Partitioned Collective Communication

Daniel J. Holmes
Collis Holmes Innovations Ltd
Scotland, UK
danholmes@chi.scot

Anthony Skjellum
Univ. of Tennessee at Chattanooga
Chattanooga, TN 37403, USA
tony-skjellum@utc.edu

Julien Jaeger
CEA, DAM, DIF
Arpajon, F-91297, France
julien.jaeger@cea.fr

Ryan E. Grant
Queen's University
Kingston, Ontario, Canada
ryan.grant@queensu.ca

Purushotham V. Bangalore
University of Alabama
Tuscaloosa,AL 35401, USA
pvbangalore@ua.edu

Matthew G.F. Dosanjh
Sandia National Laboratories
Albuquerque, NM 87185, USA
mdosanj@sandia.gov

Amanda Bienz
University of New Mexico
Albuquerque, NM, USA
bienz@unm.edu

Derek Schafer
Univ. of Tennessee at Chattanooga
Chattanooga, TN 37403, USA
derek-schafer@utc.edu

***Abstract*—Partitioned point-to-point communication and persistent collective communication were both recently standardized in MPI-4.0. Each offers performance and scalability advantages over MPI-3.1-based communication when planned transfers are feasible in an MPI application. Their merger into a generalized, persistent collective communication with partitions is a logical next step, with significant advantages for performance portability. Non-trivial decisions about the syntax and semantics of such operations need to be addressed, including scope of knowledge of partitioning choices by members of the communicator's group(s). This paper introduces and motivates proposed interfaces for partitioned collective communication.**

Partitioned collectives will be particularly useful for multithreaded, accelerator-offloaded, and/or hardware-collective-enhanced MPI implementations driving suitable applications, as well as for pipelined collective communication (e.g., partitioned allreduce) with single consumers and producers per MPI process. These operations also provide load imbalance mitigation. Halo exchange codes arising from regular and irregular grid/mesh applications are a key candidate class of applications for this functionality.

Generalizations of lightweight notification procedures `MPI_Parrived` and `MPI_Pready` are considered. Generalization of `MPIX_Pbuf_prepare`, a procedure proposed for MPI-4.1 for point-to-point partitioned communication, are also considered, shown in context of supporting ready-mode send semantics for the operations. The option of providing local and incomplete modes for initialization procedures is mentioned (which could also apply to persistent collective operations); these semantics interact with the `MPIX_Pbuf_prepare` concept and the progress rule.

Last, future work is outlined, indicating prerequisites for formal consideration for the MPI-5 standard.

I. Introduction

MPI collective operations have been among the most widely used operations in the MPI Standard since its inception. They are broadly used by many applications and have highly optimized performance in every major MPI implementation. Collective operations have been recently expanded in MPI 4.0 [20] to include persistent collectives [16], [19], which enable the user to take advantage of the repetition of the same collective operations (new data only) in their applications to allow the MPI library to optimize data movement. Partitioned point-to-point operations [14], [15] were also introduced in MPI-4.0, as a means to leverage partial data buffer availability to initiate data movement as soon as possible. This behavior enables use of the network at an earlier stage than would otherwise be possible waiting for the entire buffer to become available, giving more time/bandwidth to the communication overall. Partitioned point-to-point communication is especially useful in applications that use large amounts of thread-based parallelism or offload/accelerator hardware that needs a lightweight signalling method for indicating that data is available to send or has arrived.

At present, MPI collective operations cannot take advantage of the early availability of portions of a buffer to begin data transmission before the entire buffer is ready. Partitioned point-to-point communication operations, as they are defined in MPI-4.0, utilize persistent-style operation semantics. Therefore, the addition of persistent collective communication operations in MPI 4.0 suggests a logical extension of persistent collectives to support partitioned transfers through extrapolation of the point-to-point capability.

Partitioned collective operations would serve many of the same purposes as point-to-point operations: enhance overlap, eliminate unexpected messages and long receive queues, and match application use-cases, such as halo codes. Partitioned collectives would have the advantage of describing collective operations for such use cases with the added optimizability conveyed by a persistent interface.

In this paper, we explore prototype APIs and semantics to provide partitioned collective communications for MPI. We motivate that the further development of these new operations would be helpful to MPI application performance by providing opportunities for communication optimizations. We re-examine the benefits that a "partial" collective data movement could have in reducing the wait times [11] when all processes must wait until all other processes have arrived at the MPI collective call in order to be able to significantly progress the collective operation. For example, with a partitioned collective, it is our position that entire collective operations could be progressed to near completion even though a limited number of processes have not yet arrived at the collective call. A collective that progressed with potential "holes" in the data set could then be completed faster than waiting for all of the processes to arrive at the beginning

of the operation. In the case where there is a small number of laggard processes, one could imagine using a more costly but simpler solution like a broadcast to deliver the last data to fill any "holes." We conjecture that such a solution could be faster than today's collectives even with the cost of the broadcast operation because the opportunity to distribute the rest of the data before the laggard process arrived at the collective call would outweigh any additional cost of a single higher cost operation in the collective. Early completion of parts of collective operations offers to enhance overlap of communication and computation as well.

