Design of a Portable Implementation of Partitioned Point-to-Point Communication Primitives

Andrew Worley
Tennessee Tech University
Cookeville, Tennessee, USA
apworley42@tntech.edu

Prema Soundararajan
University of Alabama Birmingham
Birmingham, Alabama, USA
prema@uab.edu

Derek Schafer
Univ. of Tennessee at Chattanooga
Chattanooga, Tennessee, USA
derek-schafer@utc.edu

Ryan E. Grant
Matthew G.F. Dosanjh*
Sandia National Laboratories
Albuquerque, New Mexico, USA
regrant@sandia.gov
mdosanj@sandia.gov

Purushotham V. Bangalore
University of Alabama Birmingham
Birmingham, Alabama, USA
puri@uab.edu

Anthony Skjellum
Univ. of Tennessee at Chattanooga
Chattanooga, Tennessee, USA
tony-skjellum@utc.edu

Sheikh Ghafoor
Tennessee Tech University
Cookeville, Tennessee, USA
sghafoor@tntech.edu

ABSTRACT

The Message Passing Interface (MPI) has been the dominant message passing solution for scientific computing for decades. MPI point-to-point communications are highly efficient mechanisms for process-to-process communication. However, MPI performance is slowed by concurrency protections in the MPI library when processes utilize multiple threads. MPI's current thread-level interface imposes these overheads throughout the library when thread safety is needed. While much work has been done to reduce multithreading overheads in MPI, a solution is needed that reduces the number of messages exchanged in a threaded environment.

Partitioned communication is included in the MPI 4.0 standard as an alternative that addresses the challenges of multithreaded communication in MPI today. Partitioned communication reduces overall message volume by creating a buffer-sharing mechanism between threads such that they can indicate when portions of a communication buffer are available to be sent. Separation of the control and data planes in MPI is enabled by allowing persistent initialization and single occurrence message buffer matching from the indication that the data is ready to be sent. This enables the usage of underlying hardware primitives like triggered operations, where commands (destination, size, etc.) can be set up prior to data buffer readiness with readiness triggered by a simple doorbell/counter later. This approach is useful for future development of MPI operations in environments where traditional networking commands can have performance challenges, like accelerators (GPUs, FPGAs).

In this paper, we detail the design and implementation of a layered library (built on top of MPI-3.1) and an integrated Open MPI solution that supports the new, MPI-4.0 partitioned communication feature set. The library will enable applications to use currently released MPI implementations and older legacy libraries to provide partitioned communication support while also enabling further exploration of this new communication model in new applications and use cases. We will compare the designs of the library and native Open MPI support, provide performance results and comparisons between the two approaches, and lessons learned from the implementation of partitioned communication in both library and native forms.

We find that the native implementation and library have similar performance with a percentage difference under 0.94% in microbenchmarks and performance within 5% for a partitioned communication enabled proxy application.

ACM Reference Format:
Andrew Worley, Prema Soundararajan, Derek Schafer, Ryan E. Grant, Matthew G.F. Dosanjh, Purushotham V. Bangalore, Anthony Skjellum, and Sheikh Ghafoor. 2021. Design of a Portable Implementation of Partitioned Point-to-Point Communication Primitives. In *50th International Conference on Parallel Processing Workshop (ICPP Workshops '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3458744.3474046

1 INTRODUCTION

The Message Passing Interface (MPI) is the de facto standard for communicating between a large number of peer processes. The first MPI standard was released in 1994 [10]. Thread support was subsequently added in MPI-2.0 [11]. The inclusion of thread support and an

*Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S.Department of Energy Office of Science and the National Nuclear Security Administration.

ICPP Workshops '21, August 9–12, 2021, Lemont, IL, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8441-4/21/08…$15.00
https://doi.org/10.1145/3458744.3474046

increasing availability of multicore processors added both the possibility and necessity of hybrid programming with MPI (often denoted by MPI+X, where X=Pthreads, OpenMP, GPU acceleration kernels, etc). Supporting threaded programs well is complex and represents a major investment of effort for an MPI implementation. In light of this fact, the standard allows for different levels of thread reentrancy support, and does not require all implementations to provide all options. The thread support level indicates how much concurrent interaction is allowed between application threads and the MPI library. All thread support levels except the highest, `MPI_THREAD_MULTIPLE`, limit the interaction between threads and the MPI API, and thus each form a unique programming model. There are always locking and related overheads associated with supporting reentrancy in MPI.

There have been attempts over the years to streamline the hybrid programming model by integrating features into implementations. The endpoints concept was one such attempt [1, 17], which expanded the inter-process message addressing to include 'thread' as a unique label, thus allow directed communication between threads controlled by different processes. Unfortunately, the proposed model has foundered in the MPI Forum's standardization process. While endpoints supported thread ranks (in groups and communicators) and thus addressability in point-to-point and collective communication, the proposal was not adopted because improvements could be adopted in the MPI implementations transparently [19] and because it was judged that the likely impacts on transport state and receive-queue lengths would be unacceptable with large-count multicore nodes in current and forthcoming scalable systems [2, 14].

The finepoints model [5, 7], introduced by Grant et al., is another such model, now called partitioned point-to-point communication. This model was designed to address the queuing concerns of endpoints. Further, partitioned communication was designed such that it was possible to require only a portion of the library to function with full thread support, easing some of the optimization costs [7].

Our motivation for the present work was to prototype and make available a robust standalone MPI extension library that enables this partitioned communication, and that works with any compliant MPI-3.1 implementation (that is, via a layered implementation). Since partitioned communication is new in MPI 4.0 [3, 12], the public implementations currently available are new as well, including the Open MPI one developed for this paper. We implemented partitioned communication in Open MPI, based in part on our layered library approach, which uses low-level internal calls to enable partitioned communication. It is available in the development branch of Open MPI and will be included in an upcoming official release. Also, the MPICH implementation of MPI now supports partitioned communication in a pre-release form [13].

Although we mention drawbacks of a layered library below, there are also certain advantages to a library implementation that is independent of the underlying MPI implementation. First, the library approach means that partitioned communication can be provided in most or all MPI implementations immediately. In addition, the library can be used with older MPI versions, allowing for backwards compatibility for codes adapted to use partitioned communication. Finally, availability of a full, portable extension library will enable application adoption while providing a reference implementation for MPI-library-specific realizations of this functionality that are likely to be optimized over the next few months as MPI-4.0 becomes fully standardized and supported [12]. Lastly, we note that the layered library provides a vehicle for exploring extensions such as partitioned collective communication operations, due to be proposed in MPI-5.

This paper makes the following contributions:

- The first open-source portable library for partitioned communication.
- An MPI implementation agnostic platform for developing future partitioned communication concepts and interfaces.
- One of the first native MPI library implementations of partitioned communication.
- The first performance comparison between a partitioned communication library and a native MPI implementation.

