A Portable Implementation of Partitioned Point-to-Point Communication Primitives

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

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

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

Ryan E. Grant
Sandia National Laboratory
Albuquerque, NM, USA
regrant@sandia.gov

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

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

ABSTRACT

Partitioned point-to-point communication primitives provide a performance-oriented mechanism to support 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 a layered library that provides the functionality of these newer APIs and support application development using these newer APIs. This library will also provide 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.

KEYWORDS

high-performance computing, hybrid programming, middleware, message passing, portability

ACM Reference Format:
Purushotham V. Bangalore, Andrew Worley, Derek Schafer, Ryan E. Grant, Anthony Skjellum, and Sheikh Ghafoor. 2020. A Portable Implementation of Partitioned Point-to-Point Communication Primitives. In *EuroMPI/USA 2020, September 21–24, 2020, Austin, TX*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Partitioned point-to-point communication was introduced [5] as an alternative to the failed MPI endpoints concept [1]. 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+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). Furthermore, this model addresses the concerns raised for the send-receive model and endpoints model of 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 [5].

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 [2, 3], there are few, if any, public implementations currently available. Availability of a functional library that is portable 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 MPIX (MPI-eXtension) library that 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.

The remainder of this poster précis is as follows: Section 2 covers the core of our work: the design of our library and related implementation details. Section 3 then covers the results we aim to find and as well areas we are planning to take our standalone library.

2 DESIGN OVERVIEW

The library is designed to run on top of existing MPI implementations and utilizes the underlying persistent point-to-point APIs to

EuroMPI/USA 2020, September 21–24, 2020, Austin, TX

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

Partitioned Functions	Reimplemented functions
MPI_Psend_init	MPI_Start
MPI_Precv_init	MPI_Startall
MPI_Pready	MPI_Wait
MPI_Pready_list	MPI_Waitall
MPI_Parrived	MPI_Waitany
	MPI_Waitsome
	MPI_Test
	MPI_Testall
	MPI_Testany
	MPI_Testsome
	MPI_Request_free

Table 1: List of Provided Functions

complete the partitioned requests. In order to track the progress of a partitioned communication—since this external library does not have access to the internals of the `MPI_Request` object in an MPI implementation—a new request object was created along with several supporting functions to match current semantics. The core functions in this library utilize this object to fulfil the partitioned communication APIs accepted by the standard.

2.1 Implementation Details

The functions listed in Table 1 represent those that our library has implemented on top of the existing MPI library. The five partitioned communication functions on the left-hand side represent new functionality that is to appear in the MPI 4.0 standard. The functions on the left are reimplemented (augmented) functions to enable support for the new request object.

2.1.1 New Request Object. Normal `MPI_Request` objects have no mechanism to keep track of partial progress of the request. As partitioned requests are a collection of underlying partial communications, it is necessary to maintain a more flexible record of completion. As such a new request object was created for use with the library. As shown in Listing 1, the new request object contains metadata about the overall request as well as an internal array of requests to fulfil.

```
typedef struct _mpix_request
{
  int state;
  int size;
  int side;
  int sendparts;
  int recvparts;
  int readycount;
  MPI_Request *request;
} MPIX_Request;
```

Listing 1: MPIX_Request Object

2.1.2 Internal Request Negotiation. As the library is built on top of existing point-to-point APIs, certain conditions need to be maintained. Since each `MPIX_Request` object contains a number of standard requests, 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. As such 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`, `MPIX_Precv_init`, and `MPIX_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 Figure 1. This allows the program to continue (return immediately) until progress is required of the partitioned request, thus effectively postpones 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).

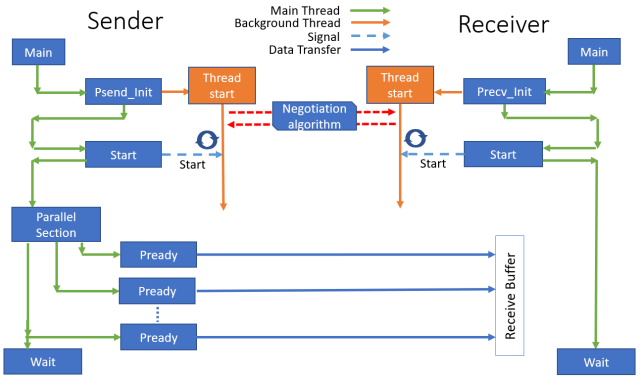

Figure 1: Basic Library Control Flow

2.1.3 Partition Remapping. When the number of send-side and receive-side partitions are not equal, to further abstract the user defined partitions away from the internal communication structure, the data can be remapped inside the receiving buffer. 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.

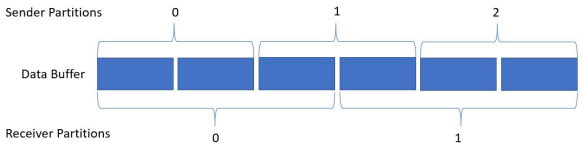

Figure 2: Partition Re-mapping

2.2 Limitations

As this library current exists on top of existing MPI implementations, there are some limitations we'd 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 datatypes and contiguous datatypes. We intend to consider support for other user-defined datatypes as part of this effort.

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` 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) to inform users of potential performance benefits they might get from using partitioned communications. As 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.

3 ONGOING EFFORTS

Although library is still in development, we have plans to benchmark the library against more traditional methods of communication. In particular, the library will be tested against `MPI_Send`, `MPI_ISend`, and `MPI_Send_init`. Thod testing will consist of multiple runs using the buffer size, partition count, and datatype as control parameters. While we expect that a single partition will have similar performance to the persistent send, further correlations between the parameters require additional testing.

As we continue to work on the coverage of our library, we are aiming to eventually support all valid datatype options that can be passed to the partitioned communication functions, which includes derived datatypes and differing datatypes[1] at each end of the communication. Further experimentation with the setup mechanism in Figure 1 is also expected. We are also looking into the integration of the MPI_PSync operation (to be proposed for MPI-4.1) into our model [4][2].

[1]By differing datatypes, we mean datatypes that have compatible typemaps, but different structures.

[2]This added function supports channelized sequencing between the sender and receiver ends of the partitioned point-to-point communication to ensure that receive buffers are available when the first partitions begin arriving; it is an important optimization feature and particularly of importance when the producing and/or consuming threads may be offloaded to accelerators such as GPUs or FPGAs.

ACKNOWLEDGMENTS

This work was performed with partial support from the National Science Foundation under Grants Nos. CCF-1562306, CCF-1822191, CCF-1821431, and OAC-1925603. 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 or the Sandia National Laboratories.

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

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] Message Passing Interface Forum. 2015. *MPI: A Message-passing Interface Standard, Version 3.1 ; June 4, 2015.* High-Performance Computing Center Stuttgart, University of Stuttgart. https://books.google.com/books?id=Fbv7jwEACAAJ

[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 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)*, Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan (Eds.), Vol. 11501. Springer, 330–350. https://doi.org/10.1007/978-3-030-20656-7_17