The remainder of this paper is organized as follows. First, we provide a motivating example related to all-to-all communication in Section II. As background, we describe how the interfaces for both persistent collectives and partitioned point-to-point work in Section III then we propose how to combine them into partitioned collective communication APIs in Section IV. Section V provides a discussion of partitioning rules. Section VI considers the auxiliary APIs used to manage communication details. We detail some performance optimizations that could be enabled by partitioned collectives in Section VII, and conclude in Section IX.

II. A Motivating Example–Partitoned Alltoall

Many numerical methods rely on all-to-all algorithms. Fast Fourier Transform (FFT) code-bases, such as heFFTe [1], require data exchanges among all processes, with performance bottlenecks arising due to the `MPI_Alltoall` or `MPI_Alltoallv` operations. Similarly, sparse solvers such as algebraic multigrid (AMG) [2] and Krylov subspace methods rely on the performance of boundary exchanges, which can be implemented as `MPI_Neighbor_Alltoallv`-type operations [18], preferably with nonblocking or persistent variations, depending on the specific application.

The underlying performance of all-to-all operations is constrained by inter-node communication costs. This performance can be improved through the use of locality-aware optimizations that aggregate messages on-node to reduce inter-node communication while evenly distributing data across all available CPU cores per node. Locality-aware boundary exchanges achieve significant performance gains, yielding performance improvements of entire algebraic multigrid solvers by 4-8x [5], [6]. Further, heterogeneous nodes achieve large performance benefits when utilizing all available cores during all-to-all operations [7].

Partitioning the all-to-all buffers would allow further optimization opportunities (such as pipelining) of the existing MPI operations. Partitioned communication, as described further below, allows portions of data to be communicated without waiting for all data to become ready. This can have a large impact on boundary exchanges because sparse matrices often have significant load imbalances among rows. Further, the ability of threads to communicate completion of their portion of a send buffer would enable communication to fully saturate available bandwidth without manually splitting data across all CPU cores per node [7]. Finally, partitioned communication complements locality-aware optimizations. Together, these optimizations can enhance load balance between synchronization costs and inter-node communication constraints.

III. Background

The partitioned collective communication operations proposed here are based on the conflation of persistent collective and partitioned point-to-point communication primitives that were both introduced in the MPI-4.0 [20] standard. In this section, we provide a brief overview of these communication primitives and compare their semantic differences to provide a rationale for proposed semantics of partitioned collective communication primitives, which, in this approach, comprise one-for-one supersets of each persistent collective operation (except `MPI_Barrier_init`). This paper focuses on intra-communicator operations; disussion of inter-communicator modes remains for a future communication.

A. Persistent Collective Operations

Persistent collective communication operations enable the MPI programmer to specify collective communication operations with the same argument list that is repeatedly executed within a parallel computation; only the data in buffers is allowed to change from invocation to invocation. Persistent collective operations enable an MPI implementation to select from a polyalgorithmic set of potential strategies a more efficient way to perform the collective operation based on the parameters specified at initialization than might otherwise be possible when confronted with a collective operation. This "planned-transfer" approach can offer significant performance benefits for programs with repetitive communication patterns in which the only change between invocations is the data buffer values [19]. The trade-off is that the operation must be precisely the same; for instance, counts and offsets cannot change from call to call.

For each nonblocking collective and nearest-neighbor (topological) collective operation (as defined in MPI-3.1 [13]), a corresponding initialization procedure was included with an additional info argument These return a new type of `MPI_Request` called the persistent collective request. Existing MPI starting procedures `MPI_Start` and `MPI_Startall` are extended to support the use of these persistent collective requests. Similarly, completion functions such as `MPI_Wait` and its variants and query functions such as `MPI_Test` and its variants are extended to use these persistent collective requests. Inactive persistent collective requests can be freed using the freeing function `MPI_Request_free`.

Unlike persistent point-to-point initialization procedures, persistent collective initialization procedures are non-local, may communicate and may synchronize (though not required) with other MPI processes in the specified communicator. Similarly to blocking collective procedures and non-blocking collective initialization procedures, persistent collective initialization procedures must be called in the same order across different MPI processes associated with a given communicator. However, after initialization, the starting procedures with persistent collective requests could be started in any order across the MPI processes in a given communicator. Lastly, the matching rules for persistent collective operations are similar to that of nonblocking collective operations. That is, persistent collective operations cannot be matched with either blocking or nonblocking collective operations.

B. Partitioned Point-to-Point Operations

Partitioned point-to-point communication was introduced [15] as an alternative to the failed MPI endpoints concept [8]. This concept was based on application developer feedback from a major survey in the US for exascale MPI development that indicated that applications wanted RMA-like performance benefits for multi-threading but with a send-receive-type model [4].

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 send-receive in the 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 to function in the lower performance `MPI_THREAD_MULTIPLE` mode [15], [22].

As of now, partitioned communication libraries have been built for MPI using persistent operations [3], MPI RMA [15] and comparative evaluations between the approaches have been accomplished [10].