Given these contributions, the audience for this paper are early adopters of partitioned communication and MPI library writers who seek to utilize a correct, exemplary baseline code for their efficient implementations, as well as researchers (such as the authors of this paper) who seek to extend partitioned communication in the near future with proof-of-concept prototyping.

The remainder of the paper is organized as follows: Section 2 offers motivations and briefly describes the new partitioned communication APIs introduced in MPI 4.0. Section 3 presents a high level view of how the partition communication functions are implemented in our library. This section also includes some commentary on the variations and optimizations for specific scenarios, while Section 3.4 describes the limitations and restrictions of the library. Section 4 shows the results of the initial benchmark testing. Section 5 discusses future work, including a discussion on GPU support. Lastly, Section 6 recapitulates our findings.

2 PARTITIONED COMMUNICATION

Partitioned communication provides a thread-friendly interface to MPI without introducing crosscutting changes to groups, process numbering in groups and communications (loosely, ranks), and mux/demux of messages. Rather, it enables applications to associate multiple threads with buffer partitions on the and/or receiver. Threads remain anonymous while buffer partitions become named sub-entities addressable locally on the sender/receiver. The message remains the end-to-end abstraction as in normal point-to-point communication.

In particular, partitioned point-to-point operations provide a thread-interface for message passing that supports pipelined and parallel message buffer filling and emptying, with the potential for overlapping buffer completion with transfer. This pipelining can have significant benefits for hybrid programming, such as MPI+OpenMP with fork-join assembly of messages in non-overlapping partitions (send-side overlap of communication and computation) and/or partitioned completion of messages for overlapping receipt and work as data is received (receive-side overlap of communication and computation).

2.1 Partitioned Communication API

The status and progress of persistent communication operations in the MPI 3.1 standard are controlled through functions called on semi-permanent (reusable) request objects. Partitioned communications use a similar control methodology. The functions introduced as part of partitioned point-to-point communication API with function

Table 1: MPI 4.0 and 4.1 Partitioned Point-to-Point Communication Application Programmer Interface (API) [12]

Approved MPI 4.0 Functions	**C Language Binding**
`MPI_Psend_init`	`void *buf, int partitions, MPI_Count count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Info info, MPI_Request *request`
`MPI_Precv_init`	`void *buf, int partitions, MPI_Count count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Info info, MPI_Request *request`
`MPI_Pready`	`int partition, MPI_Request *request`
`MPI_Pready_range`	`int partition_low, int partition_high, MPI_Request *request`
`MPI_Pready_list`	`int length, int array_of_partitions[], MPI_Request *request`
`MPI_Parrived`	`MPI_Request *request, int partition, int *flag`
Proposed MPI 4.1 Functions	
`MPI_Pbuf_prepare`	`MPI_Request request`
`MPI_Pbuf_prepareall`	`int count, MPI_Request requests[]`

prototypes can be seen in Table 1, and are described briefly below. In addition to the functions listed in the 4.0 API, there are two functions that have not yet been implemented that are proposed for inclusion in MPI 4.1, as well as minor behavioral changes for when MPI_Start is called on a partitioned request.

MPI_Psend_init tells the library to start preparing a partitioned send operation. This function sets up the required plan for transfers and synchronization that will be needed for executing the operation later and is nonblocking (incomplete).

MPI_Precv_init operates analogously to MPI_Psend_init and shares the same nonblocking requirement and needs to synchronize, in general with its corresponding MPI_Psend_init. This function prepares the receive-side to support a partitioned receive.

MPI_Pready is a local, nonblocking operation that tells the library that the partition with the supplied ID is ready to be sent. Even though the request must be started for a transfer to occur, individual partitions will not be sent until they are marked as ready with a MPI_Pready or its variants. MPI_Pready_range enables the marking of a number of partitions in a request with an id between partition_low and partition_high inside a single function call while MPI_Pready_list marks those partitions with ids that are enumerated in the supplied array argument.

MPI_Parrived tests for the successful arrival of a particular partition on the receive side and if it is available for use. This function is nonblocking can return true before the entire partitioned send request is complete as long as the tested partition is ready at the receiver.

2.2 Partitioned Communication Operations

The syntax and semantics for partitioned communications is based on persistent communication functions present in the MPI 3.1 standard, with extensions. Once the request object is initialized on each side (in partitioned point-to-point, there is only one matching instance between sender and receiver), the library waits for various control functions shown in the previous section to be to be called and reacts accordingly. The functions and progress flow for a simple application using partitioned communication is shown in Figure 1.

The communication begins with the initialization of the request objects. This is accomplished by calling the init function on each side of the request. Similar to other persistent MPI communication functions, the MPI_Psend_init and MPI_Precv_init functions take a pointer to a buffer, a source or destination, a communicator, a tag,

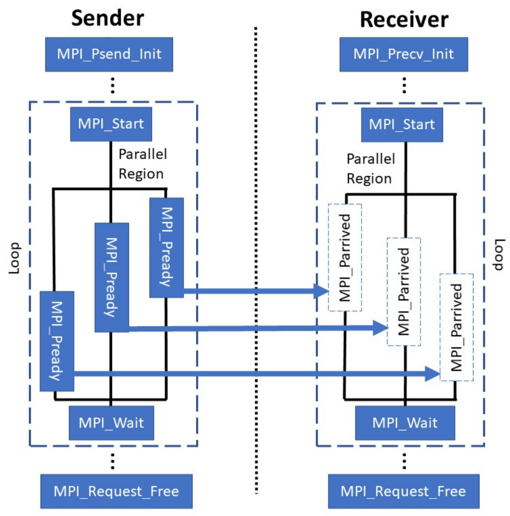


Figure 1: Functional, user-level view of partitioned communication according to the MPI 4.0 standard

and a MPI_Request handle. The application data buffer is logically divided into a number of equal-sized partitions given by the user. While the partitions on each side do not have to be the same, the total number of items on the receive size (type signature) must be greater than or equal to the number of items sent. Additionally, the partitions on each side do not prescribe how many messages are sent over the network. The supplied number of partitions defines how many external-facing partitions the buffer is split into on each side of the communication. The currently approved scenario using a set number of equal sized partitions causes behavior similar to existing MPI collective operations in that elements in the buffer outside of the range of count $\times$ partition are not sent. For example, if a buffer is fifty integers long and the send request is set up to send four partitions with ten integers, then the last ten integers will not be sent.

