be mixed with other `MPI_Request` objects and requires specific `MPIX` APIs to be used. However, if our approach were to be added to an existing MPI implementation, a unique `MPIX_Request` object is no longer needed.

¹While generalized requests do exist in MPI, their use would require additional calls from the application compared to using `MPIX` objects.

TABLE I
SET OF APIs PROVIDED BY OUR LIBRARY (INCLUDES APIs THAT NEED A CUSTOM MPIX VERSION)

API	Change
<code>MPIX_Topo_dist_graph_create_adjacent</code>	Replace <code>MPI_Comm</code> with new topology object (<code>MPIX_Topo</code>)
<code>MPIX_Topo_dist_graph_neighbors_count</code>	Replace <code>MPI_Comm</code> with new topology object (<code>MPIX_Topo</code>)
<code>MPIX_Topo_dist_graph_neighbors</code>	Replace <code>MPI_Comm</code> with new topology object (<code>MPIX_Topo</code>)
<code>MPIX_Topo_free</code>	New API for freeing new topology object (<code>MPIX_Topo</code>)
All <code>MPI_Neighbor_X</code> collectives	Add parameter for new topology object (<code>MPIX_Topo</code>)
All <code>MPI_Ineighbor_X</code> collectives	Add parameter for new topology object (<code>MPIX_Topo</code>) Replace <code>MPI_Request</code> with <code>MPIX_Request</code>
All <code>MPI_Neighbor_X_init</code> collectives	Add parameter for new topology object (<code>MPIX_Topo</code>) Replace <code>MPI_Request</code> with <code>MPIX_Request</code>
All request <code>MPI_Request</code> related functions (e.g. <code>MPI_Test</code> , <code>MPI_Request_get_status</code> , <code>MPI_Wait</code>)	Replace <code>MPI_Request</code> with <code>MPIX_Request</code>

Internally, local topology creation is straightforward. The novel topology abstractions are condensed down to wrapper functions, which allocate memory and cache the parameters passed to the topology creation function, as shown in Algorithm 5. While this simplified approach is enough to

Algorithm 5: `topology_neighbor`

```

Input: rank           {ID of a given process}
send_info    {Processes rank sends to, and what to send to each}
recv_info    {Processes rank receives from, and what to receive}
comm          {MPI communicator}

// Declare variables
int reorder      {allocate to 0 or 1}
MPI_Info info    {allocate or use MPI_INFO_NULL}

// Store send and receive in a topology object
MPIX_Topo topo
MPIX_Topo_dist_graph_create_adjacent (
  recv_info.n, recv_info.procs,
  recv_info.counts, send_info.n,
  send_info.procs, send_info.counts,
  info, reorder, &topo)

// Perform data exchange
MPIX_Neighbor_alltoallv(send_info.buf,
  send_info.counts, send_info.displs,
  send_info.type, recv_info.buf,
  recv_info.counts, recv_info.displs,
  recv_info.type, comm, topo)

// Free Topology object
MPIX_Topo_free(&topo)

```

remove synchronization overheads, many other optimization approaches to caching the data could be applied.

The neighborhood and persistent neighborhood collectives are currently implemented as wrappers for Algorithms 1 and 2, respectively. The specific algorithm under these collectives was not the main focus of this work, but aggregation and other data exchange optimizations could be applied internally.

While the lack of a unique channel is only a concern for the collective operations, the implementation of neighborhood collectives within an MPI extension library can incur additional collisions. As noted in Section II-B, MPI neighborhood collec-

tives do not require tags. However, since our collectives wrap existing MPI point-to-point operations, a tag value must be provided with each send and receive. To avoid collisions with application tag usage, MPI Advance can adjust `MPI_TAG_UB` to reserve a portion of the tag space for internal usage if necessary; our current implementation does not yet do so.

