D R A F T

Document for a Standard Message-Passing Interface

Message Passing Interface Forum

May 28, 2019

This work was supported in part by NSF and ARPA under NSF contract CDA-9115428 and Esprit under project HPC Standards (21111).

This is the result of a LaTeX run of a draft of a single chapter of the MPIF Final Report document.

Chapter 4

Partitioned Point-to-Point Communication

4.1 Introduction

Partitioned communication modes extend the behavior of persistent point-to-point communication as defined in Chapter 3. Unlike persistent point-to-point communication, partitioned communication uses a non-local initialization call. Partitioned communication is "Partitioned" because it allows for multiple contributions of data to be made from potentially multiple threads to a single communication operation.

> *Advice to users.*
>
> The techniques of Partitioned Communication were known as "finepoints" before their adoption into the MPI standard. The original design goals, functioning, initial implementation and performance improvements were described in detail in a set of research papers that we refer the interested reader to [2, 1].
>
> (*End of advice to users.*)

For such operations, request objects have extended transfer semantics, with a persistent communication style start.../test.../wait sequence, but allows additional partitioned test operations that can expose partial completion of the operation.

Partitioned communication is different in two fundamental ways from persistent point-to-point operations in MPI. First, partitioned communication performs all of the initialization required to enable data transfer between two points in its initialization phase. This means that matching and buffer setup can occur in the MPI_P*_init calls, effectively creating a communication "full channel" between two MPI processes. Second, partitioned communication allows for multiple MPI_Pready calls (contributions) for a single message.

4.2 Semantics of Partitioned Point-to-Point Communication

A valid MPI implementation guarantees certain general properties of partitioned point-to-point communication progress within point-to-point communication, which are described in this section.

Persistent communications use opaque **request** objects to identify communication operations and match the operation that initiates the communication with the operation that

terminates it as described in Section 3. Partitioned communication uses these same semantics.

Partitioned communication provides fine-grained transfers on either or both sides of a send-receive operation described by requests. Persistent communication semantics are ideal for partitioned communication; they provide MPI_Psend_init and MPI_Precv_init functions that allow partitioned communication setup to occur prior to message transfers. This means that partitioned communication initialization functions are non-local; unlike persistent communications. The partitioned communication initialization includes inputs on the number of requested partitions on the send and receiver side. Once a MPI_Psend_init call has been made, the user may begin the operation with a call to MPI_Start and complete the operation with multiple MPI_Pready calls equal to the requested number of send partitions followed by a MPI_Wait/Test call. A MPI_Pready call notifies the MPI library that a portion of the data buffer is ready to be sent. Notification of completion on the receiver-side can be done via fine-grained MPI_Ptest calls before a final MPI_Test/MPI_Wait on the request itself, representing overall message completion. Likewise for send-side operations, notification of completion can be done via MPI_Test/MPI_Wait. A full set of methodologies for starting and completing partitioned communication is given in the following sections.

> *Advice to users.* Having a large number of receiver-side partitions can increase overheads as the completion mechanism needs to work with finer-grained notifications. Using a small number of receive-side partitions *may*provide higher performance.
>
> A large number of send-side partitions may be aggregated by an MPI implementation, making performance concerns of a large number of partitions less potentially impactful than receiver-side granularity. (*End of advice to users.*)

> *Advice to implementors.* It is expected that an MPI implementation will provide best-effort to aggregate data transfers for the requested partition counts on the send-side and receiver-side to allow optimization for different hardware. In some cases it may be desirable to resize the data transfers of the partitions to combine partitions in fractional partition sizes (e.g. 2.5 partitions in a single data transfer). (*End of advice to implementors.*)

First, we provide an example of a simple partitioned transfer, in which send-side and receive-side partitioning is identical.

```
#include "mpi.h"
#define PARTITIONS 8
#define PARTLENGTH 16
int main( int argc, char *argv[]) /* same send/recv partitioning */
{
  double message[PARTITIONS*sizeof(MPI_DOUBLE*5)];
  int partitions = PARTITIONS,
      partlength = PARTLENGTH, stride = PARTLENGTH;
  int myrank, i;
  int count, dest, tag = 1;
  int source = 0;
  MPI_Request request;
```

```
  MPI_Datatype xfer_type;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  MPI_Type_contiguous(5,MPI_DOUBLE,&xfer_type);
  MPI_Type_commit(&xfer_type);
  if (myrank == 0)    /* code for process zero */
  {
     MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
               MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     for(i = 0; i < partitions; ++i)
     {
        /* compute and fill partition #i, then mark ready: */
        MPI_Pready(request, i);
     }
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  else if (myrank == 1)  /* code for process one */
  {
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
               MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  MPI_Finalize();
  return 0;
}
```

