Chapter 13

Queued Communication

13.1 Introduction

MPI programs may need to carefully order communication operations with respect to each other, including specifying when communication operations on different communicators may be initiated and completed concurrently, and when operations may be initiated or completed concurrently. In addition, MPI programs may need to order communication operations with respect to each other and computation in asynchronous programming systems that have specialized execution models. This is particularly important because such systems, including GPU streams, graph programming systems, fine-grain threading models, and asynchronous many-task systems, may have execution semantics that efficiently support only limited computational capabilities; for example, *blocking* operations are often disallowed or strongly discouraged in these systems.

This chapter describes a general MPI API for carefully ordering MPI communication operations with respect to each other and with computation in external asynchronous programming systems. To do so, this chapter defines a MPI_Queue object for ordering initiation and completion of persistent communication operations with respect to each other and with an optional (opaque) external execution context. It also refines the matching, ordering, progress, and concurrency semantics of the initiation and completion of enqueued MPI persistent communication operations and the exact semantics of enqueued MPI communication operations when no opaque external execution context is provided.

The semantics of enqueued persistent communication operations with respect to computation when an opaque external execution context is provided are outside of the scope of the MPI standard. Systems in which this interface is implemented should provide a separate reference document that describes how operations on *queue* objects interact with its programming environment. Such a document may also specify additional system-specific MPI extensions for interacting with enqueued MPI persistent communication operations.

13.2 Matching Operations

Persistent requests used with enqueueing operations shall be **matched** (via MPI_MATCH or one of its variants) before their start may be enqueued on a *queue*. Matching resolves the pairing between all buffers and exchanges the information required to carry out the data transfer such that subsequent starts enqueued on the *queue* do not require per-operation tag matching, buffer negotiation, or other similar synchronization with the remote MPI process. This separation of matching and other *nonlocal* initialization operations from data movement is essential for use in external execution contexts in which these operations cannot be executed directly.

MPI_MATCH, MPI_IMATCH, MPI_MATCHALL, or MPI_IMATCHALL on a persistent

point-to-point communication request or a partitioned persistent point-to-point communication request is a point-to-point operation between two ranks, the sender and receiver ranks of the corresponding point-to-point operations. When called on persistent point-to-point communication requests, these functions follow the same MPI matching rules as for point-to-point communication (see Section 3) with communicator, tag, and source dictating message matching but may only match persistent sends with persistent receives. In the event that the communicator, tag, and source of persistent point-to-point requests do not uniquely identify a message, the order in which the relevant MPI_MATCH function is called is the order in which they are matched.

When called on partitioned persistent communication requests, these functions follow the matching rules for partitioned communication (see Section 4)

MPI_MATCH, MPI_IMATCH, MPI_MATCHALL, or MPI_IMATCHALL on a persistent collective communication request is a collective operation between all ranks on the communicator associated with the request. These functions may complete only when all persistent collective communication initialization operations associated are complete.

Once *matched*, a *persistent request* retains its match permanently until the request is freed via MPI_REQUEST_FREE. It is erroneous to call MPI_MATCH or one of its variants on a *persistent communication request* that has already been *matched*.

MPI_MATCH(request)

INOUT	request	inactive persistent request to match (handle)

C binding

```
int MPI_Match(MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Match(request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_MATCH(REQUEST, IERROR)
    INTEGER REQUEST, IERROR
```

MPI_MATCH blocks until the *persistent request* request is *matched* with its corresponding *persistent request* on the remote MPI process. The request argument shall be a handle to an *inactive persistent request* created by a persistent point-to-point or collective operation. It is erroneous to pass a *nonpersistent*, *nonblocking*, or generalized request to MPI_MATCH. After MPI_MATCH returns, request remains *inactive* and may be enqueued via MPI_ENQUEUE_START.

MPI_MATCH is a *nonlocal* operation.