D. Application Integration

For applications already using MPI distributed graph topologies, integration with our proposed approach would entail switching some MPI calls to the MPIX versions and keeping track of the `MPIX_Topo` object. For applications implementing a neighbor collective using a hand-crafted set of point-to-point operations, shifting to an explicit topology requires only marginally more effort to coalesce topology parameters into arrays. Our experience is that most applications performing such exchanges implement a well-defined abstraction to make these exchanges for software engineering reasons, limiting the scope of any such required changes. Therefore, the overall level of effort required should be similar to the approach we used to integrate into the xRAGE application, described below.

We illustrate the process required to adapt an application to use our topology abstraction in the Los Alamos xRAGE (Radiation Adaptive Grid Eulerian) code, an Eulerian radiation/hydrodynamics code that has been under active development at LANL for over 30 years [9]. During each timestep, xRAGE refines/de-refines the adaptive mesh and then load balances through the redistribution of cells across all processes in the system. These steps result in changing irregular neighbor communication patterns that are recomputed each timestep. To manage these communication patterns, xRAGE uses “token” communication objects. The token persists the associated communication pattern, allowing it to be used multiple times if needed. xRAGE’s neighbor exchange code uses the approach previously shown in Algorithm 2, but optimized to interleave reduction operations while waiting on receive operations.

We modified xRAGE tokens to use both traditional MPI neighbor communicators and our new topology abstraction by first having token creation create, store, and use either MPI distributed graph communicators or `MPIX_Topo` objects. Because xRAGE tokens support bidirectional communication, this required creating *two* objects for each token. We then

modified xRAGE tokens to use a neighborhood collective (either the MPI or the MPIX version) instead of the hand-built point-to-point operations. With this change, the built-in reduction operations performed by xRAGE tokens must now wait to be performed until after the collective finishes. Finally, the two communicators or `MPIX_Topo` objects are freed when xRAGE discards a token object.

Due to token objects having well-defined methods for both the setup and communication phases, all of the above changes were able to be contained to their respective methods. For example, swapping to the MPIX neighborhood collectives only required altering the method inside the token class responsible for carrying out the neighborhood collective. We followed a similar approach to adapt *hypre* to use these abstractions, and other applications and frameworks with similar primitives should be able to follow a similar approach.

V. PERFORMANCE RESULTS

To evaluate the impact of the communication topology objects on codes with dynamic sparse data exchanges, we conducted benchmark studies and integrated the MPIX implementation of the topology object into multiple production codebases. In the remainder of this section, we describe our experimental setup, present benchmark results that compare the fine-grain performance of the new MPIX topology operations with alternative approaches, and demonstrate the impact of integrating this abstraction into two production codebases that rely on dynamic, sparse data exchanges.

A. Experimental Setup

In general, we compare the performance of three different approaches to describing communication topologies and performing sparse data exchanges:

- 1) Using standard MPI Communicators with associated process topology information and neighborhood collectives, the approach for modern MPI optimizations.
- 2) Using hand-built point-to-point communication to perform sparse data exchanges, the state of the practice for most applications today.
- 3) Using MPIX Communication Topology objects and neighborhood collectives on MPI communicators *without* an associated process topology.

All benchmark and application runs presented in this section used the following systems and MPI implementations:

- **Quartz:** An Intel Xeon E5-2695 v4 CPU system from Lawrence Livermore National Laboratory with 36 cores per node and a Cornelis Networks OmniPath interconnect. The MPI implementations used were OpenMPI 4.1.2 and MVAPICH2 2.3.7.
- **Lassen:** An IBM Power9 and NVIDIA V100 system from Lawrence Livermore National Laboratory with 40 usable cores per node and an Infiniband EDR interconnect. The default system modules for both Spectrum MPI and MVAPICH2 were used for MPI implementations.

