Parallel Computing 108 (2021) 102827

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Implementation and evaluation of MPI 4.0 partitioned communication libraries

Matthew G.F. Dosanjh [a], Andrew Worley [b], Derek Schafer [c], Prema Soundararajan [d], Sheikh Ghafoor [b], Anthony Skjellum [c,*], Purushotham V. Bangalore [e], Ryan E. Grant [a,f]

[a] Sandia National Laboratories, Albuquerque, NM, United States of America
[b] Tennessee Tech University, Cookeville, TN, United States of America
[c] University of Tennessee at Chattanooga, Chattanooga, TN, United States of America
[d] University of Alabama at Birmingham, Birmingham, AL, United States of America
[e] University of Alabama, Tuscaloosa, AL, United States of America
[f] Queens University, Kingston, Ontario, Canada

ARTICLE INFO

Keywords:
MPI
Partitioned communication
Point-to-point partitioned

ABSTRACT

Partitioned point-to-point communication primitives provide a performance-oriented mechanism to support a hybrid parallel programming model and have been included in the upcoming MPI-4.0 standard. These primitives enable an MPI library to transfer parts of the data buffer while the application provides partial contributions using multiple threads or tasks or simply pipelines the buffers sequentially. The focus of this paper is the design and implementation of a layered library that provides the functionality of these newer APIs and supports application development using these newer APIs. This library provides an opportunity to explore potential optimizations and identify further enhancements to the APIs. Initial experience in designing this library along with preliminary performance results are presented. In addition, the library is compared to initial prototype libraries that have recently become available that have been updated to the standard-compliant interface. These prototype libraries were built on remote-memory-access (RMA) primitives, offering insight into different implementation strategies. In general, we observe an interesting trade-off space, with the RMA-based implementation proving more performant for send-side partitioning, with increases in perceived bandwidth 8.9x on average over a single send, compared to the persistent-based implementation, which shows improvements 4.0x on average. In comparing the two implementations, we find that the persistent-based implementation enables more overlap for receive-side partitioning up to 5.37X the RMA library's overlap, while the RMA-based implementation provides better send-side performance of up to 70%.

1. Introduction

Partitioned point-to-point communication was introduced [1] as an alternative to the failed MPI endpoints concept [2]. This was based on application developer feedback from a major survey in the US for exascale MPI development that indicated applications wanted RMA-like performance benefits for multi-threading but with a send-receive-type model [3]. 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 type of pipelining can have significant benefits for hybrid programming, such as MPI and OpenMP with fork-join assembly of messages in non-overlapping partitions (send-side overlap of computation and communication) and/or partitioned completion of messages for overlapping receipt and work as data is received (receive-side overlap of communication and computation). Further, this model addresses the concerns raised for the send-receive model and endpoints model regarding the growth of the message queues on the receive-side of transfers while also avoiding the need for the entirety of an MPI implementation function in the lower performance `MPI_THREAD_MULTIPLE` mode [1,4]. Our motivation for the present work is to make a standalone MPI library that enables partitioned communication in MPI implementations. Since partitioned communication is new in MPI 4.0 [5], there are few, if any, public implementations currently available. Availability of a functional library that is portable

* Corresponding author.
E-mail addresses: mdosanj@sandia.gov (M.G.F. Dosanjh), apworley42@tntech.edu (A. Worley), derek-schafer@utc.edu (D. Schafer), prema@uab.edu (P. Soundararajan), sghafoor@tntech.edu (S. Ghafoor), tony-skjellum@utc.edu (A. Skjellum), pvbangalore@ua.edu (P.V. Bangalore), ryan.grant@queensu.ca (R.E. Grant).

https://doi.org/10.1016/j.parco.2021.102827
Received 15 February 2021; Received in revised form 10 June 2021; Accepted 13 August 2021
Available online 28 August 2021

will enable adoption and provide a reference implementation for MPI library-specific implementations that will be optimized over the next months as MPI-4.0 becomes fully supported. We have prototyped a portable MPI extension library (with the MPIX function name denoting functions that are not included in the MPI Standard yet). This extension library implements the partitioned communication API on top of any existing MPI library. This design choice means that our library can be used to help applications experiment with partitioned communication as one of the communication options. These applications can test out how partitioned communication helps new algorithms and provides developers with an opportunity to get familiar with how partitioned communication operations behave. Then, when larger-scale MPI implementations have fully integrated the partitioned communication API into their libraries, developers and applications will be ready for deployment in a production settings without changes to their source code.

2. Background

As interest in efficient multi-threaded MPI programming has increased in recent years [3], there have been several proposals for supporting large numbers of threads with MPI that differ from the traditional multi-threaded environment offered in the MPI standard. Expressing concurrency to the MPI library in a form more complicated than asking for multi-thread safe MPI calls has the potential to unleash greater performance from MPI. The first of such recent attempts was called endpoints [2] and aimed to expose individual threads as network address targets in MPI (i.e., MPI process ranks). Unfortunately, this effort failed to be adopted by the MPI Forum because, although it offered a new way to address threads, it had drawbacks to how hardware could efficiently support such MPI mechanisms. For example, it expanded the MPI "process rank space" from the default `MPI_COMM_WORLD` communicator in codes today to one of size in proportion to the number of threads used. This made it difficult to use hardware-level addressing translations in networking hardware that supported such features [6]. And, much of the performance benefit of the endpoints approach could also be hidden from the user inside the MPI library [7,8].

The successor effort to endpoints was finepoints [1,9]. The approach in finepoints was to express a single MPI operation in such a manner that multiple threads could contribute to it in an efficient manner. This deviated from the endpoints approach significantly since multiple threads contributed to a single operation (as opposed to each performing separate operations), and threads were not individually addressable with an MPI process. This kept the process rank space from expanding beyond that expressed by `MPI_COMM_WORLD`. The motivations for this design were results indicating issues around the processing of the large number of messages that could be extrapolated for future highly concurrent systems [4]. While it had been apparent for some time that network interface cards (NICs) that onloaded most of the communication message processing to the CPU could become a bottleneck with the move to many-core type architectures [10], it was not until recent studies and purpose-built benchmarks were developed that the magnitude of the overheads of processing such large volumes of messages became clear [11]. Finepoints addresses this issue by keeping the MPI level message volume identical to the single-threaded MPI model. Finepoints eventually evolved into MPI partitioned point-to-point communication in the MPI standard in version 4.0, formally approved by the MPI Forum in June, 2021.