After the init functions complete, the resulting object is then used to monitor and progress the request throughout its lifespan. Once the request has been created, MPI_Start activates the object; however, unlike a persistent request, no data is sent immediately. The MPI_Start function in this case only switches the request state to active. In order for the request to begin sending data, one or more partitions must be marked as ready to send. This is accomplished by calling MPI_Pready or a variant on that partition. Eventually,

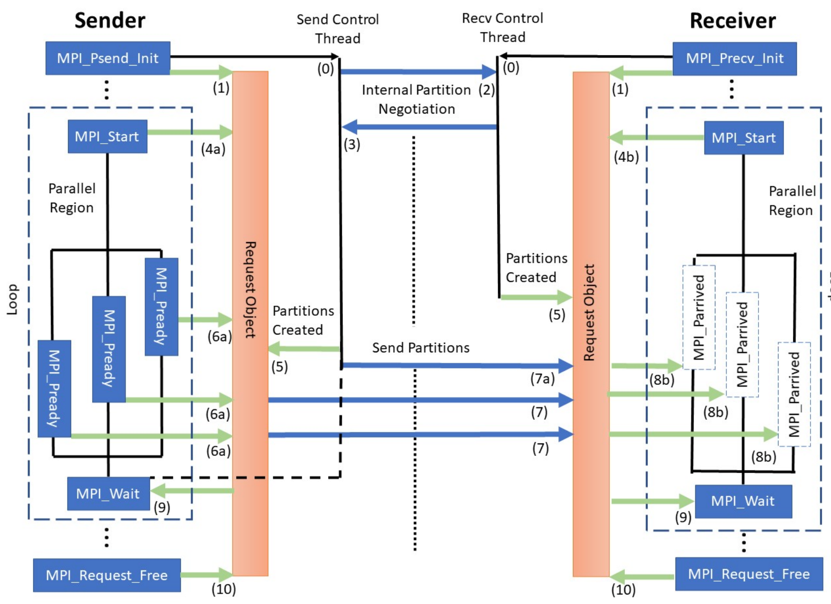


Figure 2: Architectural overview of our portable library - Green arrows represent control variable signals and blue arrows represent inter-process communication.

all partitions must be marked as ready for the send operation to complete. A MPI_Wait is also required to follow the marking of all partitions as ready, to ensure that the buffer is emptied, and that another partition send can be initiated.

The receive side has an optional function to inquire about the status of a particular partition, MPI_Parrived. The function returns whether or not the provided partition has arrived. However, a MPI_Wait or MPI_Test call must still be used to complete and finalize the receive-side request. The optional MPI_Parrived call is not useful if there is only one receive-side partition, because the user must still call MPI_Wait or MPI_Test.

An inactive or completed request may be restarted multiple times; however, the size and buffer are fixed at the creation of the request. MPI_Request_free must be called to free the resources associated with the request on each side (and only when they are inactive, a further difference from MPI persistent send and receive operations).

All the new functions used in the partitioned API are considered nonblocking; however in order to ensure efficient operation between both sides of the communication some synchronization method is beneficial. The MPI_Pbuf_prepare and MPI_Pbuf_prepareall functions are being developed by some of the authors of this paper and others to fulfill this role [4]. For more details about the integration of MPI_Pbuf_prepare, see Section 5.

3 ARCHITECTURAL OVERVIEW

There were two key goals for the design of the library: fully implement the approved API while achieving maximize portability. These goals led to the production of a layered form of the library that uses the standard C language binding of an MPI implementation. This allows the library to run on top of any MPI-3.1 compliant implementation. While a layered library supports portability, it limits the use of opaque objects, in particular: MPI_Request, MPI_Status, MPI_Datatype, and MPI_Comm. These limitations constrained the design and resulted in some key observations. We considered the various options for implementing the external library using the existing MPI point-to-point and one-sided APIs. Since the init calls are local, we cannot use blocking or one-sided APIs to implement them, leaving us with nonblocking and persistent point-to-point APIs. We decided to use the persistent point-to-point APIs in this library since they have a reasonable semantic match.

The operation of the library follows the basic protocol shown in Figure 2. The partition communication operation begins with the initialization of the MPI_Request object (arrow 1). After the object is created, the library then needs to agree on the number of internal partitions. Since the initialization functions are nonblocking, our library spawns a short-lived control thread on each side (arrow 0) to exchange information about the number of partitions and datatypes (arrows 2 and 3). One control thread is spawned for each init function call and each thread performs a nonblocking send and a receive operation and waits for the operation to complete. Having a separate control thread allows the initialization functions on both the send-side and receive-side to return immediately without waiting for this information exchange. Once the control thread completes, the corresponding send and receive operations, the agreed-on number of internal requests are created.

User-level requests are inactive until MPI_Start is called on the given request. On the send side, the request is marked active (arrow 4a) and execution continues. On the receive side, if the control thread has completed, the library activates the partitions of the request (arrow 4b). Otherwise, it sets a flag to activate the partitions after the control thread finishes the setup (arrow 5). The next stage is sending the partitions as they are marked ready (arrow 6a). If the send control thread has finished synchronization (arrows 3 and then 5), the underlying MPI implementation starts progressing the associated partitions (arrow 7). Otherwise, the `MPI_Pready` call marks the partition(s) as ready to be sent. When the send control thread finishes synchronizing, it will send any marked partitions (arrow 7a). On the receive side, the status of each internal request can be tested by `MPI_Parrived` (arrow 8b). The program can wait for the communication to finish by calling `MPI_Wait` on both sides. After all the internal partitions are successfully completed, the request marks that `MPI_Wait` can return.

The objects and resources can be released by calling `MPI_Request-_free`. This deallocates the partitioned request object and the persistent internal requests, as well as terminating any existing control threads (arrow 10).

3.1 `MPIX_Request` Object

In order to fulfill the needs of the API, the library has to track the transmission status of each partition. The traditional `MPI_Request` object lacks the capacity to keep track of multiple partially completed requests. Since the library is designed to be portable, we had to create a framework that would allow us to observe and control the transmission of each partition. For this purpose, we created a new `MPIX_Request` object. The basic layout of the object is shown in Listing 1.

Listing 1: The `MPIX_Request` object.

```
typedef struct _mpix_request
{
  int state;
  int size;
  int side;
  int sendparts;
  int recvparts;
  int readycount;
  MPI_Request *request;
  ... // Other thread information
} MPIX_Request;
```

The `MPIX_Request` object contains two major sections: status metadata, and an internal array of normal MPI requests. The metadata includes the number and size of the partitions in the request as well as status information on the request progress. Additional information is used to map between the abstraction layers inside the request. The internal array of requests enable us to leverage the optimizations of the host implementation, and send each partition individually and track each partition's completion status.