- **Chicoma:** An AMD EPYC 7H12 system from Los Alamos National Laboratory with 128 cores per node and a Cray Slingshot 11 network. The MPI implementation used was cray mpich (version 3.4.2), with the setting `FI_CXI_RX_MATCH_MODE` set to `software`.
- **Rocinante:** An Intel Xeon Max 9480 CPU and Intel Xeon 8479 CPU system from Los Alamos National Laboratory, featuring 128 Xeon MAX 9480 nodes with 112 cores per node and 128 GByte of HBM2e Memory and 380 Xeon 8479 nodes with 112 cores per node and 256 GByte DDR5 Memory. Rocinante was used only to run xRAGE due to restrictions on systems on which xRAGE could be run and the limited availability of this system. Cray mpich using the Slingshot 11 network interface was used as the underling MPI implementation.

B. Benchmark Experiments

We created a simple benchmark to test the performance of various approaches to managing sparse data exchanges based on executing a simple sparse matrix-vector multiplication with different input matrices. Sparse matrix-vector multiplications require sparse communication based on the structure of the sparse matrix, so changing matrix also changes the resulting communication pattern. To vary these communication patterns, we used input matrices obtained from the University of Florida sparse matrix collection [17]. In each case, we repeatedly setup communication (creating the `MPIX_Topo` or `MPI_Comm` objects, as appropriate), performed a single sparse matrix-vector multiply and associated communication (e.g. neighbor collective or looped point-to-point operation), and then destroyed the created topology object or communicator. Each communication approach was carried out for 200 iterations, with the maximum average time across all processes recorded. The number of nodes used in the benchmark was doubled until either the system limit or crashes prevented further scaling. At each scale, the benchmark was run three times using the maximum number of processes per node. The averages of each run are shown in Figure 1.

We observe the following results from these experiments:

- For each matrix/MPI/system tuple, all three communication approaches had roughly equal times.
- When using Spectrum MPI, creating MPI topology constructs was considerably slower than any other MPI implementation, regardless of the number of total ranks and the input matrix.
- MPIX topology creation times were at least two orders of magnitude cheaper than the corresponding communication costs and traditional MPI topology creation costs.
- The MPIX topology creation and communication times were largely equivalent to the point-to-point implementation, though this is expected as they have essentially similar (unoptimized) implementations.

From these results, we conclude that the proposed new approach to topology creation (1) offers significant performance advantages over the previous communication and (2) traditional communicator-based topologies would have to achieve