2.1. Introduction to partitioned point-to-point communication

MPI partitioned point-to-point communication adds several new functions to the MPI standard. The functions added in MPI 4.0 are shown in Table 1.

To illustrate the basic usage of partitioned point-to-point communication, we provide two code examples demonstrating simple use cases for the send-side in Listing 1 and for the recv-side in Listing 2 as MPI code in C. The basic setup of a partitioned is similar to persistent communication operations. The partitioned operation begins with a call to an initialization function, Line 1 in Listings 1 and 2. These functions returns a handle to a request object used to query the operation. As part of the call, the user defines a number of partitions on each side of the communication. Each partition consists of an independent portion of the communication buffer. Of note is that the number of sender and receiver partitions do not have to be related to each other, only the overall size of buffer must be consistent.

After initialization, the overall partitioned operation is then marked as active by a call to `MPI_Start`, Line 4; however, unlike persistent operations, data transmission is not started immediately. Data transmission is instead devolved to a secondary function associated with each user-defined partition. This function, `MPI_Pready`, is only called on the sending side Listing 1, Line 8, and is called once per partition. Each `MPI_Pready` call activates a portion of the overall request to allow that partition to be transmitted. Before the call to `MPI_Pready`, the buffer partition has not entered the active stage, thus can be modified. After a partition calls `MPI_Pready` and transitions to an active state, the transmission of the partition is handled by the library.

The MPI standard grants a high degree of freedom for how this transmission occurs, including permission to delay some initialization steps up until the point of transmission. Completion of the overall operation can be queried by current existing functions such as `MPI_Test` or `MPI_Wait`. Additional information on the completion status of individual receiver partitions can be found by calling `MPI_Parrived` on the partition in question. Once the operation is complete, resources can be released by calling `MPI_Request_Free` on the appropriate MPI_Request handle.

In our receive-side example in Listing 2, we have denoted several opportunities for the application to do computational work while waiting for the incoming partitioned communication operation to complete. While it is possible for the receive-side to be aware of individual partition completions, we have not demonstrated this in our code example. Next, we will demonstrate the use of the `MPI_Parrived` function in a high level flow diagram.

```
1  MPI_Psend_init(&buffer, partitions,
       count, datatype, dest, tag, comm,
       info, &request);
2
3  for (i=0; i<num_iterations; i++){
4     MPI_Start(&request);
5     /*Parallel loop with some number of
       threads*/
6     parallel for (partition=0; partition
       <10; partition++) {
7        /* Do work to fill partition#
       portion of buffer */
8        MPI_Pready(partition, &request);
9     }
10    MPI_Wait{&request};
11 }
```

Listing 1: Example Partitioned Point-to-Point Communication send-side code

```
1  MPI_Precv_init(&buffer, partitions,
       count, datatype, source, tag, comm,
       info, &request);
2
3  for (i=0; i<num_iterations; i++){
4     MPI_Start(&request);
5     /*Parallel loop with some number of
       threads*/
6     parallel for (loop=0; loop<10; loop
       ++) {
7        /* do compute work  */
```

Table 1
MPI 4.0 and 4.1 partitioned point-to-point communication application programmer interface (API) [5].

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[]`

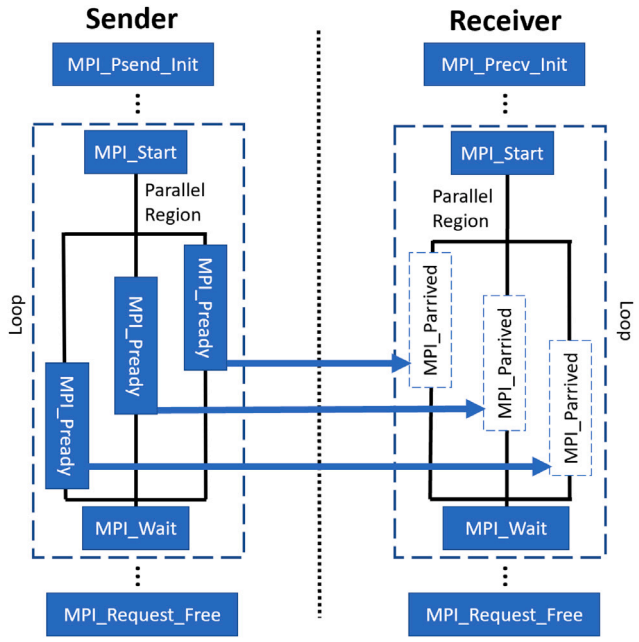

Fig. 1. Overview of Partitioned Point-to-Point Communication.

```
8       MPI_Parrived(&request, loop, &flag);
9       /* do work on early arrivals if
        available if not goto next iteration
        */
10     }
11     MPI_Test(&request, &flag,
        MPI_STATUS_IGNORE);
12 }
```

Listing 2: Example Partitioned Point-to-Point Communication Receive-side Code

The basic control and data flow is illustrated in Fig. 1. Like our previous code examples, the developer calls an initializing function like `MPI_P*_init` followed by a call to `MPI_Start`. As portions of the data buffer become ready to send corresponding to a given partition expressed to MPI in the initialization call, the user calls `MPI_Pready` to indicate that it is safe to move the data in that partition to the receiver. Unlike our previous code examples, Fig. 1 shows the optional receive-side `MPI_Parrived` call that the receiver can use to determine if it can access a given partition's portion of the incoming data buffer before the MPI operation itself is complete (the MPI operation is indicated as complete after a successful return from `MPI_Wait` or `MPI_Test`).

3. Partitioned communication

Partitioned communication is a technique introduced in MPI 4.0 to perform point-to-point communications efficiently with high levels of concurrency (threads). Previous work on partitioned communication has focused on prototype libraries [1,9] and performance that was used as a basis for the interface that was adopted in MPI 4.0, but varied in significant ways, making the libraries not directly comparable to ones that implement the final MPI-4.0-approved interface. Some of these differences in interface included buffer length and management, and whether the MPI library or the user should define the partitions. The official MPI 4.0 interface asks the user to define the equal-size partitions rather than asking the library to handle all edge cases where the user could erroneously define a partition layout that was not easily partitioned.