Advice to users. In practical uses of this communication infrastructure, a higher level of send-side partitioning as compared to receive-side partitioning is likely to be more efficient, based on an MPI implementation's overhead in managing receive-side partitions. (*End of advice to users.*)

Rationale. Partitioned Communication is designed to provide opportunities for MPI to optimize data transfer. MPI is free to choose how many transfers to do within a partitioned communication send independent of how many partitions are reported as ready to MPI through MPI_Pready calls. Aggregation is permitted but not required. Ordering is permitted but not required. A simple implementation–that is fully valid—can simply wait for the entire message buffer to be marked ready before any underlying transfers occur on the network. However, this modality of communication gives MPI far more flexibility. (*End of rationale.*)

4.2.1 Communication Initiation under Partitioning

Communication initiation for partitioning uses persistent-like APIs to initiate the message transfer (using MPI_Start/MPI_Startall). Predefined Datatypes may be used with partitioned communication. For such communications, invoking the operation (resp, doing the

start) **does not** initiate transfer of any part of the user buffer but may negotiate any required protocols and may optionally do other handshaking. Upon communication initiation, individual partitions of a message may be indicated as ready to be transferred by MPI via MPI_Pready calls.

MPI_PSEND_INIT(buf, partitions, count, datatype, dest, tag, comm, request)

IN	buf	initial address of send buffer (choice)
IN	partitions	number of partitions (non-negative integer)
IN	count	number of elements sent (non-negative integer)
IN	datatype	type of each element (handle)
IN	dest	rank of destination (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Psend_init(const void* buf, int partitions, int count,
              MPI_Datatype datatype, int dest, int tag, MPI_Comm comm,
              MPI_Request *request)

MPI_Psend_init(buf, partitions, count, datatype, dest, tag, comm, request,
              ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_PSEND_INIT(BUF, PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST,
              IERROR)
    <type> BUF(*)
    INTEGER PARTITIONS, COUNT, DATATYPE, DEST, TAG, COMM, REQUEST, IERROR
```

Creates a persistent partitioned communication request and binds to it all the arguments of a partitioned send operation. This call is required to be matched with the following MPI_Precv_init call and is non-local.

MPI_PRECV_INIT(buf, partitions, count, datatype, source, tag, comm, request)

IN	buf	initial address of receive buffer (choice)
IN	partitions	number of partitions (non-negative integer)
IN	count	number of elements to be received (non-negative integer)
IN	datatype	type of each element (handle)
IN	source	rank of source (integer)
IN	tag	message tag (integer)
IN	comm	communicator (handle)
OUT	request	communication request (handle)

```
int MPI_Precv_init(const void* buf, int partitions, int count,
             MPI_Datatype datatype, int source, int tag, MPI_Comm comm,
             MPI_Request *request)
```

```
MPI_Precv_init(buf, partitions, count, datatype, source, tag, comm,
             request, ierror)
    TYPE(*), DIMENSION(..), INTENT(IN), ASYNCHRONOUS ::  buf
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_PRECV_INIT(BUF, PARTITIONS, COUNT, DATATYPE, SOURCE, TAG, COMM,
             REQUEST, IERROR)
    <type> BUF(*)
    INTEGER PARTITIONS, COUNT, DATATYPE,SOURCE, TAG, COMM, REQUEST, IERROR
```

Creates a persistent communication receive request and binds to it all the arguments of a partitioned receive operation. This call is required to be matched with a MPI_Psend_init call.

MPI_PREADY(request, partition)

IN	request	partitioned communication request (handle)
IN	partition	partition to mark ready for transfer (integer)

```
int MPI_Pready(MPI_Request request, int partition)
```

```
MPI_Pready(request, partition, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    TYPE(INTEGER), INTENT(IN) ::  partition
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_PREADY(REQUEST, PARTITION, IERROR)
    INTEGER PARTITION, REQUEST, IERROR
```

MPI_Pready indicates that a given partition is ready to be transferred. The valid partitioning is defined when the persistent operation is defined. In both cases, specifying a partition number that is larger than the number of partitions declared in the P*_init call

initializing the operation is erroneous. MPI_Pready should only be called once per partition for each complete inactive-active-complete cycle in the persistent operation.

MPI_PREADY_RANGE(request, partition_low, partition_high)

IN	request	partitioned communication request (handle)
IN	partition_low	partition to mark lowest partition ready for transfer (integer)
IN	partition_high	partition to mark highest partition ready for transfer (integer)