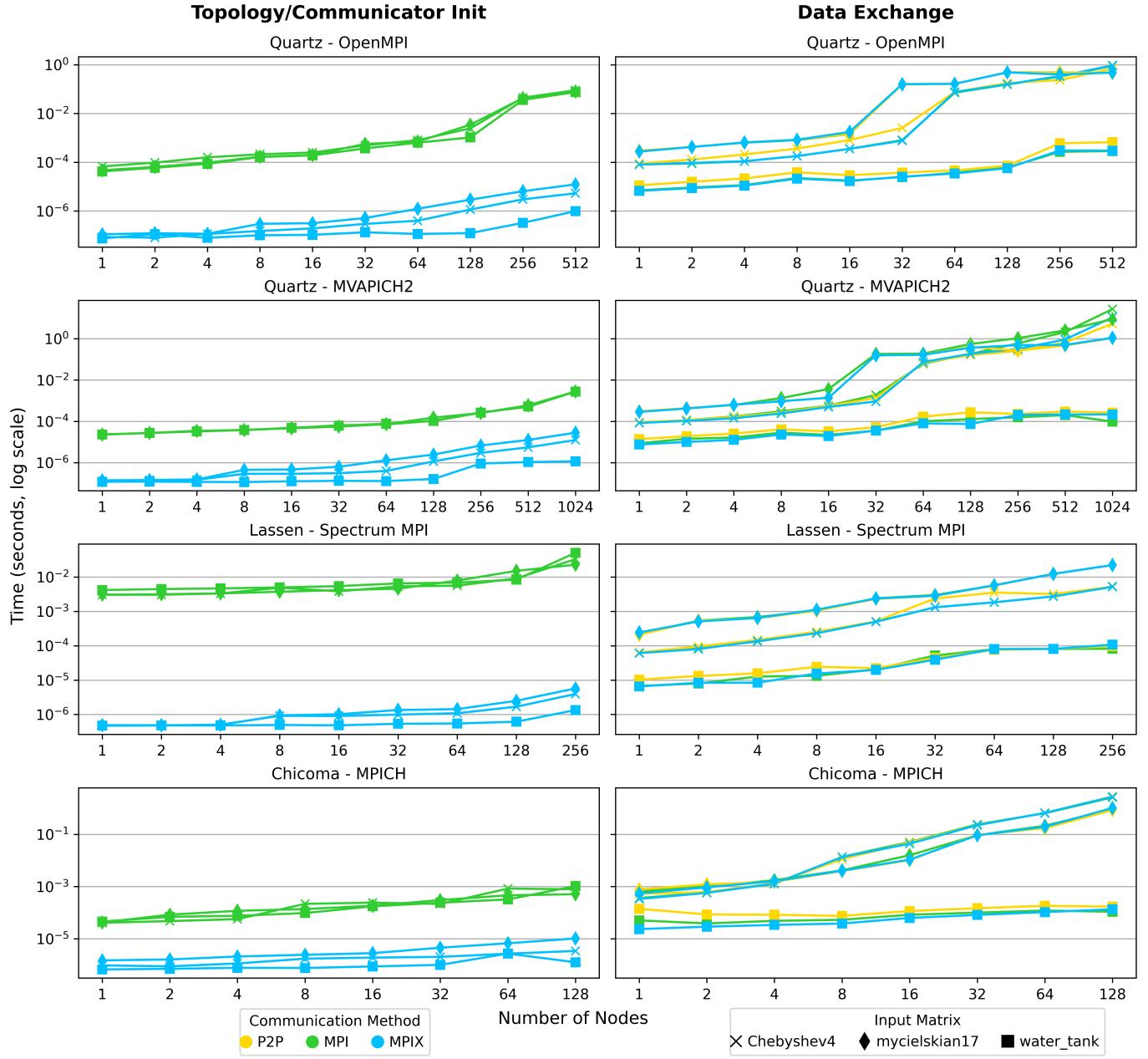


Fig. 1. The cost of strongly scaling a sparse matrix-vector multiply to over 10,000 processes, split into topology/communicator initialization costs (left) and data exchange costs (right). Runs on Quartz has 36 processes per node; runs on Lassen had 40 processes per node; and runs on Chicoma had 128 processes per node. For all graphs, colors indicate communication approach; symbols on a line distinguish different input matrices. Error bars are not shown for clarity of the figure, but they were not significant.

very large performance gains in comparison with our new approach to be able to successfully amortize away their significantly higher creation and destruction overheads.

C. Application Experiments

Following the benchmark study, we integrated our new communication topology primitives into the *hypre* and xRAGE applications to quantify the impact of communication topology objects on the performance of real applications.

1) *hypre Results*: *hypre* BoomerAMG is an algebraic multi-grid (AMG) solver that is widely used to solve linear systems arising from partial differential equations [6]. A *hypre* solve consists of two phases: a setup phase, in which a hierarchy of successively coarser matrices are formed, and a solve phase, which iteratively reduces error across this hierarchy through relaxation and coarse grid correction. As a result, *hypre* executes varying communication patterns across the different levels of the matrix hierarchy. At scale, *hypre* is bottlenecked by the irregular communication required by sparse matrix