Partitioned communication is applicable to highly threaded CPU-side MPI codes but has significant predicted utility for GPU-side MPI kernel calls with low expected overheads. Current proposals are underway for MPI 4.1 further to optimize a GPU-side implementation of a `MPI_Pready` call on the GPU to trigger data movement. Therefore, providing partitioned communication libraries as soon as possible to the MPI application developer community is a critical aspect of application readiness for next-generation supercomputers. While the libraries presented here are not GPU-based, they nonetheless allow developers to being the coding process today and make performance analyses for their codes. The process of converting a CPU-side partitioned communication exchange to a GPU-side one is relatively straightforward.

Since partitioned communication is included in MPI 4.0, all MPI libraries will have to add this functionality to their implementations. This work provides the first comparison between two different, reasonable approaches to implementing partitioned communication calls. As partitioned communication adopts some aspects of MPI RMA because of its design being intended to work with Remote Direct Memory Access (RDMA) semantics. Therefore, a partitioned communication implementation based on RMA/RDMA is a logical approach. Partitioned communication also adopts elements of persistent communication in MPI, making it another obvious approach on which to build partitioned communication support. This paper extends previous work focusing on a persistent communication library implementation of partitioned communication [12] by comparing performance between it and an RMA-based library.

3.1. Partitioned operation library via persistent MPI interfaces

A library called MPIPCL was implemented to perform partitioned communication based on persistent point-to-point MPI operations [12]. MPIPCL layers on top of existing MPI implementations and provides all of the MPI 4.0 standard functionality. Therefore, there are not as many opportunities to provide aggregation of messages and overlap of communication and computation with the partitioned operations as with a native solution. Partitions are broken up into multiple persistent point-to-point operations and triggered to move data when the user calls `MPI_Pready`.

Table 2
List of provided functions in MPIPCL.

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. MPIPCL library implementation details

The functions listed in Table 2 represent those that our library has implemented on top of the existing MPI library. The six partitioned communication functions on the left-hand side that start with MPI_ represent new functionality that is to appear in the MPI 4.0 standard, the remaining two are not yet accepted in the standard and use the MPIX_ denotation. The functions on the right are reimplemented (augmented) functions to enable support for the new request object (an attendant side effect of a layered implementation).

3.2.1. New request object

Normal MPI_Request objects have no mechanism to keep track of partial progress of the request. Because partitioned requests are a collection of underlying partial communications, it is necessary to maintain a more flexible record of completion than would normally be needed. As such, a new request object was created for use with the MPIPCL library. As shown in Listing 3, the new request object contains metadata about the overall request as well as an internal array of requests to fulfill.

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

Listing 3: The MPIX_Request object for the MPI persistent operation-based library (MPIPCL).

3.2.2. Internal request negotiation

MPIPCL is built upon MPI point-to-point communications and as such requires some additional information to be held in MPI_Request objects. These expanded request objects are not part of the MPI standard and therefore are delineated as experimental with the use of MPIX at the start of their function names as the standard requires for any non-standard extensions to MPI. Each MPIX_Request object contains a number of standard requests, so the number of sends and receives needs to be balanced. The API has no defined mechanism for this to occur; but, both sides of the communication need to agree in advance on the internal number of requests generated. The library is meant to be portable; as such, it does not have access to the implementation-specific control structures. And, the library is forced to use additional communication requests to share the required metadata. This is further complicated by the realization that this setup phase is a blocking process, yet MPI_Psend_init, MPI_Precv_init, and MPI_Start are all required to be non-blocking functions. The implemented solution to this is the spawning of a new thread to execute part of this initialization operation while still returning control of the main thread back to the program, see Fig. 1. This thread enables the program to continue (return immediately) until progress is required of the partitioned request, thus effectively postponing any blocking activities until a blocking procedure is called. Currently, communication is limited to sharing the number of internal requests being sent to the receiver, but could be used as part of data aggregation or optimization algorithms (something intended for fully optimized partitioned communication implementations).

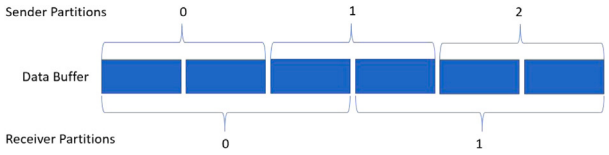

Fig. 2. Partition Remapping.

3.2.3. Partition remapping

When the number of send-side and receive-side partitions are unequal, the data can be remapped inside the receiving buffer (to further abstract the user defined partitions away from the internal communication structure). The current method is a simple mapping based on offsets from the start of the buffer. The MPI_Parrived function calculates which portion of the buffer is needed to for the requested partition then finds which internal receive requests contain some part the required data. The data is marked as available only after all the dependent partitions have arrived. This effectively allows the user to define any number of partitions at either end, while permitting internal message aggregation to proceed unhindered. Fig. 2 demonstrates this approach.

3.2.4. Limitations

Because this library current layers on top of existing MPI implementations, there are certain limitations that we would like to address. 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 primitive built-in datatypes and contiguous datatypes.[1]

It is anticipated that applications that use partitioned communication will use multiple threads to achieve the best performance; that is, multiple threads 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 thread support in the MPI implementation, necessitating at least the MPI_THREAD_SERIALIZED level of support 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. Additionally, since this library is not fully implemented in a unified way inside any one library, we are limited in how we can optimize our functions. However, we are still aiming to achieve minimal extra overhead (if any)

[1] The most performant modes of partitioned communication are likely to be contiguous, partitioned buffers, so this is not a major limitation for an initial library implementation. But, addressing packing/unpacking of partitions is important future work for non-contiguous partitions based on MPI user-defined (derived) data types. We intend to consider support for such user-defined datatypes as a future part of this effort.

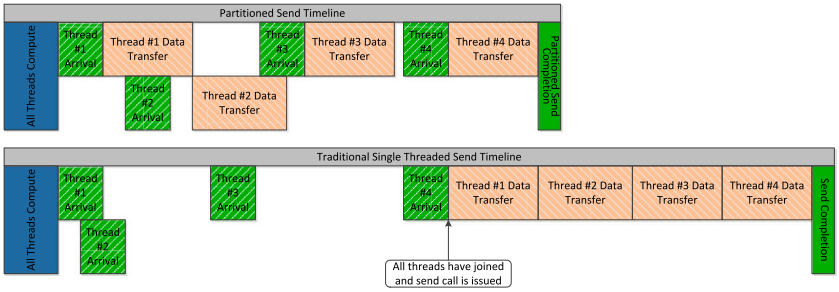

Fig. 3. An example of early-bird communication, where with partitioned communication data movement occurs when portions of the buffer become available, thereby allowing use of the network over a longer period of time than the traditional send model. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 3
List of provided functions in RMA-based partitioned communication library.