```
int MPI_Pready_range(MPI_Request request, int partition_low,
              int partition_high)

MPI_Pready_range(request, partition_low, partition_high, ierror)
    TYPE(MPI_Request), INTENT(IN) ::  request
    TYPE(INTEGER), INTENT(IN) ::  partition_low, partition_high
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_PSTART_RANGE(REQUEST, PARTITION_LOW, PARTITION_HIGH, IERROR)
    INTEGER PARTITION_LOW, PARTITION_HIGH, REQUEST, IERROR
```

Partitions marked with MPI_Pready_range can be transferred in any order, in groups, or all at once, there is no ordering implied by MPI_Pready_range. That is completely at the discretion of the MPI implementation.

MPI_PREADY_LIST(count, request, array_of_partitions)

IN	count	list length (non-negative integer)
IN	request	partitioned communication request (handle)
IN	array_of_partitions	array of non-negative integers

```
int MPI_Pready_list(int count, MPI_Request *request, int partition_list[])

MPI_Pready_list(count, request, array_of_partitions, ierror)
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    TYPE(INTEGER), INTENT(IN) ::  partition_list(count)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_PREADY_LIST(REQUEST, PARTITION_LIST, IERROR)
    INTEGER PARTITION_LIST(*), REQUEST, IERROR
```

Partitions marked with MPI_Pready_list can be transferred in any order, in groups, or all at once; there is no ordering implied by MPI_Pready_list. That is completely at the discretion of the MPI implementation.

4.2.2 Communication Completion under Partitioning

The functions MPI_WAIT and MPI_TEST (and relatives) are used to complete a partitioned communication operation. The completion of a send-type partition operation indicates that the sender is now free to call MPI_Start to restart the operation and subsequent MPI_Pready for locations in the send buffer (the send operation itself leaves the content of the send buffer unchanged). It does not indicate that the partitions of the message have all been received; rather, it may have been buffered by the communication subsystem.

The completion of a receive partition operation through MPI_Wait or MPI_Test indicates that the receive buffer contains all of the partitions. Functions for probing the partial completion of the receive buffer are provided with the MPI_Ptest family of functions. A MPI_Ptest call can be used to determine if the receive buffer contains the received partition; the receiver is now free to access this partition (as well as any others that previously completed for that operation) and that the status object is set for this partition. It does not indicate that matching send operation(s) needed to fulfill this partition has/have completed (but indicates, of course, that such send or sends were initiated). Note that the send and receive partitioning are never required to be the same, so there is no guarantee of one-to-one correspondence between received partitions and sent partitions. There is no guarantee of the number or order of transfers implementing the underlying transport, regardless of send-side partitioning; a valid MPI can aggregate partitions as it sees fit.

MPI_PTEST(request, flag, partition, status)

INOUT	request	communication request (handle)
IN	partition	partition received (integer)
OUT	flag	true if operation completed (logical)
OUT	status	status object (Status)

```
int MPI_Ptest(MPI_Request *request, int *partition, int *flag,
              MPI_Status *status)
```

```
MPI_Ptest(request, partition, flag, status, ierror)
    TYPE(MPI_Request), INTENT(INOUT) ::  request
    TYPE(INTEGER), INTENT(INOUT) ::  partition
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_TEST(REQUEST, PARTITION, FLAG, STATUS, IERROR)
    LOGICAL FLAG
    INTEGER PARTITION, REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

A call to MPI_PTEST returns flag = true if the operation identified by request for the specified partition is complete. In such a case, the status object is set to contain information on the completed operation.

The return status object for a receive operation carries information that can be accessed as described in Section 3.2.5.

The request is persistent and therefore is not marked as complete/inactive (resp, deallo-

cated and set to NULL) by this operation. A subsequent MPI_TEST/MPI_WAIT operation is required to do the normal completion of the whole message, as described in Chapter 3.

For symmetry with MPI_TEST, one is allowed to call MPI_PTEST with a null or inactive request argument. One is also allowed to call the test for a partition that has already been completed without subsequently being restarted. In either case, the operation returns with flag = true and empty status.

The function MPI_PTEST can be used to test completion of both sends and receive partitioned operations.