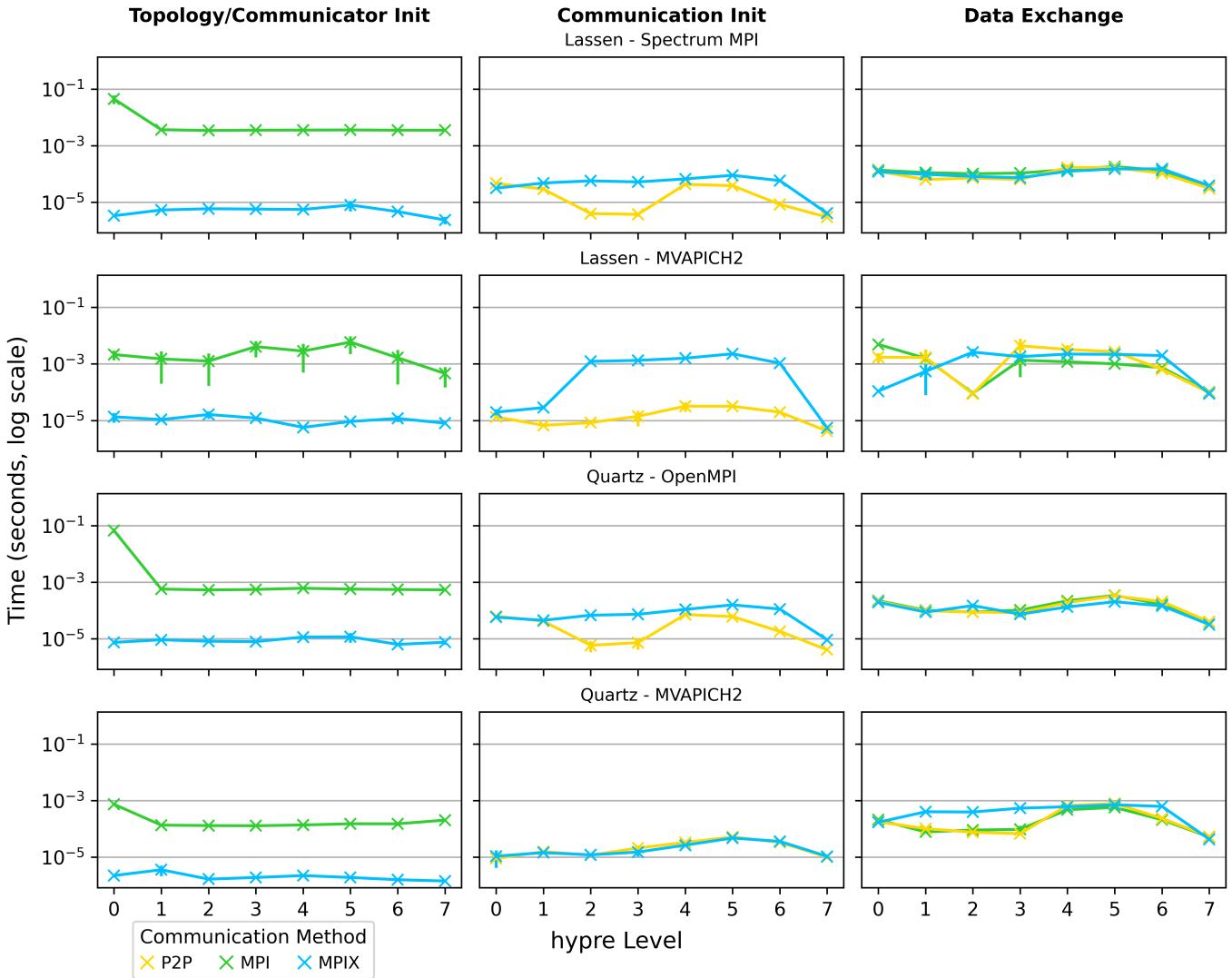


Fig. 2. Comparison of topology communication approaches across *hypre*'s AMG matrix hierarchy on Lassen and Quartz (64 nodes, 2048 processes for all runs). Error bars represent the standard error of the mean. The left column of graphs represent the per-topology creation creation costs; the middle column represents the per communication initialization costs; and the right column represents the per iteration communication costs. For all graphs, color indicates communication approach.

operations in both phases, particularly on coarse levels near the middle of the hierarchy with large communication constraints.