As the `MPIX_Request` objects are replacing the `MPI_Request` objects, the various MPI functions that interact these objects had to be adapted to accept the `MPIX_Request` object. These re-implemented functions are listed in the first column of Table 1. The augmented functions share the same prototype and behavior, except that all instances of a `MPI_Request` instead require a `MPIX_Request`. `MPIX_-Request` objects are functionally the same as `MPI_Request` objects, but are not interchangeable (a side-effect of a the layered architecture).

Because of the layered structure of the library, we have limited access to the `MPI_STATUS` objects that are normally returned with these functions, and internally all status objects have been replaced with `MPI_STATUS_IGNORE`. This is further discussed in Section 3.4.

Table 2: List of Provided Functions

Partition Functions	Augmented functions
`MPI_Psend_init`	`MPI_Start`
`MPI_Precv_init`	`MPI_Startall`
`MPI_Pready`	`MPI_Wait`
`MPI_Pready_range`	`MPI_Waitall`
`MPI_Pready_list`	`MPI_Waitany`
`MPI_Parrived`	`MPI_Waitsome`
`MPIX_Pbuf_prepare`	`MPI_Test`
`MPIX_Pbuf_prepareallall`	`MPI_Testall`
	`MPI_Testany`
	`MPI_Testsome`
	`MPI_Request_free`

3.2 Internal Partitions

Since the library is designed on top of existing point-to-point communications and uses an internal array of requests, the library must comply with the semantics involved in those operations. This necessitates that both sides have the same number of internal requests, otherwise communication becomes unbalanced. A solution to this is to communicate the number of internal requests between the processes. While this can be avoided by having a preset number of partitions assigned by an `MPI_Info` key, such a choice limits the optimizations that can be done internally and places the optimization on the developer's shoulders (contrary to the spirit of the design goals of partitioned communication).

The initial communication is necessary and most implementations have several internal features that could be used to resolve the issue. Since the library is external, we do not have access to those features and are forced to use the user-level point-to-point communication functions. This forces the library to wait for the number of requests to be decided before allocating the array of internal requests. Without creating the internal requests, we cannot initiate any transfer. Hence, this causes a dependency on the completion of the internal partition negotiation. This conflicts with the approved API in the standard as both `init` functions are defined as nonblocking calls. Thus we needed a way to wait for the communication to complete without blocking the return of the initialization (and/or starting) call(s).

To resolve this issue, we spawn a separate control thread to perform the necessary communication and synchronization and to ensure nonblocking behavior for the initialization calls. The control thread waits on the completion of the send/receive operations (shown by arrows 2 and 3 in Figure 2), while the initialization functions can return immediately. The partition generation (arrow 5) must be completed before data transfer can occur. It is possible that the main thread will continue to the next MPI call and attempt to progress the operation before the control thread completes. The continued existence of the control thread signifies that the required setup operations are not complete. As such, it is unsafe for the other threads to progress the library execution. Instead, the main thread adds the task to the control threads task queue. Once the control thread has completed the initialization tasks, it continues to process tasks queued by other threads. If the control thread has completed the initialization tasks before a call to `MPI_Pready`, then the library calls the start function on the corresponding partition.

3.3 Optimization of Specific Cases

A key feature of the partitioned communication model is that it supports a different number of partitions on each side of the transfer. At each end of a send-receive pair, partitioning does not control the remote semantics, only the local transfer semantics. Some of these use cases are listed in Table 3. Here, the number of send-side and receive-side partitions are represented by M and N respectively. The fourth column shows the internal number of messages generated for the given case. Cases 1 and 2 avoid additional communication during initialization. Case 1 uses a fixed number of internal transfers while Case 2 assumes an equal number of partitions on both sides. Cases 3 and 4 require additional communication during initialization to negotiate the number of partitions. Cases 5 and 6 extend Cases 3 and 4 to include additional optimization by (dis)aggregating messages based on the number of partitions. Lastly, Case 7 exploits the underlying architecture to choose appropriate partition sizes and provide highly optimized communication.

Table 3: Design Cases for Partitioning: Here M (resp, N) represents the number of partitions of the sender (resp, receiver). The parameter k is a positive integer multiple and X is an implementation-specific partition size.

Case	**Partitions**		**No. of Transfers**
	Send	**Recv**	
1	M	N	1 (no init comm.)
2	M	M	M (no init comm.)
3	M	N ≠ M	M (init comm.)
4	M	N ≠ M	N (init comm.)
5	M=kN	N	kN (init comm., aggregation)
6	M	N=kM	kM (init comm., dis-aggregation)
7	M	N ≠ M	X (aggregation & optimization)

These multiple cases result in partitioned communication having a large state space that makes general optimization difficult; however, optimization for specific cases is fairly straightforward. The `MPI_Info` object supplied to the `init` functions is designed to inform the underlying implementation which case is applicable to the request. This in turn enables the use of code optimized for that case. Our library currently supports cases 1 through 3, while support for the remaining cases remains under development.

Case 1: In this case, both sides have the different number of partitions, and the internal number of partitions is set to a predetermined number. Since both sides of the request are assumed to receive the same `MPI_Info` object in the initialization function, this allows the initial communication to be safely ignored. However, in the future, there may be a small communication to confirm that both sides received the same `MPI_Info` key. This case allows the library to skip the thread spawning and negotiation process and immediately start making the internal requests and progressing through the model. This makes the communication represented by arrows 0, 2, 3, 5, and 7a (as well as the control thread) shown in Figure 2 unnecessary.

Case 2: This case has the same number of partitions on each side as the number of internal partitions. This is similar to the previous case in that the initial communication can be safely ignored, but also allows a one-to-one mapping of partitions for each layer of abstraction. This greatly simplifies the abstraction process and removes the need for any internal remapping.

Case 3: This case allows different partitions on each side; however, the internal number of partitions is determined by the number of external partitions on the send side. For example, a transfer might have six send partitions and three receive partitions. Six internal partitions will be created on each side. Further, this case allows arrow 3 from Figure 2 to be ignored since the sender already has all the information needed for initialization.

3.4 Limitations of the Layered Model

As this library current exists on top of existing MPI implementations, there are some limitations that are inherent to any layered libraries. Currently, because we cannot access the library's internals, we cannot properly return a fully formed `MPI_Status` object. Instead, our current implementation ignores any status objects passed to the function, and similarly uses `MPI_STATUS_IGNORE` as arguments to any function that requires a status object. In the future, we plan to provide our own status object to fulfill these needs.

The current implementation only supports built-in and contiguous datatypes. We intend to consider support for other user-defined datatypes as part of future effort. Also, our current implementation assumes that the datatypes on both the send-side and receive-side are identical (rather than type signatures). Support for distinct datatypes on sender and receiver is planned for future releases of the library. It is anticipated that applications that use partitioned communication will use multiple threads to achieve the best performance; that is, fill or empty buffer partitions concurrently. In addition, our current library has an extra thread to achieve some of the progress needed to keep the partitioned communications moving. Our library depends on the support of `MPI_THREAD_SERIALIZED` (or greater) by the underlying MPI implementation and any application using our library must call `MPI_Init_thread` (with the aforementioned thread support level or greater) instead of `MPI_Init`.