> *Advice to users.* The use of the nonblocking MPI_PTEST call allows the user to schedule alternative activities within a single thread of execution. An event-driven thread scheduler can be emulated with periodic calls to MPI_PTEST. (*End of advice to users.*)

4.2.3 Semantics of Nonblocking Communications in Partitioned Mode

The semantics of nonblocking partitioned communication is defined by suitably extending the definitions in Section 3.5.

Interpretation of count and datatype for partitioned communication Partitioned communication uses the count and datatype arguments in the partitioned communication initialization functions to describe a single partition. The argument partitions specifies how many partitions of a number (count) of datatypes make up the entire buffer to be transferred in the partitioned communication. Once a partitioned transfer starts, at least one transfer of the entire buffer must occur between the completion of the send-side and the completion of the receive-side. However, because it is permitted to use wait/start over and over again prior to completing a single send or single receive, arbitrary amounts of data can be transferred through a single partitioned transfer.

Order Partitioned communication operations *on the overall message* are ordered according to the execution order of the calls that initiate the communication. Overall, messages do not overtake. The non-overtaking requirement of Section 3.5 is extended to nonblocking communication, with this definition of order being used. However, partitions of a single message are allowed to be *overtaking*. They may arrive out-of-order by default.

4.2.4 Multiple Partitioned Completions

It is convenient to be able to wait for the completion of any, some, or all the operations in a list (such as multiple partitions on a single request or multiple requests with corresponding partitions). A call to MPI_PTESTANY can be used to wait for the completion of one out of several partitions of a single request. A call to MPI_PTESTALL can be used to test for all pending operations in a list of requests and partitions. A call to MPI_PTESTSOME can be used to test for completion of all enabled operations in a list of requests and partitions.

MPI_PTESTANY(count, array_of_requests, index, partition, flag, status)

IN	count	list length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
OUT	index	index of operation that completed, or MPI_UNDEFINED if none completed (integer)
INOUT	partition	partition of operation that completed or MPI_UNDEFINED if none completed (integer)
OUT	flag	true if one of the operations is complete (logical)
OUT	status	status object (Status)

```
int MPI_Ptestany(int count, MPI_Request array_of_requests[], int *index,
             int *partition, int *flag, MPI_Status *status)

MPI_Ptestany(count, array_of_requests, index, partition, flag, status,
             ierror)
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(count)
    INTEGER, INTENT(OUT) ::  index
    TYPE(MPI_COUNT), INTENT(INOUT) ::  partition
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Status) ::  status
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_PTESTANY(COUNT, ARRAY_OF_REQUESTS, INDEX, PARTITION, FLAG, STATUS,
             IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), INDEX, PARTITION,
    STATUS(MPI_STATUS_SIZE),
        IERROR
```

This operation tests for completion of either one or none of the partitioned operations associated with active handles. In the former case, it returns flag = true, returns in index the index of this request in the array, and in partition the partition of the partitioned operation completed in the indexed request, and returns in status the status of that operation. If the request is an active persistent request, it is marked not marked as inactive; if the request is ephemeral, it is not deallocated nor is it set to MPI_REQUEST_NULL. A subsequent MPI_TEST/MPI_WAIT operation is required to do the normal completion of the whole message, as described in Chapter 3.

In the latter case (no operation completed), it returns flag = false, returns a value of MPI_UNDEFINED in both index and partition and status is undefined.

The array is indexed from zero in C, and from one in Fortran. The array may contain null or inactive handles. If the array contains no active handles then the call returns immediately with flag = true, index = partition = MPI_UNDEFINED, and an empty status.

If the array of requests contains active handles then the execution of MPI_PTESTANY(count, array_of_requests, index, partition, status) has the same effect as the execution of MPI_PTEST(&array_of_requests[i], flag, partition, status), for i=0, 1 ,..., count-1 in some arbitrary order, until one call returns flag = true, or all fail. In the former

case, index is set to the last value of i and partition is set to the partition of the operation that completed, and in the latter case, both index and partition are set to MPI_UNDEFINED. MPI_PTESTANY with an array containing one active entry is equivalent to MPI_PTEST.