The status and progress of persistent communication operations in the MPI 3.1 standard (persistent point-to-point operations) are controlled through functions called on semi-permanent (reusable) request objects. Partitioned point-to-point communications use a similar control methodology. ~~The functions introduced as part of partitioned point-to-point communication API are described briefly below.~~ The syntax and semantics for partitioned communications is based on persistent communication functions present in the MPI 3.1 standard, with extensions. In addition to the functions listed in the MPI 4.0 API, there are minor behaviorial changes for when `MPI_Start` is called on a partitioned request.

`MPI_Psend_init` informs the MPI library to start preparing a partitioned send operation. This function begins the process of setting 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 requirement to match, in general, with its corresponding `MPI_Psend_init`. This function begins to prepare the receive-side to support a partitioned receive.

`MPI_Pready` is a local, nonblocking operation that informs the MPI library that the specified partition is ready to be sent. Even though the partitioned send request must be started for a transfer to occur, individual partitions will not be sent until they are marked as ready with an `MPI_Pready` or its variants. For instance, `MPI_Pready_range` enables the marking of a number of partitions in a request with a partition 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 of a partitioned transfer and if that partition is available for use. It is nonblocking; it returns true before the entire partitioned send request is complete as long as the tested partition is ready at the receiver.

`MPIX_Pbuf_prepare` is a protocol-like operation, proposed for MPI-4.1, that at present is called by both sides of a partitioned send-receive channel. Some controversy remains as of the writing of this paper, and explanations that follow build on recent discussions of the MPI Forum during September and early October of 2021. As currently envisaged, it is used to ensure that receive buffers are indeed ready on the receiver-side when `Pready`s are declared on the sender side. This is currently designed to ensure that optimal offload to threads and, more importantly, GPU/accelerator-offloaded kernels of partition-filling operations avoid complex interactions and overheads. While the exact syntax is debatable, this has the effect of a "ready-to-receive" (RTR) semantic between the end-points of a persistent virtual channel. Regardless of final determination of syntax, this operation is definitely helpful for initial alignment between endpoints in a partitioned send/receive pair, which might not otherwise negotiate their channel properties until the first `Wait/test` operations under weak progress assumptions. However, it is also noted that, with double buffering, the operation could be avoided with appropriate programming, other than the first trip through the protocol. Inclusion of a "ready" assertion that enables `Pbuf_prepare` to become a nop after the first trip through the transfer is also considered when the sender can semantically be assured that the receiver has started, along the lines of `MPI_Rsend`. Double buffering is just one programmatic means by which this assertion could be properly raised.

Finally, it must be noted that is still necessary to call a completing procedure such as `MPI_Wait`, even if a call to `MPI_Parrived` has returned *true* for each partition of the receive buffer. Likewise, the send-side is also obliged to call `MPI_Wait` after it has marked all partitions as ready.

C. Semantic Differences

The semantics of partitioned point-to-point operations are similar to persistent point-to-point operations with the exception of freeing active requests. While freeing an active persistent point-to-point is allowed (even though it is strongly discouraged), it is erroneous to free active requests associated with partitioned point-to-point operations. They also differ in their matching requirements. While blocking and nonblocking point-to-point procedures could be used to match persistent point-to-point primitives, partitioned point-to-point procedures require matching partitioned point-to-point procedures. Further, sender/receiver matching only takes place once in a partitioned point-to-point operation, not per transfer.

The semantics of persistent collective initialization operations are similar to those of partitioned point-to-point initialization operations with the exception that the former are are nonlocal and incomplete. The initialization procedures for partitioned

TABLE I
SEMANTIC DIFFERENCES BETWEEN PERSISTENT POINT-TO-POINT, PARTITIONED POINT-TO-POINT, PERSISTENT COLLECTIVE, AND PARTITIONED COLLECTIVE PROCEDURES (PROPOSED)

Procedure Type	Persistent Point-to-Point	Partitioned Point-to-Point	Persistent Collectives	Partitioned Collectives
Initialization	incomplete & local		incomplete & non-local	
Starting	incomplete & local			
Completing	complete & non-local‡			
Freeing	freeing and local for inactive requests			
	marks request for freeing for active requests	freeing active requests is erroneous		

‡complete & local for requests associated with `MPI_BSEND_INIT` and `MPI_RSEND_INIT`

point-to-point operations are incomplete and local and may return before any non-local procedures have been called[1].

IV. PARTITIONED COLLECTIVE OPERATIONS

As mentioned, partitioned collective communications arise from the application of the partitioning concept to MPI persistent collective operations. In addition to the new collective initialization procedure to leverage partitioned capabilities, partitioned collectives also induce an extension of the `MPI_Pready*` and `MPI_Parrived` procedures. Appropriate analogs for the `MPIX_Pbuf_prepare` concept are also required.

A. API and Semantics

The proposed initialization procedures for partitioned collectives communications are presented in the top part of Figure 1. Each of the sixteen original collective communication with communication buffers is presented with a partitioned initialization procedure except for `MPI_Barrier`, which moves no data.