Partitioned functions	Limitations
`MPI_Psend_init`	Not non-blocking
`MPI_Precv_init`	Not non-blocking
`MPI_Pready`	None
`MPI_Pready_range`	None
`MPI_Pready_list`	None
`MPI_Parrived`	None

to inform 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 see the performance benefits of persistent point-to-point APIs, if they are optimized by the underlying MPI implementation.

3.3. RMA-based library

An alternative implementation approach for MPI partitioned communication is to build the operations on top of MPI RMA. While this is not as ideal as building an implementation on top of RDMA primitives, using RDMA directly is only possible when done internally to an MPI library, making the changes implementation specific and non-portable between MPI libraries. As such, we have developed a library based on early prototypes [1] used to prove the usefulness of MPI partitioned communication. The original prototype libraries needed additional implementation work to bring them into compliance with the MPI specification as well as further optimizations to represent the benefit of some of the interfaces changes made for MPI 4.0. A list of the MPI partitioned communication functions supported by this library is given in Table 3.

This implementation uses two, single-direction RMA windows: one for the data and another for completion flags. To ensure isolation from other partitioned communication operations, these windows use a separate communicator between the two processes involved. Because of the use of a separate flag window, the two processes need to agree on a number of completion partitions (the number of completion flags) during initialization. For simplicity, this is done using the greatest common denominator, to ensure both send and receive partitions only correspond to a single flag.

When `MPI_Pready` is called, `MPI_Put` is called corresponding to the referenced partition. After that, a completion partition counter is atomically fetched and incremented, if that counter has reached a threshold (the number of send partitions per completion partition) `MPI_Flush` is called on the data window, `MPI_Put` is called setting the flag for the completion partition, and finally `MPI_Flush` is called on the completion flag window.

3.3.1. Limitations

Like the persistent-operation-based library, the RMA-based library is limited to building on top of MPI RMA. This means that the partitioned operations are limited by the feature set available at the MPI level and cannot take full advantage of lower-level RDMA primitives. For example, the RMA-based library cannot provide an entirely non-blocking initialization call as it is based on `MPI_Win_create`, which is itself a blocking call. There are also limitations to the number and types of optimizations that can occur with regards to aggregation as the RMA-based library is not aware of the underlying hardware being used by the MPI implementation. Therefore, parameters like the network Maximum Transmission Unit (MTU) cannot be optimized for since there is no way of knowing what the relevant network values to optimize for are. This limitation can be remedied by using a fully integrated approach in the MPI libraries that would allow for further optimization. However, even with the RMA-based libraries limitations, application developers can build upon it with the knowledge that the eventual integrated support will provide better performance and less restrictive initialization semantics.

4. Experimental results

4.1. Experimental platform and benchmarks

To compare both of these approaches, we have run a modified version of partitioned communication benchmark used in our prior work [1,13]. These modifications include updating the interface to match the MPI 4.0 spec In this section, we will compare and contrast the two MPI partitioned communication library implementations with each other and with traditional MPI communication sends. All of the tests take two parameters to define a computation region, compute and noise. Compute is the expected compute time, and noise is the percentage delay that is applied to a single thread's compute time. Noise serves as an analogue to the system noise we would expect to see on a system, such as having the operating system functionality interrupt computation on a core. We will examine two types of traditional MPI send approaches. The first we will refer to as single-send. Single-send is a single `MPI_(I)Send` operation for the entire buffer, the most commonly used MPI point-to-point communication call. The second approach we refer to as multi-send. The multi-send approach uses a separate `MPI_(I)Send` call for each thread which is the most similar method in traditional MPI to accomplish an approach like partitioned communication. Third, we will test sender-side partitioning, by converting the `MPI_(I)Send` calls in the multi-send test with `MPI_Pready` calls on different partitions a single combined partitioned request. Finally, we added a new test to measure receive-side partitioning. This test adds a measurement to time between when the first partition has arrived (as reported by `MPI_Parrived`) and when the final partition arrived (as reported by `MPI_Wait`).

These benchmarks emulate a threaded computation phase with noise (jitter) on a single thread prior to a communication phase every

iteration. This emulated computation is done by sleeping the threads for the amount of time equal to a user-specified computation time, with the noise-impacted thread sleeping for a user-specified additional percentage of that time. Of these two tests, the first test measures the performance effect of sender-side 'early bird' communication and the second test measures the effect of receiver-side early notification. Both use a compute time and noise factor to emulate a computation period where a single thread is delayed by noise (details are in Section 4.2). The first test measures "perceived bandwidth" from the sender-side perspective; the bandwidth as it appears from the last thread joining the communication to the completion of the communication. Perceived bandwidth is equivalent to how fast a network would have to be to observe the same time period between calling the single large buffer send call and the MPI completion function (e.g., `MPI_Wait` or `MPI_Test`) returning. A visual example of this can be seen in Fig. 3 between the time noted as all threads being completed to the green completion box on the timeline. Because this test does not utilize `MPI_Parrived`, the number of partitions at the receiver can be set to 1. The second test measures the receive-side extra compute time available through early notification; the time between the first partitioned being flagged as arrived and the final partition being flagged. Also measured in this test is *communication time* as perceived by the receiver (the time from when the last thread on the sender-side joins the communication to the communication being complete, on the receiver process).

For these experiments, we used a cluster, Mutrino, that is a testbed for the Trinity supercomputer. These experiments use the Haswell partition of this cluster. This cluster is a Cray XC40 with 16 core Xeon E5-2698v3 processors running at 2.3 GHz over an Aries interconnect. Each test was run in the MPI threading mode required by the operations performed with the best performance (i.e. single send is run not run in `MPI_THREAD_MULTIPLE` mode). Each test was run 10 times (to randomize allocated nodes) for 100 iterations with a dedicated node for each process, unless otherwise noted. For the send-side results, there are additional baselines of multi-send (replacing partitioned communication with individual sends and receives) and single send (where the communication happens after the threads converge). All of these tests use a thread per core (in this case 64) each responsible for a single partition (a total of 64 partitions).