In order to evaluate the performance of the new topology object in *hypre*, we conduct two separate studies. First, we measure the communication initialization and data exchange costs across each level of the AMG hierarchy for three approaches: traditional non-persistent MPI neighbor collectives, optimized persistent point-to-point sparse exchanges standard in recent releases of *hypre*, and the new MPIX topology objects and associated persistent neighbor collectives. Second, we conducted a strong scaling study comparing two versions of *hypre*: one using MPIX topology objects and one using hand-crafted persistent point-to-point operations.

a) *AMG Per-level Primitive Performance*: Figure 2 shows the average results of the communication cost com-

parison across the AMG matrix hierarchy, taken from three separate test runs². We conducted the test on 64 nodes and 2048 processes, solving a 512^3 27-point Laplacian with AMG best practices, including aggressive coarsening. MPI timing functions around key parts were used to capture times.

For communication within *hypre*, the cost of topology communicator creation in MPI is significant, exceeding the cost of the subsequent communication exchange for some MPI implementations. In the most extreme case of Spectrum MPI, the cost of calling `MPI_Dist_graph_create_adjacent` exceeds the cost of the exchange by more than an order of magnitude. Notably, any optimizations within the system MPI

²A handful of data points for the Lassen MVAPICH2 run experienced variations with a standard deviation larger than their average. Standard error of the mean was chosen here to prevent the error bars from going negative.

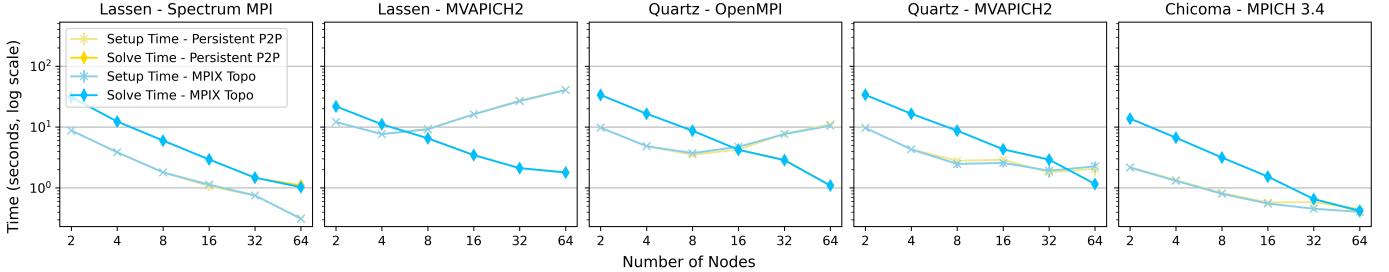


Fig. 3. Strong scaling comparison of *hypre* setup and solve times across a variety of systems as reported by *hypre* itself. Runs on Lassen and Quartz utilized 32 MPI ranks per node; runs on Chicoma utilized 128 MPI ranks per node. Here, the comparison is between using hand-crafted point-to-point operations (yellow) and using our topology construct within *hypre* (blue). We note that the lines are on top of one another for almost all runs.

implementation fail to make up for the cost of creating the new communicator. In contrast, the cost of creating the new topology primitive is two to three orders of magnitude faster than creating a new communicator.

b) *Strong scaling study*: Figure 3 presents a strong scaling study of *hypre* with the new topology primitives compared against point-to-point communication currently utilized throughout BoomerAMG. For the study, we scaled from 2 to 64 nodes using the same problem parameters described in the previous test. For each scale, the figure shows the average of three test runs, using times reported by *hypre*. Regardless of system or MPI implementation tested, the new neighborhood collective implementation achieved almost identical performance to point-to-point, avoiding any additional cost from communicator creation. A significant benefit, however, is that the neighborhood collective version of *hypre* can now utilize any implementation optimizations behind the neighborhood collectives by simply updating the MPI implementation.

municating using that abstraction, respectively. The *Topology Initialization* column includes the time xRAGE spends finalizing neighbor discovery, as that is performed in lock-step with the creation of the topology; the code for neighbor discovery remains the same for all runs. All times are in seconds.