As with partitioned point-to-point communication, each communication buffer in the collective call required an additional argument to specify its number of partitions. So, for collective communication using a unique argument buffer, such as the broadcast operation, the proposed partitioned initialization procedure has only one extra argument compared to persistent collective initialization procedure. This can be seen on the second procedure listed in Figure 1, `MPIX_Pbcast_init`; its second argument is the number of partitions for the unique buffer argument. For collectives using two buffer arguments, such as the gather operation, the proposed partitioned initialization procedure presents two extra arguments as compared to the persistent initialization procedure: one specifying the number of partition for the input buffer and specifying the number of partitions for the output buffer. This can be observed on the third procedure listed in Figure 1, `MPIX_Pgather_init`. The two partition arguments can have different values. This behavior matches the use of `MPI_Psend_init` and `MPI_Precv_init`, which can match each other even with different values for the number of partitions on each side. Finally, for the collective operations providing multiple sizes for either or both the input and output buffers thanks to displacements and counts arrays, the number of partitions argument is also provided as an array. This allows one to specify different partition numbers for each buffer size involved in the communication. This can be observed on the fourth procedure listed in Figure 1, `MPIX_Pgatherv_init`.

The introduction of partitioning is not merely a rote syntactic extrapolation. The scope of knowledge of what each process has for partitions informs the complexity of the initialization procedures, and the variety of data transfer patterns enabled by each new API. As such, scope of partition sameness and independence is covered in some detail below (for a subset of all the operations). Such decisions are still subject to additional debate as prototype implementations and use cases are assembled in the near future, and the proposed API is fully vetted for all 21 operations (16 "original" and five neighbor). See for instance, Figure 2, which shows two possible partitioning rules for `MPIX_Pscatterv_init`; we discuss below which option is ultimately preferable in the authors' view.

The semantics of the proposed partition collective procedures is the same as for persistent collective procedures, as shown in Table I.

V. PARTITIONING RULES

For a representative subset of the 21 operation supported, we describe their behavior on intra-communicators. Extension to inter-commmunicators is left for future work.

A. One-to-all group communication

Rule: For `MPIX_Pbcast_init`[2], allow different leaf partitions everywhere, and, either,

- all processes know the root partitioning value,
- processes don't know the root partitioning (MPIX_PBCAST_ROOT_PART_UNKNOWN);

Also, we will provide an info key assertion (with operation scope) for when all leaves have the same partitioning value (MPIX_PBCAST_LEAVES_ALL_PART_SAME). This is the highest performance mode.

Proposed Rule for both Pscatter/Pscatterv: The entire incoming buffer is partitioned in each process; the outgoing buffer is partitioned in the corresponding manner (each per-process segment is partitioned). Rationale: the scatter operation is *as if* P send operations at root process to other processes and one receive operation at each process from root process—each send-receive message is partitioned. The result is that this requires calls to `MPI_Pready` for $P \times N$ distinct partitions at the root process and permits calls to

[1]We hope to bridge this gap with local versions of the persistent and partitioned collective operations in MPI-5. The main problem arises from the zero-trip case when index and offset buffers are involved. This is fully covered in the paper by Holmes et al. [19]. The main problem is an API problem of when MPI is able to release resources back to an application in edge cases.

[2]Again, note the use of MPIX_ for prestandard API definitions.

```
int MPIX_Pbcast_init(void *buf, int partitions, MPI_Count count, MPI_Datatype datatype, int root,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pgather_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvparts, MPI_Count recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pgatherv_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int recvparts[], const MPI_Count recvcounts[], const MPI_Aint displs[],
        MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pscatter_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvparts, MPI_Count recvcount, MPI_Datatype recvtype, int root,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pscatterv_init(const void *sendbuf, const int sendparts[], const MPI_Count sendcounts[],
        const MPI_Aint displs[], MPI_Datatype sendtype, void *recvbuf, int recvparts, MPI_Count
        recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pallgather_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvpart, MPI_Count recvcount, MPI_Datatype recvtype, MPI_Comm comm,
        MPI_Info info, MPI_Request *request);
int MPIX_Pallgatherv_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, const int recvparts[], const MPI_Count recvcounts[], const MPI_Aint displs[],
        MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Palltoall_init(const void *sendbuf, int sendparts, MPI_Count sendcount, MPI_Datatype sendtype,
        void *recvbuf, int recvpart, MPI_Count recvcount, MPI_Datatype recvtype, MPI_Comm comm,
        MPI_Info info, MPI_Request *request);
int MPIX_Palltoallv_init(const void *sendbuf, const int sendparts[], const MPI_Count sendcounts[],
        const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvparts[],
        const MPI_Count recvcounts[], const MPI_Aint rdispls[], MPI_Datatype recvtype,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Palltoallw_init(const void *sendbuf, const int sendparts[], const MPI_Count sendcounts[],
        const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvparts[],
        const MPI_Count recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Preduce_init(const void *sendbuf, void *recvbuf, int partitions, MPI_Count count,
        MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pallreduce_init(const void *sendbuf, void *recvbuf, int partitions, MPI_Count count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Preduce_scatter_block_init(const void *sendbuf, void *recvbuf, int partitions, MPI_Count recvcount,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Preduce_scatter_init(const void *sendbuf, void *recvbuf, const int partitions[], const MPI_Count
        recvcounts[], MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pscan_init(const void *sendbuf, void *recvbuf, int partitions, MPI_Count count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pexscan_init(const void *sendbuf, void *recvbuf, int partitions, MPI_Count count,
        MPI_Datatype datatype, MPI_Op op, MPI_Comm comm, MPI_Info info, MPI_Request *request);
```

```
int MPIX_Pneighbor_allgather_init(const void *sendbuf, int sendparts, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, int recvparts, MPI_Count recvcount, MPI_Datatype recvtype,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pneighbor_allgatherv_init(const void *sendbuf, int sendparts, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, const int recvparts[], const MPI_Count recvcounts[],
        const MPI_Aint displs[], MPI_Datatype recvtype, MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pneighbor_alltoall_init(const void *sendbuf, int sendparts, MPI_Count sendcount,
        MPI_Datatype sendtype, void *recvbuf, int recvparts, MPI_Count recvcount, MPI_Datatype recvtype,
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
int MPIX_Pneighbor_alltoallv_init(const void *sendbuf, const int sendparts[], const MPI_Count sendcounts[],
        const MPI_Aint sdispls[], MPI_Datatype sendtype, void *recvbuf, const int recvparts[],
        const MPI_Count recvcounts[], const MPI_Aint rdispls[], MPI_Datatype recvtype, MPI_Comm comm,
        MPI_Info info, MPI_Request *request);
int MPIX_Pneighbor_alltoallw_init(const void *sendbuf, const int sendparts[], const MPI_Count sendcounts[],
        const MPI_Aint sdispls[], const MPI_Datatype sendtypes[], void *recvbuf, const int recvparts[],
        const MPI_Count recvcounts[], const MPI_Aint rdispls[], const MPI_Datatype recvtypes[],
        MPI_Comm comm, MPI_Info info, MPI_Request *request);
```