4.2. Factors in assessing partitioned communication performance

Partitioned communication provides several performance benefits over traditional MPI point-to-point communications in that it allows early data movement before an entire buffer is ready. This motivates a testing environment which allows for typical system noise that causes slight variation in MPI process loop completion times. This manifests as a range of arrival times to any given synchronization point in the application code. Typically such points occur as communication completion events, either for collective operations or for point-to-point communication. For a typical system, this variation is in the low single digit range, with 4% representing a reasonable target in this range [14]. Applications can be impacted with larger amounts of noise, especially those with unbalanced loads and resource contention. We measured larger amounts of noise, 10% in this paper, to explore the range of realistic noise levels. Partitioned communication can take advantage of the time variance between different thread compute completion by sending data as soon as it becomes available from the threads that first complete the computational task. This "early-bird" communication [1] enables overlapping of data transmission from the completed portions of a buffer with computation that is still occurring for incomplete portions of a data buffer. Therefore, the analysis of partitioned communication must include an evaluation of the overlap potential enabled by the different implementation approaches. An example of the overlap being measured is shown in Fig. 3.

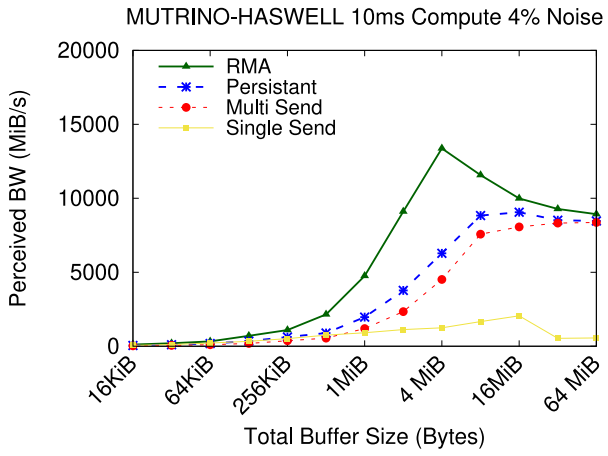

Fig. 4. 10 ms compute 4% noise.

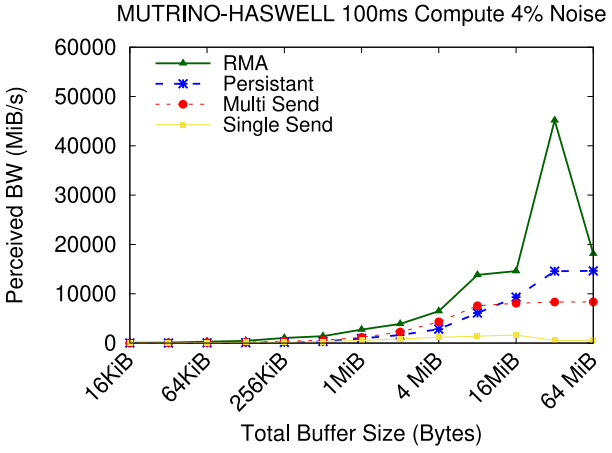

Fig. 5. 100 ms compute 4% noise.

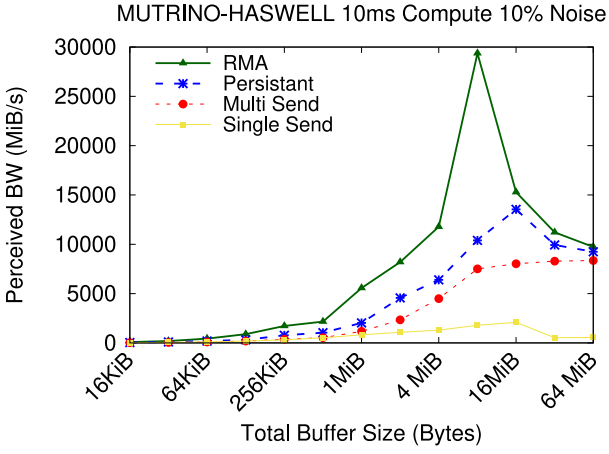

Fig. 6. 10 ms compute 10% noise.

4.3. Send-side results

Figs. 4–7 show the results for the send-side experiments. There are a number of different independent variables, including compute time, noise, transfer mechanism, and total buffer size. There are a couple of general observations one can make here, including that both partitioned communication implementations perform better than single send

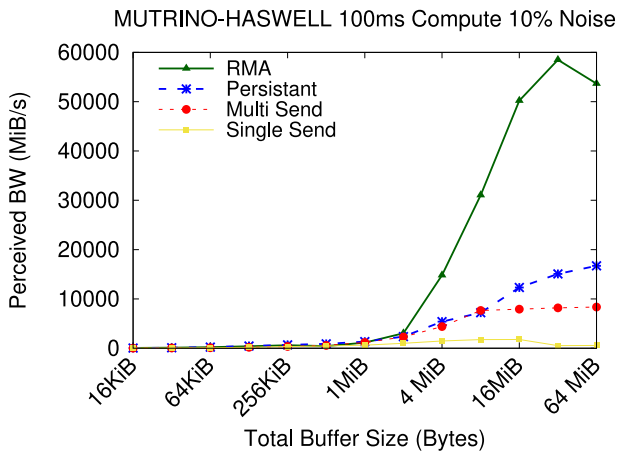

Fig. 7. 100 ms compute 10% noise.

Table 4
Delay and peak perceived BW for the RMA implementation.