MPI_PTESTALL(count, array_of_requests, array_of_partitions, flag, array_of_statuses)

IN	count	lists length (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
INOUT	array_of_partitions	array of the partitions (array of integers)
OUT	flag	(logical)
OUT	array_of_statuses	array of status objects (array of Status)

```
int MPI_Ptestall(int count, MPI_Request array_of_requests[],
             int array_of_partitions[], int *flag,
             MPI_Status array_of_statuses[])
```

```
MPI_Ptestall(count, array_of_requests, array_of_partitions[], flag,
             array_of_statuses, ierror)
    INTEGER, INTENT(IN) ::  count
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(count)
    TYPE(INTEGER), INTENT(IN)::  array_of_partitions(count)
    LOGICAL, INTENT(OUT) ::  flag
    TYPE(MPI_Status) ::  array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror
```

```
MPI_TESTALL(COUNT, ARRAY_OF_REQUESTS, ARRAY_OF_PARTITIONS, FLAG,
             ARRAY_OF_STATUSES, IERROR)
    LOGICAL FLAG
    INTEGER COUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_PARTITIONS(*),
        ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

This operation returns flag = true if all communications associated with active handles in the array have completed (this includes the case where no handle in the list is active). In this case, each status entry that corresponds to an active request is set to the status of the corresponding operation. Active persistent requests are **not** marked inactive. Requests of any other type are **not** deallocated nor are the corresponding handles in the array set to MPI_REQUEST_NULL. Each status entry that corresponds to a null or inactive handle is set to empty.

Otherwise, flag = false is returned and the values of the status entries are undefined. This is a local operation.

When one or more of the communications completed by a call toMPI_PTESTALL fail, it is desirable to return specific information on each communication. The function MPI_PTESTALL will return in such case the error code MPI_ERR_IN_STATUS and will set the error field of each status to a specific error code. This code will be MPI_SUCCESS, if the specific communication completed; it will be another specific error code, if it failed; or it can be MPI_ERR_PENDING if it has neither failed nor completed. The function MPI_PTESTALL will return MPI_SUCCESS if no request had an error, or will return another error code if it

failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

Rationale. This design streamlines error handling in the application. The application code need only test the (single) function result to determine if an error has occurred. It needs to check each individual status only when an error occurred. (*End of rationale.*)

MPI_PTESTSOME(incount, array_of_requests, array_of_partitions, outcount, array_of_indices, array_of_statuses)

IN	incount	length of array_of_requests (non-negative integer)
INOUT	array_of_requests	array of requests (array of handles)
INOUT	array_of_partitions	array of the partitions (array of integers)
OUT	outcount	number of completed requests (integer)
OUT	array_of_indices	array of indices of operations that completed (array of integers)
OUT	array_of_statuses	array of status objects for operations that completed (array of Status)

```
int MPI_Ptestsome(int incount, MPI_Request array_of_requests[],
             int array_of_partitions[], int *outcount,
             int array_of_indices[], MPI_Status array_of_statuses[])

MPI_Ptestsome(incount, array_of_requests, array_of_partitions, outcount,
             array_of_indices, array_of_statuses, ierror)
    INTEGER, INTENT(IN) ::  incount
    TYPE(MPI_Request), INTENT(INOUT) ::  array_of_requests(incount)
    TYPE(INTEGER), INTENT(IN) ::  array_of_partitions(incount)
    INTEGER, INTENT(OUT) ::  outcount, array_of_indices(*)
    TYPE(MPI_Status) ::  array_of_statuses(*)
    INTEGER, OPTIONAL, INTENT(OUT) ::  ierror

MPI_PTESTSOME(INCOUNT, ARRAY_OF_REQUESTS, ARRAY_OF_PARTITIONS, OUTCOUNT,
             ARRAY_OF_INDICES, ARRAY_OF_STATUSES, IERROR)
    INTEGER INCOUNT, ARRAY_OF_REQUESTS(*), ARRAY_OF_PARTITIONS(*),
    OUTCOUNT, ARRAY_OF_INDICES(*),
        ARRAY_OF_STATUSES(MPI_STATUS_SIZE,*), IERROR
```

This operation does not block and returns the associated requests with active handles in the list have completed for the corresponding partition. It returns in outcount the number of requests from the list array_of_requests (with corresponding array_of_partitions) that have completed. If no operation has completed it returns outcount = 0. If there is no active handle in the list it returns outcount = MPI_UNDEFINED. The function returns in the first outcount locations of the array array_of_indices the indices of these operations (index within the array array_of_requests; the array is indexed from zero in C and from one in Fortran). Returns in the first outcount locations of the array array_of_status the status for these completed operations.

Completed active persistent requests are marked **not** as inactive; any other type or request that completed is **not** deallocated, nor is the associated handle is set to MPI_REQUEST_NULL. A subsequent MPI_TEST/MPI_WAIT operation is required to do the normal completion of the whole message, as described in Chapter 3.

If the list contains no active handles, then the call returns immediately with outcount = MPI_UNDEFINED.

When one or more of the communications completed by MPI_PTESTSOME fails, then it is desirable to return specific information on each partitioned communication. The arguments outcount, array_of_indices and array_of_statuses will be adjusted to indicate completion of all communications that have succeeded or failed. The call will return the error code MPI_ERR_IN_STATUS and the error field of each status returned will be set to indicate success or to indicate the specific error that occurred. The call will return MPI_SUCCESS if no request resulted in an error, and will return another error code if it failed for other reasons (such as invalid arguments). In such cases, it will not update the error fields of the statuses.

4.3 Partitioned Communication Examples

This section provides concrete examples of the utility of partitioned communication in realistic settings. n

4.3.1 Partition Communication with Channels

The equal partitioning on send-side and receive-side example shown on 3 is shown below using non-equal partitioning. In this case, the receive-side uses a single partition, while the sender uses eight partitions. Note that the MPI_Psend_init and MPI_Precv_init functions match each other like in the previous example.