Fig. 1. Proposed Partitioned Collective and Neighborhood Collective Communication C Language Bindings. The `MPIX_` prefix indicates prestandard. The APIs are all in "large count" form since MPI-4 resolved this issue.

`MPI_Parrived` for N distinct partitions at each process. See Figure 2 (bottom half). Note: We are considering the case where all processes know all partitions in this discussion. Generalizations are also possible to weaken these assumptions.

Receive-buffer readiness: This is non-synchronizing; with offload implementations, a form of ready-mode support may be needed. This remains for future work.

B. All-to-one Communication

Proposed Rule: For `MPIX_Preduce_init`, there is one unique root partition, and one unique non-root partition. If all processes know these partitions, this is the simplest case and most performant case. As with `MPIX_Pbcast_init`, appropriate info arguments will be defined.

For `MPIX_Pgather_init`, we will follow the same scheme as for `MPIX_Pscatter_init`. Here again, considering the v-version, the preferred rule shown in Figure 2 applies. But, the incoming and outgoing partitioning schemes are reversed.

For `MPIX_Preduce_scatter_init`[3], the input buffer partition scheme should follow `MPIX_Preduce_init` while the output buffer should follow the `MPIX_Pscatterv_init`.

For `MPIX_Preduce_scatter_block_init`, the input buffer partition scheme should follow `MPIX_Preduce_init`, while the output buffer should follow `MPIX_Pscatter_init`.

Overall, `MPIX_Pgather_init` and `MPIX_Pgatherv_init` are the "reverse concepts" of `MPIX_Pscatter_init` and `MPIX_Pscatterv_init`.

Receive-buffer readiness: These are non-synchronizing; with offload implementations, a form of ready-mode support may be needed. This remains for future work.

C. Allreduce

Type: One/two buffer symmetric operations.

Rule: For `MPIX_Pallreduce_init`, each buffer specified has the same partitioning across the group of the communicator.

Receive-buffer readiness: This is synchronizing across the group. With offload implementations, a mechanism to ensure that receive buffers are ready can be accomplished as part of startup overhead for the operation. In the large-transfer limit, this should pose negligible overhead.

`MPI_IN_PLACE`: This mode does not appear to make significant impact on the operation under partitioning.

D. Alltoall* Operations

Alltoall-type operations and their neighbor variants have quite simple interpretations of their partition counts because they are simple abstractions on top of bundles of sends and receives.

Partitioning Rules and Specializations: For each send buffer, a unique partitioning scheme may be selected. For each corresponding receive buffer, a) either the same partitioning may be selected, b) or a different partitioning. While that appears to be a tautological explanation, a standardized `MPI_Info` key `MPIX_EACH_PARTITION_PAIR_SAME` can be asserted by the application when condition 'a' applies for all transfers for a given partitioned alltoall-type operation. Two other `MPIX_Info` keys `MPIX_ALL_SEND_PARTITIONS_SAME` or `MPIX_ALL_RECV_PARTITIONS_SAME` may be asserted when there is one unique send-side partition count, or receive-side partition count for all specified transfers. Only two of these three assertions are ever needed.

Receive-buffer readiness: This is (optionally) synchronizing across the group. With offload implementations, a mechanism to ensure that receive buffers are ready can be accomplished as part of startup overhead for the operation. In the large-transfer limit, this should pose negligible overhead.