Compute & noise	Delay	Buffer size	Peak perceived bandwidth
10 ms 4%	0.4 ms	4 MiB	13372 MiB/s
10 ms 10%	1 ms	8 MiB	29367 MiB/s
100 ms 4%	4 ms	32 MiB	45166 MiB/s
100 ms 10%	10 m	32 MiB	58484 MiB/s

and multi-send. Additionally, the RMA-based implementation performs better than the persistent implementation. This is due to a number of factors including reduced overhead from one-sided communication and differences in the notification method. For the notification method, the RMA implementation relies on a small window of completion flags, in this case a single element, to provide notification reducing the amount of messages on the network. On the other hand, the persistent implementation uses a point-to-point communication for each send-side partition; this requires network traffic for completion of each send-side partition regardless of the number of receive-side partitions. The peaks observed in Figs. 4–7 demonstrate the channel packing effect that is the result of early-bird communication. At the point at which the partitioned communication can send all of the data that is available early and then have little to no time to wait until the final laggard data arrives, the channel is well packed, leading to peak performance. Having more data to send than can be transmitted in the early-bird phase and the laggard data leads to network having a backlog of data to send after the laggard threads have completed. This impacts the overall perceived network bandwidth as the data transfer is dominated by the data transmission volume lessening the perceived impact of the early bird data transmission opportunities. One can observe that the persistent communication implementation occasionally peaks at a larger message size than RMA, this is due to the overheads in the persistent send versus the RMA Put, which leads to a shift in the channel packing best performance buffer size. Ultimately with extremely large message sizes, the partitioned communication and transitional send methods will converge as the time to send the data dominates any opportunities to send early-bird data.

Shown in Table 4 is noise-induced delay compared against the highest performing buffer size for the RMA implementation. This shows that increased delay is associated with better performance at larger total buffer sizes. Additionally, the amount of noise induced delay also appears to be associated with higher-peak bandwidth. This matches with expectations as the noise induced delay provides a constant overlap with communication. As that delay grows, more communication calls can be progressed before the last thread reaches the communication. Additionally, as the total buffer size grows, we expect the perceived bandwidth to converge back down to the measured bandwidth (assuming a constant delay) as after the peak, any extra data must be handled after the last thread joins the communication section.

We observe similar maxima for two of the tests using MPIPCL. In the 10 ms 4%, we observe a maxima at 16 MiB total buffer size with 9058 MiB/s. Additionally, in the 10 ms 10% we observe a maxima at 16 MiB with 13546 MiB/s. These are not as high as the peaks for the respective RMA implementation experiments. We do not observe maxima in the 100 ms. The results will eventually converge with the single-send results.

4.4. Receive-side results

Figs. 8–11 show the results for the receive-side experiments. The independent variables are mostly the same, but two dependent variables are measured, communication time and extra compute time. Communication time is the time from when the first `MPI_Pready` call is made, until the point at which MPI_Wait returns on the receiver. Extra compute time measures from when the first partition is flagged as complete (via `MPI_Parrived`) to the time `MPI_Wait` returns. This measures the additional time to work on the data that has arrived early before computation could normally begin. It is important to note that the number of receive-side partitions is equal to the number of send-side partitions for these tests.

We can make some general observations about this data. First is that, for both cases, the extra compute time is often greater than the observed communication time, particularly for the 100 ms tests. In the 100 ms 4% test for example, we observe extra compute times of up to 5.37 times larger than communication time. This means that the first partition arrives before the last partition is sent. This is why these results are measured in time, rather than perceived bandwidth; the time between the last partition to be called for `MPI_Pready` and first partition to return true for `MPI_Parrived` can be negative.

The second observation we can make is that, as transfers become increasingly larger, the extra compute time decreases for the RMA implementation. This is because the `MPI_Put` calls for the data and the notification are separate calls to the network; if all the data transfers enqueued before the notification transfers (which becomes increasingly likely with larger buffer sizes) the data traffic may be prioritized. By way of contrast, because the persistent implementation uses a single point-to-point message, this approach avoids the ordering problem. We observe that the extra compute time actually increases at high message counts, due to the extra time spent in communication.

Finally, we observe that, for smaller buffer sizes, the RMA implementation has a lower communication time than the persistent implementation. At larger buffer sizes, this can change to where the persistent implementation performs better. The exception to this is shown in Fig. 11 where the time that the emulated noise is large (10% of 100 ms). This contrasts with the send-side experiment where the RMA implementation is more performant in all of the experiments we have done. The communication time results of the receive-side experiments can be compared to the send-side experiments as they both measure the time from the last call to MPI_Pready to the MPI_Wait call. In both of these experiments, the time is measured on the send side to measure communication time, so the major difference in these experiments is how many receive-side partitions are declared. One of the major things that has changed (that could be the cause of this behavior) is the extra notification calls. In the send-side experiments, the RMA implementation used a single completion flag resulting in 64 calls to `MPI_Put` on the data buffer, one `MPI_Flush` call on the data buffer, one `MPI_Put` call on the completion flags buffer, and one call to `MPI_Flush` on the completion flags buffer. However, in the receive-side tests, the RMA implementation makes 64 of each of those calls. This could be detrimental to performance because of a significant increase in overhead.

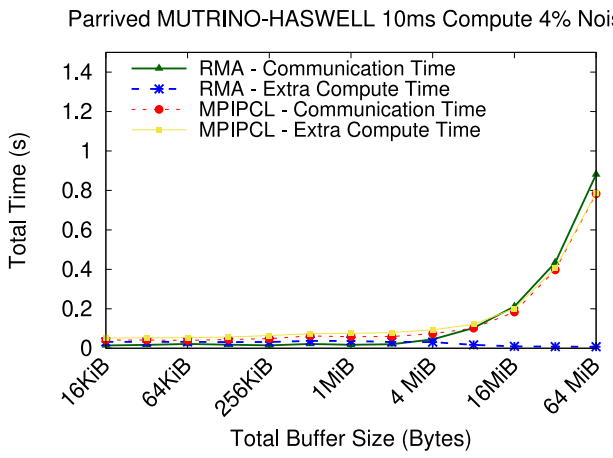

Fig. 8. Receive Side - 10 ms compute 4% noise.

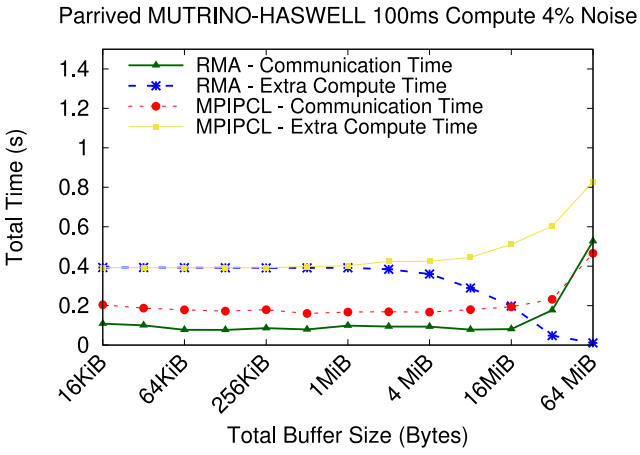

Fig. 9. Receive Side - 100 ms compute 4% noise.