Adding traditional MPI topology objects to xRAGE increases the overall run time by about 1.5 seconds (7% slower). We hypothesize that this performance cost stems from the need to create two MPI communicators across thousands of MPI ranks, which adds two seconds (111% slower). Note, however, that native MPI neighbor collectives *reduces* the in topology communication times by about half a second (20% faster).

Since creating MPIX_Topos requires no external communication, using this topology creation routine in xRAGE did not significantly increase the time spent in topology initialization. MPI Advance's unoptimized neighbor collectives performed slightly worse than the hand-tuned interleaving reduction operations in xRAGE in topology communication (6% slower). Despite this, the runtime of xRAGE using the new topology objects is competitive with the original version using hand-optimized sparse data exchanges.

VI. CONCLUSIONS AND FUTURE WORK

This paper describes research on the design, implementation, and evaluation of a new communication topology approach for MPI, focused on improving the performance of codes with sparse data exchanges with changing topologies. Based on the analysis of the needs of such codes and the abstractions available in the latest version of the MPI standard, we propose a new first-class communication topology object that can be created, modified, and destroyed without inter-process communication. This approach avoids the high cost of creating process topologies, while also leveraging persistent collectives to support future optimization of neighbor collectives using this abstraction. We then present an external MPI library that provides a full implementation of this topology construct and neighbor collectives that can leverage it.

The evaluation of the proposed approach uses both benchmark and production applications across a range of different systems, network interconnects, and underlying MPI implementations. Benchmark measurements use sparse matrix-

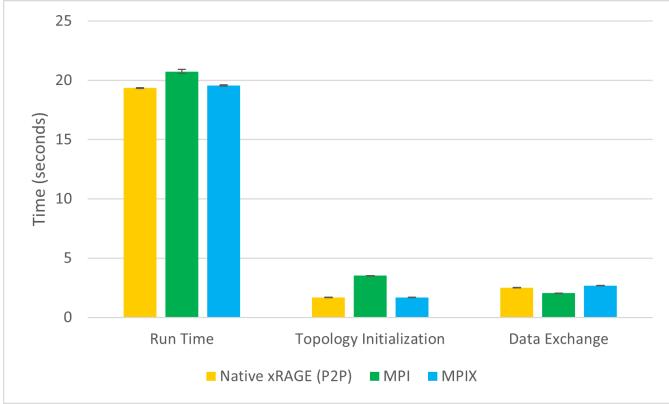


Fig. 4. Results from running xRAGE on Rocinante (32 nodes, 110 processes per node), with one standard deviation as the error bars. Colors indicate communication approach.

2) *xRAGE Results*: Figure 4 shows the results of all three implementations running an asteroid impact code on 32 nodes of LANL's Rocinante system with 110 processes per node, with times extracted using the Caliper framework [18]. The *Topology Initialization* and *Data Exchange* categories represent the percentage of total time xRAGE spent creating its custom topology abstraction and time xRAGE spent com-

vector multiply operations to measure the performance of different operations required to specify and communicate sparse data exchanges, including the standard communicator-based MPI approach, hand-crafted point-to-point sparse exchanges, and first-class communication topologies implemented as an MPIX extension library. Application evaluations use the *hypre* and xRAGE codes with production input decks.

Results from the evaluation of the proposed abstraction demonstrate (1) it can be effectively integrated into production codes with minimal changes, (2) it significantly reduces the time taken to create or change communication topologies compared to native MPI, and (3) it results in performance and total application runtimes competitive to hand-built point-to-point sparse data exchanges without major optimization effort.

The end goal of our research is to create a robust set of MPI primitives and associated optimized implementations that make it easier to develop robust, performant codes that feature irregular, dynamic communication patterns. Based on the results described in this paper, a key future direction for work in this area are MPI persistent neighbor collectives implementations that fully leverage process topology, network topology, and communication topology information to schedule and optimize communication performance. In addition, researching new MPI primitives for optimizing topology discovery, a performance-sensitive portion of many applications with irregular, dynamic communication, is also promising.