[3]As has been noted proverbially, there is an anachronism with `MPI_Reduce_scatter` and `MPI_Reduce_scatter_block`. `MPI_Reduce_scatter` should be `MPI_Reduce_scatterv`, and `MPI_Reduce_scatter_block` should be `MPI_Reduce_scatter`. They just got the names wrong in the standard.

VI. Auxiliary APIs and Semantics

This section addresses the APIs that work with buffer completion notification and protocol synchronization.

A. `Pready` and `Parrived` extensions

`MPI_Pready`, `MPI_Pready_range`, `MPI_Pready_list`, and `MPI_Parrived` procedures are part of the partitioned semantics. As recalled in Section III-B, the `MPI_Pready*` procedures are called to tell the MPI library that the specified partition of a send buffer is ready to be sent, and the `MPI_Parrived` procedure is called to test if a specified partition of a receive buffer has arrived.

Though such procedures are enough for point-to-point communications, they cannot provide all the information required by all collective communication operations. For example, an MPI process that is the root of a Gather operation is supposed to receive buffer contributions from all other MPI processes involved in the collective communication. If there are more than two MPI processes involved in the Gather operation, the root MPI process will receive at least two partitioned buffers. Calling `MPI_Parrived` on the request associated with a partitioned Gather operation, the MPI library will not know on which buffer message it is supposed to check for the arrival of the specified partition number.

To address this issue, we propose to extend `MPI_Pready*` and `MPI_Parrived` procedures with a new argument, specifying for which destination the partition is marked as ready (for send partitioned buffers), and for which source the partition has arrived (for receive partitioned buffers). See Figure 3 for details.

B. `MPIX_Pbuf_prepare` generalization

In order to generalize `MPIX_Pbuf_prepare` one must recognize that some situations have uncertainty in the started protocol between receivers of data and senders of data, due to asynchrony of when `MPI_Start` happens on all processes in the communicator's group. This can lead to "unexpected transfers"; our goal here is to avoid any queueing or the complexity of original point-to-point transfers that admit unexpected transfers as normal. To do so, MPI is required, in some instances, to reveal an implicit protocol step between those receiving and those sending, equivalent to a ready-to-receive (RTR). Normally, MPI avoids any prescription of protocol, but performance, particularly when performant offload to simple accelerator devices is the goal, requires avoidance of complex decision-making. Therefore, non-synchronizing collective operations as broadcast may have to have

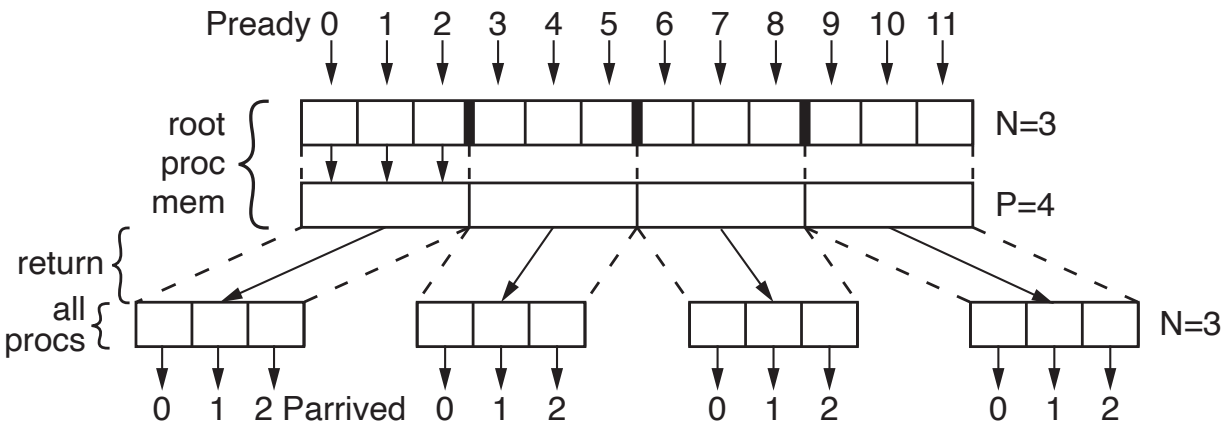

Fig. 2. Two possible partitioning schemes for the partitioned scatter operation. The top half shows the scheme where the entire outgoing buffer is partitioned into N partitions, broken up into P segments that are delivered to P processes, and then each segment is split into N partitions at each process. The bottom half shows the scheme where each of the P segments of the outgoing buffer are partitioned into N partitions, inevitably leading to a 1-to-1 mapping between outgoing partitions at the root and incoming partitions at all processes.

```
int MPIX_Pready_extended(int partition
    , int dest, MPI_Request *request);
int MPIX_Pready_range_extended
    (int partition_low, int partition_high
    , int dest, MPI_Request *request);
int MPIX_Pready_list_extended
    (int length, int array_of_partitions
    [], int dest, MPI_Request  *request);
int MPIX_Parrived_extended
    (MPI_Request *request,
    int partition, int source, int *flag);
```

Fig. 3. Proposed `Pready` and `Parrived` extensions C Language Bindings

reverse communication paths to support their effective partitioned operation. RTRs emanating from leaves will reach parent processes in a given implementation, ultimately freeing the root to begin the actual transfer. Such overheads may be acceptable because of the intent to do large transfers with partitioning.