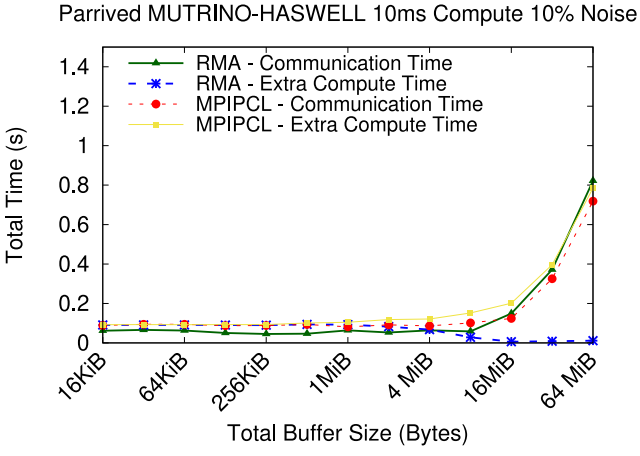

Fig. 10. Receive Side - 10 ms compute 10% noise.

4.5. Discussion

Given the results of our experiments, it is clear that both approaches are useful. However, each approach has benefits that depend on the particular use case for the partitioned communication. The RMA implementation has better data transmission performance but obtains this performance by avoiding data fences that impact the ability of

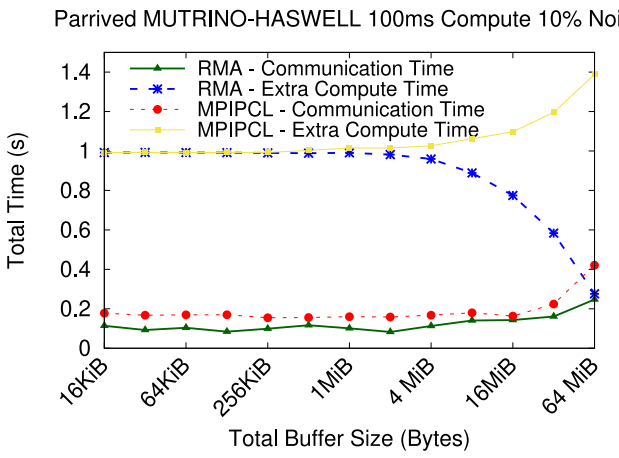

Fig. 11. Receive Side - 100 ms compute 10% noise.

the `MPI_Parrived` function to provide fine-grain notification on partition completion. The notification mechanism for the persistent implementation makes it better suited when using `MPI_Parrived` on a large buffer transfer. This motivates a further exploration of the impact of finer-grain arrival notifications with the RMA method. This should be done in addition to further data transmission optimizations like data and notification aggregation, full integration into an MPI library, and a direct conversion from the partitioned interface into RDMA calls.

To evaluate how these results might correlate to impact on real applications, it is important to view these results in the context of real world application behavior. Prior work demonstrates applications, such as LULESH (a hydrodynamics proxy application), are targeted at 50–100 ms per iteration of compute kernels [15]. Additionally, a survey of MPI usage in applications has observed that message sizes fall into a bi-modal distribution pattern, with most messages containing under 800KiB and another set of sizes greater than 1 MiB [16]. This obviously will fluctuate with system and application set-up (number of cores, memory available, number of processes per node, etc.) but this gives us a reasonable idea of the potential impact on today's applications. In addition, more optimization could occur during the initialization phase with better knowledge of the network that would be possible with a full MPI library integration. This would help in negotiating the arrival side notification to optimize for bandwidth while also allowing more overlapping compute time on early arriving data on the receiver.

5. Related work

Integrating threading internally in an MPI implementation has been attempted in FG-MPI [17]. FG-MPI treats threads as the equivalent of MPI processes. This allows for many concurrent threads and consequently creates a large amount of state for each thread/process. The concepts in FG-MPI were never incorporated into the MPI standard. Benchmarks for testing and profiling MPI threading [18] and RMA have been developed [19], but these focus on existing MPI functionality. The general concept of composing RDMA messages into a large transaction has been explored for application in unreliable datagram networks at the hardware level [20,21]. Similar benchmarks have also been developed for other one-sided communication APIs like OpenSHMEM [22]. Commercial MPI's such as MPI/Pro, which were designed for internal concurrency and the option of blocking completion notification (to avoid polling), are no longer widely available [23]. Approaches for improving performance for large scale MPI jobs by dynamically building communicators and controlling thread levels in "Sessions" [24] have been adopted into MPI 4.0.

Message aggregation is a well known method for networks, having been explored in MPI RMA [25]. Aggregation methods are also common with TCP/Ethernet networking. Previous work has introduced earlier versions of the partitioned communication libraries expanded and enhanced in this work [1,12].

The MPI Forum has had a proposal before it to enhance support for threads using endpoints [2], in which each thread can be assigned a unique rank in an endpoint communicator. However, endpoints never attempted to address the underlying communication model, and the endpoints model only adds the ability to address messages to specific threads. This work differs from previous efforts by the requirements it places on applications and the corresponding decrease both in resources needed by MPI and in synchronization overhead achieved.

6. Conclusions

This paper has evaluated two key implementation methods for MPI partitioned communication via libraries built on top of existing MPI implementations. Overall, it was shown that there are benefits to computation and communication overlap that occur with partitioned communication regardless of the implementation approach. Using remote-memory access (RMA), we observed additional benefits for data-transmission performance over the persistent-communication-based approach, and better arrival notification and opportunity for early computation on the buffer with persistent communication. We also found that partitioned communication performed best versus traditional MPI single- and multi-send methods when buffers are sufficiently large to take advantage of early thread arrival and data transmission.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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.

This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

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.

The authors wish to thank the anonymous reviewers whose feedback helped improve this paper significantly.