```
#include "mpi.h"
#define PARTITIONS 8
#define PARTLENGTH 16
int main( int argc, char *argv[]) /* same send/recv partitioning */
{
  double message[PARTITIONS*PARTLENGTH];
  int partitions = PARTITIONS,
      partlength = PARTLENGTH, stride = PARTLENGTH;
  int myrank, i;
  int count, dest, tag = 1;
  int source = 0;
  MPI_Request request;
  MPI_Info info = MPI_INFO_NULL;
  MPI_Datatype xfer_type;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  MPI_Type_contiguous(partlength*partitions, MPI_DOUBLE, &xfer_type);
  MPI_Type_commit(&xfer_type);
  if (myrank == 0)    /* code for process zero */
  {
```

```
     MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
          info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);

#pragma omp parallel for {
        /* compute and fill partition #i, then mark ready: */
     MPI_Pready(request, omp_thread_num());
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  }
  else if (myrank == 1)  /* code for process one */
  {
     MPI_Precv_init(message, partitions, count, xfer_type, source, tag,
           info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
  MPI_Finalize();
  return 0;
}
```

4.3.2 Send-only Partitioning Example with tasks and OpenMP

The previous example is tailored specifically for send-side partitioning using threads. This is an example where parallel task producers produce input to part of an overall buffer; they complete in any order and contribute to the overall buffer.

```
#include "mpi.h"
#define PARTITIONS 8
#define PARTLENGTH 16
#define MESSAGE_LENGTH PARTITIONS*PARTLENGTH
int main( int argc, char *argv[]) /* send-side partitioning */
{
  double message[MESSAGE_LENGTH];
  int send_partitions = PARTITIONS,
      send_partlength = PARTLENGTH, send_stride = PARTLENGTH;
  int myrank, i;
  int count, dest, tag = 1;
  int source = 0;
  MPI_Request request;
  MPI_Info info = MPI_INFO_NULL;
  MPI_Datatype send_type;
  MPI_Init( &argc, &argv );
  MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
  MPI_Type_contiguous(partlength*partitions, MPI_DOUBLE, &xfer_type);
  MPI_Type_commit(&xfer_type);
```

```
  if (myrank == 0)    /* code for process zero */
  {
     MPI_Psend_init(message, partitions, count, xfer_type, dest, tag,
               info, MPI_COMM_WORLD, &request);
     MPI_Start(&request);

     #pragma omp parallel {
       #pragma omp task
        {
           /* compute and fill partition #i, then mark ready: */
           /* buffer is filled in arbitrary order from each task */
           MPI_Pready(request, omp_thread_num());
        }
     }
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  else if (myrank == 1)  /* code for process one */
  {
     MPI_Precv_init(message, partitions, count, xfer_type,
               source, tag, info, MPI_COMM_WORLD, &request);

     MPI_Start(&request);
     MPI_Wait(&request, MPI_STATUS_IGNORE);
  }
  MPI_Request_free(&request);
  MPI_Finalize();
  return 0;
}
```

Bibliography

[1] Ryan E. Grant, Matthew GF Dosanjh, Michael J. Levenhagen, Ron Brightwell, and Anthony Skjellum. Finepoints: Partitioned multithreaded mpi communication. In *ISC High Performance Conference (ISC)*, 2019. 4.1

[2] Ryan E. Grant, Anthony Skjellum, and Purushotham V. Bangalore. Lightweight threading with mpi using persistent communications semantics. In *Workshop on Exascale MPI (ExaMPI)*. Held in conjunction with the 2015 International Conference for High Performance Computing, Networking, Storage and Analysis (SC15), 2015. 4.1

Index