For collectives that may synchronize, as noted in the MPI supplement [21], partitioned collective communication is free to introduce synchronization internally to effect buffer readiness. Yet, that may be overkill when the semantics of the program, such as red-black ordering between a pair of collectives, can ensure buffer readiness of the next collective to be started. This means that significant study of the extensions of `MPIX_Pbuf_prepare` are still required. Clearly, the use of info arguments provides the user with a means to describe the desired synchronization, but the authors have not yet produced a complete proposal for such semantics.

VII. Optimization Opportunities

Partitioned collective communications, much like partitioned point-to-point, have the possibility of leveraging RDMA network primitives to optimize data movement performance. The planned-transfer nature of the persistent partitioned collectives enable them to negotiate memory buffers in advance of the data transfer taking place and allow for out-of-order delivery of data. In addition to pre-negotiated buffers, partitioned collectives also enable taking advantage of the distribution in arrival times for each MPI process at the collective operation. It is well known that the "noise" in arrival time at MPI collective operations when using many MPI processes can be significant [11], in fact this is a key performance point for partitioned point-to-point operations [15]. Partitioned collectives bring this optimization opportunity to collective operations.

Aggregation of data can be a good optimization for MPI when the quantity of data being handed over to the MPI library for communication can be reasonably predicted. For example, this kind of optimization for RMA operations in a threaded environment leads to good performance results [9], [17]. Partitioned collective operations can provide this same level of optimization as they are aware of the data size and composition in advance of the data movement operation. This is particularly true for operations after the first invocation of the collective.

Since partitioned collective communications serve highly parallel local execution models in their separate execution but shared process space (e.g., OpenMP and Pthreads), one could expect that they would benefit from all of the optimizations available to partitioned point-to-point operations, but there is the exception of message matching. As collectives do not have to match individual data messages in the same way as point-to-point communication operations, they do not suffer from the multi-threaded matching match list distribution problems of typical send operations [23] or any optimizations in that space [12].

VIII. Future Work

Several categories of future work remain. These include the following for the 21 operations discussed above:

- extend the partitioning rules to all 16+5 operations shown in Figure 1 (including defining all the info keys appropriate for each special case of partition information scope within the communicator's group)
- extend the proposal to include inter-communicators
- formalize the extensions needed for `MPIX_Pbuf_prepare` for each operation.

Further, it is important to establish additional connection to application and algorithmic use cases for each of the partitioned operations, apart from symmetry by their complete inclusion.

We note the lack of `MPIX_Pscatterw` and variants thereof in MPI-4. But, a partitioned extension of `MPIX_Pscatterv_init`, namely `MPIX_Pscatterw_init`, appears useful and straightforward to add from on-going discussions. Whether this would first require the antecedent operations (blocking, nonblocking, and persistent) to be added to the standard is unclear.

Also, additional to the 21 operations shown in Figure 1, there is the potential to introduce a concept of a partitioned barrier, but this is an academic discussion at present, given the lack of any data to partition in a barrier. Multiple potential meanings for such an operation have been advanced, but it is a far future topic.

Further opportunities for employing partitioning and/or persistence also appear across additional chapters of the MPI-4 standard. For instance, there is a clear opportunity to introduce persistent operations into the I/O chapter; this immediately leads, for collective I/O to consider partitioned writes and reads. Such operations merit further exploration. Overlap of I/O with communication and computation appears to be a fertile area for exploration in MPI-5.

It has also been pointed out in working-group discussions that persistence communication modes may be extremely useful for one-sided (RMA) in some instances; therefore, partitioning may also prove of interest.

IX. Conclusion

This paper introduced the authors' current proposal to the MPI Forum—under on-going development—for partitioned collective communication, which has the potential to be adopted in MPI-5. These operations would comprise a superset of MPI-4 persistent collective operations, and partitioned point-to-point communication. They offer the benefits of both of these earlier enhanced APIs for programs that have temporal locality in communication (i.e., capable of using planned transfer). Many bulk-synchronous-parallel programs, and other programs with static communication patterns, can arguably benefit from these operations, such as halo codes that do load-balancing only occasionally. Partitioned collectives mesh well with emerging architectural features of accelerator offload, network-based collectives, and multithreaded MPI programs. Proposed operations' prototypes were given in the C interface. Potential extensions beyond baseline features were mentioned.

We note that significant additional study and debate remains pertinent regarding partition counts and constraints for the various partitioned collective operations. Further, careful study and debate of `MPIX_Pbuf_prepare` remains for partitioned point-to-point, as well as its extension offered here for collective operations. Finally, a key difference between the `_init` semantics of partitioned point to point, and persistent collectives means that partitioned collectives are initially more restrictive (from the user's point-of-view) in terms of when they return control to the application.

X. Acknowledgements

This work builds on presentations and discussions made from time to time over the past three plus years in the MPI Forum's Persistent/Collective/Partitioned Working Group. This work also builds on recent discussions of the HACC Working Group; particular acknowledgment to James Dinan (NVIDIA) is given for his contributions in the working groups and recent Forum discussions.