References

[1] R.E. Grant, M.G.F. Dosanjh, M.J. Levenhagen, R. Brightwell, A. Skjellum, Finepoints: Partitioned multithreaded MPI communication, in: M. Weiland, G. Juckeland, C. Trinitis, P. Sadayappan (Eds.), High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings, in: Lecture Notes in Computer Science, vol. 11501, Springer, 2019, pp. 330–350, http://dx.doi.org/10.1007/978-3-030-20656-7_17.

[2] J. Dinan, R.E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling communication concurrency through flexible MPI endpoints, Int. Jour. High Perform. Comput. Appl. 28 (4) (2014) 390–405.

[3] D.E. Bernholdt, S. Boehm, G. Bosilca, M. Venkata, R.E. Grant, T. Naughton, H. Pritchard, G. Vallee, A survey of MPI usage in the U.S. Exascale Computing Project, Concurr. Comput.: Pract. Exper. (2018) http://dx.doi.org/10.1002/cpe.4851.

[4] W. Schonbein, M.G. Dosanjh, R.E. Grant, P.G. Bridges, Measuring multithreaded message matching misery, in: M. Aldinucci, L. Padovani, M. Torquati (Eds.), Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings, in: Lecture Notes in Computer Science, vol. 11014, Springer, 2018, pp. 480–491, http://dx.doi.org/10.1007/978-3-319-96983-1_34.

[5] MPI Forum, MPI: A Message-Passing Interface Standard 4.0., Tech. Rep., Univ. of Tennessee, Knoxville, TN, USA, 2021.

[6] B.W. Barrett, R. Brightwell, R.E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, T. Hoefler, A.B. Maccabe, T. Hudson, The Portals 4.2 Network Programming Interface, Tech. Rep., (SAND2018-12790) Sandia National Laboratories (SNL-NM), Albuquerque, NM (United States), 2018.

[7] R. Zambre, A. Chandramowliswharan, P. Balaji, How I learned to stop worrying about user-visible endpoints and love MPI, in: Proceedings of the 34th ACM International Conference on Supercomputing, 2020, pp. 1–13.

[8] R. Zambre, A. Chandramowlishwaran, P. Balaji, Scalable communication endpoints for MPI+ threads applications, in: 2018 IEEE 24th International Conference on Parallel and Distributed Systems, ICPADS, IEEE, 2018, pp. 803–812.

[9] R. Grant, A. Skjellum, P.V. Bangalore, Lightweight Threading With MPI using Persistent Communications Semantics, Tech. Rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[10] M.G.F. Dosanjh, R.E. Grant, P.G. Bridges, R. Brightwell, Re-evaluating network onload vs. offload for the many-core era, in: IEEE Intl. Conf. on Cluster Computing, CLUSTER, IEEE, 2015, pp. 342–350.

[11] W. Schonbein, S. Levy, W.P. Marts, M.G. Dosanjh, R.E. Grant, Low-cost MPI multithreaded message matching benchmarking, in: International Conference on High Performance Computing and Communications, HPCC, 2020, pp. 1–10.

[12] P.V. Bangalore, A. Worley, D. Schafer, R.E. Grant, A. Skjellum, S. Ghafoor, A portable implementation of partitioned point-to-point communication primitives, in: EuroMPI, 2020, pp. 1–3.

[13] M. Dosanjh, R.E. Grant, Receive-Side Partitioned Communication, Technical Report SAND2019-11403, Sandia National Laboratories, 2019.

[14] K.B. Ferreira, P. Bridges, R. Brightwell, Characterizing application sensitivity to OS interference using kernel-level noise injection, in: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, IEEE, 2008, pp. 1–12.

[15] E.A. León, I. Karlin, R.E. Grant, Optimizing explicit hydrodynamics for power, energy, and performance, in: 2015 IEEE International Conference on Cluster Computing, IEEE, 2015, pp. 11–21.

[16] K.B. Ferreira, S. Levy, Evaluating MPI message size summary statistics, in: 27th European MPI Users' Group Meeting, EuroMPI/USA '20, Association for Computing Machinery, New York, NY, USA, 2020, pp. 61–70, http://dx.doi.org/10.1145/3416315.3416322.

[17] H. Kamal, A. Wagner, An integrated fine-grain runtime system for MPI, Computing 96 (4) (2014) 293–309, http://dx.doi.org/10.1007/s00607-013-0329-x.

[18] R. Thakur, W. Gropp, Test suite for evaluating performance of MPI implementations that support MPI_THREAD_MULTIPLE, in: Recent Advances in Parallel Virtual Machine and Message Passing Interface, Springer, 2007, pp. 46–55.

[19] M.G.F. Dosanjh, T. Groves, R.E. Grant, R. Brightwell, P.G. Bridges, RMA-MT: a benchmark suite for assessing MPI multi-threaded RMA performance, in: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid, IEEE, 2016, pp. 550–559.

[20] R.E. Grant, M.J. Rashti, A. Afsahi, P. Balaji, RDMA capable iWARP over datagrams, in: IEEE Int. Parallel & Distributed Processing Symp., IPDPS, IEEE, 2011, pp. 628–639.

[21] M.J. Rashti, R.E. Grant, A. Afsahi, P. Balaji, iWARP redefined: Scalable connectionless communication over high-speed ethernet, in: Intl. Conf. on High Performance Computing, HiPC, IEEE, 2010, pp. 1–10.

[22] H. Weeks, M.G.F. Dosanjh, P.G. Bridges, R.E. Grant, SHMEM-MT: A benchmark suite for assessing multi-threaded SHMEM performance, in: Workshop on OpenSHMEM and Related Technologies, Springer, 2016, pp. 227–231.

[23] R. Dimitrov, A. Skjellum, Software architecture and performance comparison of MPI/Pro and MPICH, in: Int. Conf. Computational Science, ICCS, 2003, pp. 307–315, http://dx.doi.org/10.1007/3-540-44863-2_31.

[24] D. Holmes, K. Mohror, R.E. Grant, A. Skjellum, M. Schulz, W. Bland, J.M. Squyres, MPI sessions: Leveraging runtime infrastructure to increase scalability of applications at exascale, in: Proceedings of the 23rd European MPI Users' Group Meeting, 2016, pp. 121–129.

[25] N. Hjelm, M.G.F. Dosanjh, R.E. Grant, T. Groves, P. Bridges, D. Arnold, Improving MPI multi-threaded RMA communication performance, in: Proc. of the Int. Conf. on Parallel Processing, 2018, pp. 1–10.