ACKNOWLEDGMENT

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).

This work was supported by the U.S. Department of Energy through the Los Alamos National Laboratory (LANL). LANL is operated by Triad National Security, LLC, for the National Nuclear Security Administration of U.S. Department of Energy (Contract No. 89233218CNA00001; LA-UR-24-23733).

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] T. Hoefer and T. Schneider, "Optimization principles for collective neighborhood communications," in *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012, pp. 1–10.
- [2] S. H. Mirsadeghi, J. L. Traff, P. Balaji, and A. Afsahi, "Exploiting common neighborhoods to optimize mpi neighborhood collectives," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, 2017, pp. 348–357.
- [3] G. Collom, R. P. Li, and A. Bienz, "Optimizing irregular communication with neighborhood collectives and locality-aware parallelism," in *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*, ser. SC-W '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 427–437. [Online]. Available: <https://doi.org/10.1145/3624062.3624111>
- [4] J. L. Traff and S. Hunold, "Cartesian collective communication," in *Proceedings of the 48th International Conference on Parallel Processing*, ser. ICPP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3337821.3337848>
- [5] A. Bienz, D. Schafer, and A. Skjellum, "MPI Advance : Open-source message passing optimizations," in *EuroMPI'23: 30th European MPI Users' Group Meeting*, 2023. [Online]. Available: https://eurompi23.github.io/assets/papers/EuroMPI23_paper_33.pdf
- [6] R. D. Faloutsos and U. M. Yang, "hypre: A library of high performance preconditioners," in *Computational Science — ICCS 2002*, P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 632–641.
- [7] T. Trilinos Project Team, *The Trilinos Project Website*, 2024. [Online]. Available: <https://trilinos.github.io>
- [8] J. Zhang, J. Brown, S. Balay, J. Faibussowitsch, M. Knepley, O. Marin, R. T. Mills, T. Munson, B. F. Smith, and S. Zampini, "The PetScSF scalable communication layer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 842–853, 2022.
- [9] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, D. Ranta, and R. Stefan, "The RAGE radiation-hydrodynamic code," *Computational Science & Discovery*, vol. 1, no. 1, p. 015005, nov 2008. [Online]. Available: <https://dx.doi.org/10.1088/1749-4699/1/1/015005>
- [10] P. Grete, J. C. Dolence, J. M. Miller, J. Brown, B. Ryan, A. Gaspar, F. Glines, S. Swaminarayan, J. Lippuner, C. J. Solomon, G. Shipman, C. Junghans, D. Holladay, J. M. Stone, and L. F. Roberts, "Parthenon—a performance portable block-structured adaptive mesh refinement framework," *The International Journal of High Performance Computing Applications*, vol. 37, no. 5, p. 465–486, Dec. 2022. [Online]. Available: <http://dx.doi.org/10.1177/10943420221143775>
- [11] T. Hoefer and J. L. Traff, "Sparse collective operations for MPI," in *2009 IEEE International Symposium on Parallel & Distributed Processing*. IEEE, 2009, pp. 1–8.
- [12] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard Version 4.1*, Nov. 2023. [Online]. Available: <https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf>
- [13] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.
- [14] W. Gropp and E. Lusk, "User's guide for mpich, a portable implementation of MPI," 1996.
- [15] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003, pp. 55–55.
- [16] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.
- [17] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, dec 2011. [Online]. Available: <https://doi.org/10.1145/2049662.2049663>
- [18] Boehme, David and Gamblin, Todd and Beckingsale, David and Bremer, Peer-Timo and Gimenez, Alfredo and LeGendre, Matthew and Pearce, Olga and Schulz, Martin, "Caliper: Performance Introspection for HPC Software Stacks," in *SC16: International Conference for High Performance Computing, Networking, Storage and Analysis*. Salt Lake City, UT, USA: IEEE, Nov 2016, p. 550–560. [Online]. Available: <http://ieeexplore.ieee.org/document/7877125/>