Also, since this library is not fully integrated with an underlying MPI code base, we are limited in how we can optimize. However, we are still aiming to achieve minimal extra overhead (if any) so as to show users of potential performance benefits they might get from using partitioned communications. Because the library uses persistent point-to-point APIs, applications should at least experience the performance benefits of persistent point-to-point APIs, if they are optimized, and of pipelining of partitions, provided the underlying MPI has strong progress and the partitions are sufficiently long.

The use of persistent point-to-point APIs to implement partitioned communication APIs also limits the current implementation from performing additional performance optimizations such as message aggregation as the number of send and receive calls are decided during the initialization calls.

In closing, it is worth noting that ROMIO [18] and LibNBC [9] both started as layered libraries, and have been integrated into multiple

MPI implementations with acceptable performance and long-term support. As such, the layered library approach can have extended impact, even though initial versions emphasize functionality over performance.

3.5 Integrated Model

To address some of the above-mentioned limitations, we also designed and implemented partitioned communication over persistent communication integrated into Open MPI. There are a few modifications that were made to make the library work with Open MPI. Many of these were primarily software engineering issues, such as creating C and FORTRAN bindings for the new functions, implementing a new partitioned module in Open MPI's Module Component Architecture (MCA), and adding partitioned communication support to the request structures. However, there are a few concerns that required a different behavior from our implementation.

First, many of augmented functions required different designs to support the calls. MPI_Start, MPI_Start_all, and MPI_Request_Free are directly invoked via function pointer in the request call. More significantly, MPI_Wait* and MPI_Test* rely on Open MPI's progress engine signaling requests as complete. This required us to implement and register a progress function that iterates over all active partitioned requests and checks for completeness.

Second, since Open MPI is used in a larger production environment, minimizing erroneous message matching collisions is an important consideration. To this end, we use two extra communicators that are duplicates of MPI_COMM_WORLD to isolate the communication. On the communicator specified in the MPI_Psend_init and MPI_Precv_init the origin sends a message internal partitions, ranks with respect to the dedicated partitioned communicators, and which tags to use on the dedicated communicators to avoid conflicts. Once the target has received this message, it sends a message back containing its rank on the dedicated communicators. This message is on a dedicated set-up communicator, using the unique tag specified in the previous message. Lastly, all the persistent communication requests are set up on the second dedicated communicator to ensure no conflicts arise with other MPI operations.

Finally, because we have a dedicated progress function, no additional threads are created during setup. Instead, as the progress function iterates through requests, it attempts to progress the set-up process if a request is not currently set up.

4 EVALUATION OF THE PORTABLE LIBRARY

The primary goal of our evaluation effort was to check the functionality, correctness, and portability of our library. We used two MPI library implementations: Open MPI 4.0.1 built with GCC-8.3.0-2.32 and Intel(R) MPI Library for Linux OS, Version 2017 Update 1 Build 20161016 to test our partitioned library implementation. For evaluating the performance of the library, we first focused on comparing the execution time of the partitioned APIs against existing blocking, nonblocking, and persistent point-to-point APIs. These tests were performed on a cluster in which each node of the cluster consists of two 2.4GHz Intel Xeon E5-2680 v4 (total of 28 cores per node) with 256GB RAM and EDR InfiniBand Mellanox ConnectX-5 interconnect. Since the library implements a form of point-to-point communications, only two nodes were needed for testing.

4.1 Comparison to Existing Communication Procedures

The first test is a comparison between the library and the existing blocking, nonblocking, and persistent point-to-point operations common to any MPI-3.1 implementation. The goal of this test is to establish a baseline for comparing the performance of the library with other point-to-point APIs. This test consisted of measuring the local shared memory (intranode) execution timing for different point-to-point APIs while increasing the message size. For this test, the library was run with a single partition, and each operation performed one hundred times. We performed this test with both the Open MPI and Intel MPI implementations and we notice similar patterns as shown in Figures 3 and 4. Because of the difference in the eager limits used by the two implementations, we observed that the four different APIs performance converge at different message sizes. In both cases, as expected, the execution times of the partitioned API are similar to the performance of the persistent API as the message size increases.

We tested the three cases (see Section 3.3) with the partitioned point-to-point APIs to compare the performance of the three cases and the results show that each case did not add any significant overhead compared to the persistent point-to-point operations using internode communication. We compared the performance of the partitioned point-to-point API when the number of send and receive partitions and datatypes are the same by increasing the number of partitions for different buffer sizes. We performed the initialization once and tested the starting and completion of the corresponding send and receives for 100 iterations. In the rest of this section, we present only the performance results with the Intel MPI library since the results with the Open MPI library were similar.

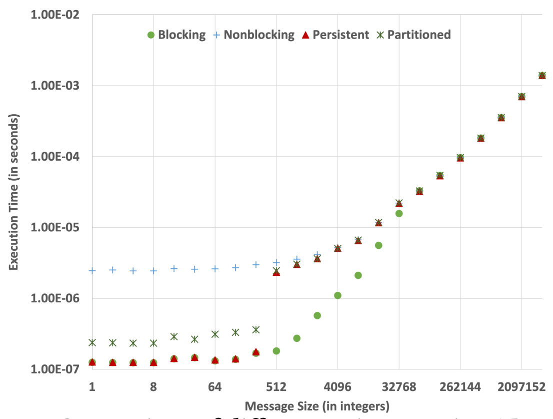


Figure 3: Comparison of different point-to-point APIs with OpenMPI

The execution time for four different buffer sizes (when the number of partitions are increased for Case 1) is presented in Figure 5a. As expected, the number of partitions does not have any impact on the execution time for a given buffer size since a fixed number of messages are sent and received irrespective of the number of partitions. We also observe that the execution time increases correspondingly when the buffer size is increased.

We repeated the above experiment by using Case 2 of the library implementation. The execution time for four different buffer sizes

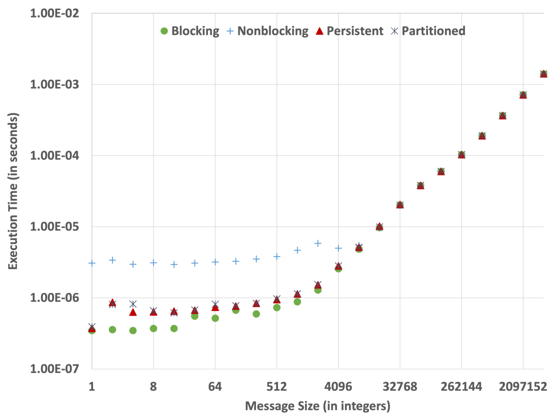


Figure 4: Comparison of different point-to-point APIs with Intel MPI

(when the number of partitions are increased for Case 2: the number of send and receive partitions are equal and the number of internal sends and receives are equal to the number of partitions) is shown in Figure 5a. As the number of partitions are increased, since the buffer size is fixed, the number of elements in each partition decreases. As a result, the library performs fewer send/receive calls on larger buffer sizes when the number of partitions are small.