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

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] Alan Ayala, Stanimire Tomov, Azzam Haidar, and Jack Dongarra. heffte: Highly efficient fft for exascale. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 262–275, Cham, 2020. Springer International Publishing.

[2] Allison H. Baker, Martin Schulz, and Ulrike M. Yang. On the performance of an algebraic multigrid solver on multicore clusters. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, VECPAR'10, page 102–115, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Purushotham V Bangalore, Andrew Worley, Derek Schafer, Ryan E Grant, Anthony Skjellum, and Sheikh Ghafoor. A portable implementation of partitioned point-to-point communication primitives. In *Proceedings of the European MPI Users' Group Meeting (EuroMPI 2020)*, 2020.

[4] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Venkata, Ryan E. Grant, Thomas Naughton, Howard Pritchard, and Geoffroy Vallee. A survey of MPI usage in the U.S. Exascale Computing Project. *Concurrency and Computation: Practice and Experience*, 32(3):e4851, 2020. DOI: 10.1002/cpe.4851.

[5] A. Bienz, L. Olson, and W. Gropp. Node aware sparse matrix-vector multiplication. *Journal of Parallel and Distributed Computing*, 130:166 – 178, 2019.

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

[7] Amanda Bienz, William D. Gropp Luke N. Olson, and Shelby Lockhart. Modeling data movement performance on heterogeneous architectures. In *(to appear) Proceedings of the 2021 IEEE High Performance Extreme Computing Conference, Boston, MA, September 20-24, 2021*, 2021.

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

[9] Matthew GF Dosanjh, Taylor Groves, Ryan E Grant, Ron Brightwell, and Patrick G Bridges. Rma-mt: a benchmark suite for assessing mpi multi-threaded rma performance. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 550–559, New York, NY, USA, 2016. IEEE, IEEE.

[10] Matthew GF Dosanjh, Andrew Worley, Derek Schafer, Prema Soundararajan, Sheikh Ghafoor, Anthony Skjellum, Purushotham V Bangalore, and Ryan E Grant. Implementation and evaluation of mpi 4.0 partitioned communication libraries. *Parallel Computing*, page 102827, 2021.

[11] Kurt B Ferreira, Patrick Bridges, and Ron Brightwell. Characterizing application sensitivity to os interference using kernel-level noise injection. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[12] Mario Flajslik, James Dinan, and Keith D Underwood. Mitigating mpi message matching misery. In *International conference on high performance computing*, pages 281–299, New York, NY, USA, 2016. Springer, Springer.

[13] Message Passing Interface Forum. Mpi: A message-passing interface standard. version 3.1. Technical report, Univ. of Tennessee, Knoxville, TN, USA, 2015.

[14] Ryan Grant, Anthony Skjellum, and Purushotham V Bangalore. Lightweight threading with mpi using persistent communications semantics. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2015.

[15] Ryan E. Grant, Matthew G. F. Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded MPI communication. In Michèle Weiland, Guido Juckeland, Carsten Trinitis, and Ponnuswamy Sadayappan, editors, *High Performance Computing - 34th International Conference, ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proceedings*, volume 11501 of *Lecture Notes in Computer Science*, pages 330–350, Frankfurt, Germany, 2019. Springer.

[16] Masayuki Hatanaka, Masamichi Takagi, Atsushi Hori, and Yutaka Ishikawa. Offloaded mpi persistent collectives using persistent generalized request interface. In *Proceedings of the 24th European MPI Users' Group Meeting*, pages 1–10, New York, NY, USA, 2017. Springer.

[17] Nathan Hjelm, Matthew G. F. Dosanjh, Ryan E. Grant, Taylor Groves, Patrick Bridges, and Dorian Arnold. Improving mpi multi-threaded rma communication performance. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, New York, NY, USA, 2018. Association for Computing Machinery.

[18] Torsten Hoefler and Jesper Larsson Traff. Sparse collective operations for MPI. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–8, 2009.

[19] Daniel J. Holmes, Bradley Morgan, Anthony Skjellum, Purushotham V. Bangalore, and Srinivas Sridharan. Planning for performance: Enhancing achievable performance for MPI through persistent collective operations. *Parallel Computing*, 81:32–57, 2019.

[20] MPI Forum. MPI: A Message-Passing Interface 4.0 Standard. Technical report, Univ. of Tennessee, Knoxville, TN, USA, 2020.

[21] MPI Forum. Summary of the Semantics of all Operation-Related MPI Procedures. Technical report, Univ. of Tennessee, Knoxville, TN, USA, 2021.

[22] Whit Schonbein, Matthew G. F. Dosanjh, Ryan E. Grant, and Patrick G. Bridges. Measuring multithreaded message matching misery. In Marco Aldinucci, Luca Padovani, and Massimo Torquati, editors, *Euro-Par 2018: Parallel Processing - 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings*, volume 11014 of *Lecture Notes in Computer Science*, pages 480–491, Turin, Italy, 2018. Springer.

[23] Whit Schonbein, Matthew GF Dosanjh, Ryan E Grant, and Patrick G Bridges. Measuring multithreaded message matching misery. In *European Conference on Parallel Processing*, pages 480–491, New York, NY, USA, 2018. Springer, Springer.