Figure 5c shows the performance when the number of send-side partitions are mapped to be equal to the number of receive-side partitions. Since equal number of partitions were used for both send and receive partitions the performance is similar to that of Case 2.

Next, we tested the library by varying the message size of each partition as the number of partitions is increased. We tested for individual buffer sizes from 1KB to 4MB and varied the number of partitions from one to 28. In similarity to the earlier tests, we performed the initialization once and tested the starting and completion of the corresponding send and receives for 100 iterations. The execution time for different partition buffer sizes when the number of partitions are increased is shown in Figure 6.

4.2 Comparing Against The Open MPI Integrated Version

The Open MPI integrated version was built using its development repository. To compare our external library and internal implementation fairly, we used the fork of the Open MPI development repository with our internal implementation for all of the comparison tests. The integrated partitioned communication implementation for these experiments is publicly available in the development repository of Open MPI.

The system used for these test has 2 24-core Intel Skylake processors per node, connected via an first generation Intel Omnipath network. Open MPI was compiled with GCC 9.2.0. Each test was run for a total of 100 iterations and was averaged across three runs on different allocations.

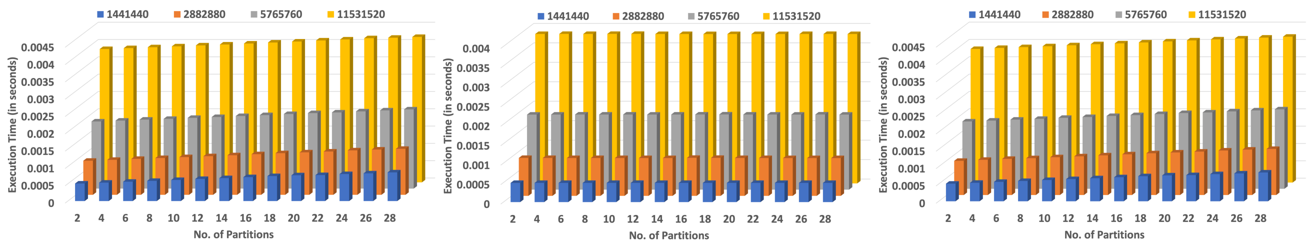
Figures 7 and 8 show the micro-benchmark results. Unsurprisingly, there is not much difference performance-wise between the two implementations, with all of the tests showing less than 1% difference in performance between implementations. The average difference is 0.32% with the external implementation showing better performance. While it difficult to make any specific conclusions with this data, the behavior may indicate that the indirect nature of completion in the integrated Open MPI implementation is causing a bit of extra overhead. In particular, as completion is done in the progress engine, there may be a delay between the completion of the underlying persistent requests and the completion of the partitioned request, while the `MPIX_Wait` of our external library can directly wait/test the underlying persistent requests. Additionally, since our external library has information on which call is being issued, it can take advantage of multi-completion calls, such as `MPI_Waitall`.

4.3 Proxy Application Evaluation

We modified MiniFE [8], a finite elements proxy application, to test the different implementations. In particular, we replaced the halo exchange communication with calls into our external library, the integrated Open MPI, and a multi-send version that uses the MPI's existing point-to-point interfaces. For these tests, we ran on the system described in Section 4.2. These runs were performed three times, using eight nodes, a single process per node, a 128^3 problem size per processes, and a variable number of threads which defined the number of partitions. It is important to note that this version of MiniFE still follows a bulk synchronous application pattern, where the communication happens in a different application phase from computation. Thus, while we test the performance of the communication subsystem, these tests don't allow for any early-bird communication (communication of partition(s) that commences before the final partition is completed on the send side).

Figure 9 shows the results for the conjugate gradient solver in MiniFE, and Figure 10 further breaks this out to just the maximum communication time (the maximum time a process spent doing point to point communication). These results show that both the external library and the integrated version perform similarly to Open MPI's highly optimized multi-threaded point-to-point implementation. With the exception of the two-threads case, both measurements show results that could be within expected noise, neither getting above 1.46% of solve time, with no distinguishing pattern to which method is more performant for communication. The notable exception here is two threads, where we measure significant communication overhead of 24% for using either partitioned implementation over multi-send. We observe extra overhead once an application start using threads, that gets quickly amortized out as the number of threads increase and this overhead seems to be greater for partitioned communication.

These results match expectations since both the integrated implementation and the external library are still using a form of MPI point-to-point to implement partitioned communication. These results highlight the fact that these initial implementations are viable development platforms until highly optimized implementations are released. It should also be noted that the miniFE benchmark does not allow for early-bird communication because of the minimal modifications to the bulk synchronous data exchange for these experiments, thereby mitigating the performance benefit that partitioned communication would otherwise have over a traditional point-to-point solution.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 5: Performance of partitioned point-to-point operations for different number of partitions and buffer sizes

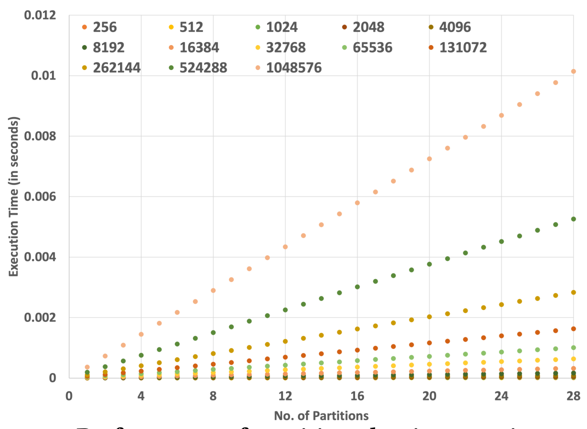


Figure 6: Performance of partitioned point-to-point operations when the size of the partition buffer is constant as the number of partitions are increased

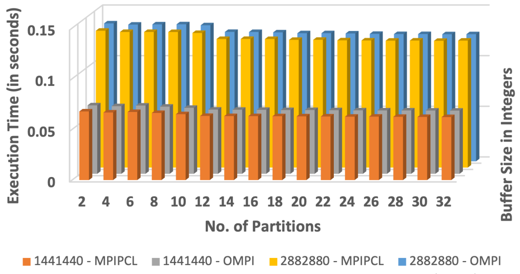


Figure 7: Comparison Between Our External Library and OMPI Integrated - Small Sizes

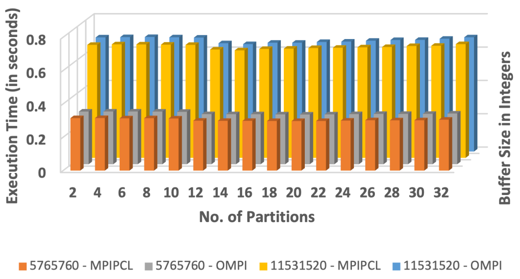


Figure 8: Comparison Between Our External Library and OMPI Integrated - Large Sizes

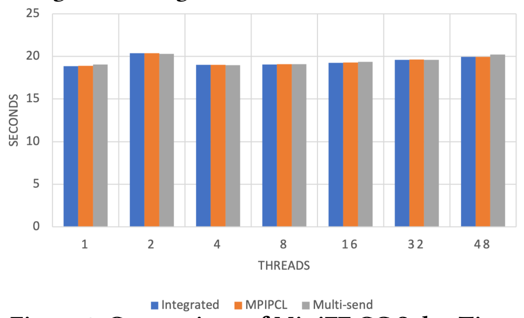


Figure 9: Comparison of MiniFE CG Solve Time

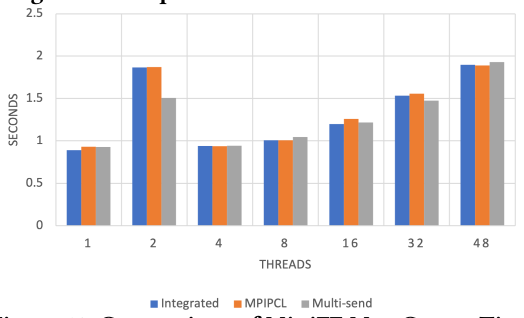


Figure 10: Comparison of MiniFE Max Comm Time

5 ENABLING NEXT GENERATION DEVELOPMENT

The library described here enables a convenient environment for the development and assessment of future extensions to MPI. We have identified three main categories of enhancements to future MPI standards that are promising avenues for future improvements and research: improving the layered library for greater performance at current functionality, expanding functionality to support more complex architectures, and speculative functionality not yet proposed or adopted in MPI-Next (versions beyond MPI-4.0). Note: both MPI-4.1 APIs shown in Table 1 and all cases enumerated in Table 3 are in scope for such enhancements.

Beyond these enhancements, generalizations of the partitioned point-to-point operations (e.g., late-binding, variable length partitions), combining partitioning with persistent collective operations, and demonstration of effective offload to GPUs and FPGAs are among the most promising future work we expect to undertake, together with other collaborators. Additionally, we expect to use the library to create a fully optimized, implementation-integrated version of partitioned communication in the near future (such as a more advanced version of the Open MPI work described in this paper).

5.1 Enhanced Structure and Assertion Handling

The optimizations for user assertions discussed at the end of Section 3 are currently implemented in separate versions of the library.

Moving forward, these various cases are going to be integrated and communicable at runtime to the library by the `MPI_Info` object provided to the `init` functions. From this, the library will call the relevant internal functions to execute the operation using code optimized for the particular assertion(s). Current planned assertions include side-driven (send or receive), and hard coded. The side-driven optimization allow for either side to drive the number of internal partitions based on the decision of the user. This simplifies the code so that the control thread is only needed on the non-driver side of the operation. Hard-coded mode allows both sides to complete the initialization process using a number of internal partitions chosen by the user. This allows the synchronization step to be omitted and removes the need to spawn a control thread.

5.2 Other Planned CPU Extensions

In future work, we will address limitations discussed above in Section 3.4. Many of these limitations could be addressed with a deeper integration with one or more MPI implementations (for instance, ExaMPI [16]). Such integration(s) are likely to reveal insights about partitioned communication because specific high-performance networks have different low-level capabilities and features. One such contribution found by the authors is the usefulness of adding an `MPI_Info` argument to the partitioned communication `init` functions [15].

5.3 GPU Support

One of the primary architectures in exascale systems is multicore servers combined with accelerators, which are either GPUs or FPGAs currently. Today, GPUs dominate this landscape. Supporting effective integration of partitioned communication with GPU offload is an important optimization for our library, even though layered on top of MPI implementations at present. And, partitioned point-to-point communication can be applied in general to high concurrency code execution environments, both highly threaded CPU code as well as massively threaded data parallel architectures like GPUs and programmable solutions like FPGAs. However, architectures such as GPUs are not well suited for traditional, high-performance MPI communication semantics, and even partitioned communication is not a perfect match for GPUs in its current form (MPI-4.0). Efforts are currently underway to introduce new partitioned communication functions in MPI-4.1 that help to optimize the performance of MPI when used in conjunction with GPUs [4]. Therefore, it is useful to have highly modifiable partitioned communication libraries, such as the one described in this paper, that can be used to prototype future functions and evaluate performance.

`MPI_Pbuf_prepare` is a prototype function in the library that is being used to explore the benefits of knowing the state of readiness of the receive-side buffer prior to calling `MPI_Pready` for the first time in a given point-to-point start/ready/wait iteration [4, 6]. Using the library, we are able to explore the benefits of such solutions by building a robust request-response exchange. In the future, this functionality will be integrated into `MPI_Pbuf_prepare` (currently a NOP). Knowing that the remote buffer is ready to receive data can allow us to reduce the code path complexity significantly. For example, if the remote buffer is always ready, there is no need to keep state on the buffer readiness, nor is there any reason to have to hold the partition locally (at sender or receiver) to ensure buffer readiness.

There are legitimate reasons as to why an implementation may want to aggregate multiple partitions to send them efficiently on a network, but for GPUs this may not be advantageous. In the case of a GPU, we want a simple code path for best performance, with as few branches and pointer-chasing lookups as possible. This is because of the data-centric designs of GPUs, where the processing pipeline has data delivered for threads as they need it. If the data is speculative (branch) or the data is a pointer to another memory location, then this slows the entire thread group in the GPU pipeline.

GPUs are not well-suited for creating commands and sending them to the network interface card (NIC) either. Assembling data in an I/O Vector (iovec) fashion and performing multiple pointer chasing lookups is not ideal in performance terms for a GPU. In addition, all of the information needed to insert a new entry into the command queue of the NIC is needed at the GPU, while much of this information resides in data structures on the CPU, or in the operating system memory space, which is expensive and difficult to access from the GPU. GPUs are, however, well suited for lightweight doorbell notifications since there is limited data needed to ring a doorbell on another PCIe attached device. Therefore, a solution that allows remote buffer readiness to always be guaranteed and allows a CPU to fill a NIC's command queue with doorbell-triggered send/write operations before launching a GPU kernel provides the best performance throughout to the system. The CPU no longer has to wait for the GPU to complete its kernel to send data, and the GPU can trigger transfer operations without requiring branch operations or creating NIC commands. Each component does the jobs for which it is best suited while still allowing data transfers to occur as soon as they are ready.

5.4 Partitioned Collective Communication

The merger of persistent collective operations [12] and partitioned concepts from partitioned point-to-point operations yields the extended operational set of persistent, partitioned collective operations. Partitioning extends logically to many of the collective operations defined by MPI, both in intra- and inter-communicator modes, such as `Bcast`, `Allreduce`, and `Neighbor_alltoall[v|w]`. We plan to prototype the syntax and semantics, explore fully non-blocking (local) `init` functions, and study the utility of both send- and receive-side partitioning for all the operations (or lack thereof).

6 CONCLUSIONS

The focus of this paper was the design and prototyping of a layered library to add the MPI-4 partitioned communication API to an MPI-3.1 library, as well as a native MPI-4 partitioned communication implementation. Initial experience in designing these implementations along with preliminary performance results were presented. The partitioned functionality is complete and baseline performance is acceptable, although actual performance gains were not the key goal at this time. MPI applications can now be modified to use partitioned point-to-point communication while production MPI implementations optimize the performance of these new features with and without accelerator offload.

Our partitioned communication library provides an effective framework for future explorations of the partitioned communication interface, which was designed to be able to support high concurrency for both traditional CPUs as well as accelerator devices (GPUs, FPGAs). As such, the interface will have further enhancements proposed for future MPI standard versions to optimize communication performance for accelerator architectures. This partitioned communication library offers a solid foundation to build such prototypes, then evaluate their performance and programmability improvements.

ACKNOWLEDGMENTS

This work was performed with partial support from the National Science Foundation under Grants Nos. CCF-1562306, CCF-1822191, CCF-1821431, OAC-1923980, OAC-1549812, and OAC-1925603, the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under the Predictive Science Academic Alliance Program (PSAAP-III), Award DE-NA0003966, the University of Alabama at Birmingham, and the Alabama Innovation Fund.

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc. for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S.Department of Energy Office of Science and the National Nuclear Security Administration.

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, the U.S. Department of Energy's National Nuclear Security Administration, or the Sandia National Laboratories.

REFERENCES

[1] James Dinan, Ryan E Grant, Pavan Balaji, David Goodell, Douglas Miller, Marc Snir, and Rajeev Thakur. 2014. Enabling communication concurrency through flexible MPI endpoints. *The International Journal of High Performance Computing Applications* 28, 4 (2014), 390–405.

[2] Matthew G. F. Dosanjh, Ryan E. Grant, Whit Schonbein, and Patrick G. Bridges. 2020. Tail queues: A multi-threaded matching architecture. *Concurr. Comput. Pract. Exp.* 32, 3 (2020). https://doi.org/10.1002/cpe.5158

[3] Ryan Grant. 2019. Partitioned Point-to-Point Communication. https://github.com/mpi-forum/mpi-issues/issues/136. Accessed: 08.13.2020.

[4] Ryan Grant. 2020. Synchronization on Partitioned Communication for Accelerator Optimization. https://github.com/mpi-forum/mpi-issues/issues/302. Accessed: 08.13.2020.

[5] Ryan Grant, Anthony Skjellum, and Purushotham V Bangalore. 2015. *Lightweight threading with MPI using Persistent Communications Semantics.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[6] Ryan E Grant. 2020. *MPI Partitioned Communication.* Sandia National Laboratories. Technical Report SAND2020-10163C, includes MPI_Psync.

[7] 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 - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11501)*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.). Springer, 330–350. https://doi.org/10.1007/978-3-030-20656-7_17

[8] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. 2009. *Improving Performance via Mini-applications.* Sandia National Laboratories. Technical Report SAND2009-5574.

[9] Torsten Hoefler and Andrew Lumsdaine. 2006. Design, Implementation, and Usage of LibNBC. *Open Systems Lab, Indiana University, Tech. Rep* 8 (2006).

[10] MPI Forum. 1994. *MPI: A Message-Passing Interface Standard. Version 1.0.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[11] MPI Forum. 1997. *MPI: A Message-Passing Interface Standard. Version 2.0.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA.

[12] MPI Forum. 2020. *MPI: A Message-Passing Interface Standard. 2020 Draft Specification.* Technical Report. Univ. of Tennessee, Knoxville, TN, USA. Note: This is a MPI-4 Draft Specification.

[13] MPICH 2021. MPICH version 4.0a1 release. https://github.com/pmodels/mpich/releases/tag/v4.0a1. Accessed: 4.22.2021.

[14] Whit Schonbein, Matthew G. F. Dosanjh, Ryan E. Grant, and Patrick G. Bridges. 2018. Measuring Multithreaded Message Matching Misery. In *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11014)*, Marco Aldinucci, Luca Padovani, and Massimo Torquati (Eds.). Springer, 480–491. https://doi.org/10.1007/978-3-319-96983-1_34

[15] Anthony Skjellum and Ryan E Grant. 2020. Add MPI_INFO argument to PSEND_INIT/PRECV_INIT calls for Partitioned Communication Chapter. https://github.com/mpi-forum/mpi-issues/issues/308. Accessed: 12.8.2020.

[16] Anthony Skjellum, Martin Rüfenacht, Nawrin Sultana, Derek Schafer, Ignacio Laguna, and Kathryn Mohror. 2019. ExaMPI: A Modern Design and Implementation to Accelerate Message Passing Interface Innovation. In *High Performance Computing - 6th Latin American Conference, CARLA 2019, Turrialba, Costa Rica, September 25-27, 2019, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1087)*, Juan Luis Crespo-Mariño and Esteban Meneses-Rojas (Eds.). Springer, 153–169. https://doi.org/10.1007/978-3-030-41005-6_11

[17] S. Sridharan, J. Dinan, and D. D. Kalamkar. 2014. Enabling Efficient Multithreaded MPI Communication through a Library-Based Implementation of MPI Endpoints. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.* 487–498.

[18] R Thakur, E Lusk, and W Gropp. 1997. Users guide for ROMIO: A high-performance, portable MPI-IO implementation. (10 1997). https://doi.org/10.2172/564273

[19] Rohit Zambre, Aparna Chandramowlishwaran, and Pavan Balaji. 2020. How I Learned to Stop Worrying About User-Visible Endpoints and Love MPI. *arXiv preprint arXiv:2005.00263* (2020).