MPI_IMATCH(tomatch, matchrequest)

INOUT	tomatch	inactive persistent request to match (handle)
OUT	matchrequest	request for the matching operation (handle)

C binding

```
int MPI_IMatch(MPI_Request *tomatch, MPI_Request *matchrequest)
```

Fortran 2008 binding

```
MPI_IMatch(tomatch, matchrequest, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: tomatch
    TYPE(MPI_Request), INTENT(OUT) :: matchrequest
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_IMATCH(TOMATCH, MATCHREQUEST, IERROR)
    INTEGER TOMATCH, MATCHREQUEST, IERROR
```

MPI_IMATCH initiates a *nonblocking* match for the *persistent request* tomatch and returns a new request handle in matchrequest that may be used to test or wait for the completion of the matching operation. The tomatch argument shall satisfy the same preconditions as the request argument of MPI_MATCH. The returned matchrequest is a *nonblocking*, *nonpersistent* request that cannot be canceled; it is considered complete when the match for tomatch has been resolved. After the matching operation identified by matchrequest has completed, tomatch may be enqueued via MPI_ENQUEUE_START.

MPI_IMATCH is a *local* operation.

MPI_MATCHALL(count, array of requests)

IN	count	list length (nonnegative integer)
INOUT	array of requests	array of inactive persistent requests to match (array of handles)

C binding

```
int MPI_Matchall(int count, MPI_Request array of requests[])
```

Fortran 2008 binding

```
MPI_Matchall(count, array of requests, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array of requests(count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_MATCHALL(COUNT, ARRAY OF REQUESTS, IERROR)
    INTEGER COUNT, ARRAY OF REQUESTS(*), IERROR
```

MPI_MATCHALL has the same effect as calling MPI_MATCH on each element of array of requests. Each request in array of requests shall satisfy the preconditions required by MPI_MATCH. MPI_MATCHALL with an array of length one is equivalent to MPI_MATCH.

MPI_MATCHALL is a *nonlocal* operation.

MPI_IMATCHALL(count, array of requests, request)

IN	count	list length (nonnegative integer)
INOUT	array of requests	array of inactive persistent requests to match (array of handles)

OUT	request	request for the matching operation (handle)

C binding

```
int MPI_IMatchall(int count, MPI_Request array of requests[],
              MPI_Request *request)
```

Fortran 2008 binding

```
MPI_IMatchall(count, array of requests, request, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array of requests(count)
    TYPE(MPI_Request), INTENT(OUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_IMATCHALL(COUNT, ARRAY OF REQUESTS, REQUEST, IERROR)
    INTEGER COUNT, ARRAY OF REQUESTS(*), REQUEST, IERROR
```

MPI_IMATCHALL initiates nonblocking matching for all requests in array of requests and returns a single new request handle in request that is complete when all requests in the array have been *matched*. Each request in array of requests shall satisfy the preconditions required by MPI_IMATCH. The returned request is *nonblocking* and *nonpersistent* and cannot be canceled. MPI_IMATCHALL with an array of length one is equivalent to MPI_IMATCH.

MPI_IMATCHALL is a *local* operation.

MPI_IS_MATCHED(request, flag)

IN	request	persistent request to query (handle)
OUT	flag	true if the request has been *matched* (logical)

C binding

```
int MPI_Is_matched(MPI_Request request, int *flag)
```

Fortran 2008 binding

```
MPI_Is_matched(request, flag, ierror)
    TYPE(MPI_Request), INTENT(IN) :: request
    LOGICAL, INTENT(OUT) :: flag
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_IS_MATCHED(REQUEST, FLAG, IERROR)
    INTEGER REQUEST, IERROR
    LOGICAL FLAG
```

MPI_IS_MATCHED sets flag to true if request has been *matched* (i.e., it has been processed by MPI_MATCH, MPI_IMATCH, MPI_MATCHALL, or MPI_IMATCHALL and the match has completed or it is a *partitioned persistent communication request*), and to false otherwise. The request argument shall be a handle to a *persistent communication request*. MPI_IS_MATCHED does not alter the state of request.

MPI_IS_MATCHED is a *local* operation.

13.3 Queue Objects

A **queue** represents an ordered set of MPI *start* and *wait* operations to execute and order with respect to each other and, optionally, with respect to computations in an external asynchronous programming system. A *queue* includes a provided *queue type* and, if that type supports it, an ordering-context object provided by an external asynchronous programming system. Examples of such ordering-context objects include, but are not limited to, CUDA streams and SYCL queues. MPI start and wait operations on the same *queue* are sequentially ordered with respect to each other and the execution order defined by the associated external execution context. MPI start and wait operations issued to a *queue* and communication operations issued outside that *queue* or to a different *queue* shall be considered concurrent. Concurrent communication operations may be started or completed by an MPI implementation in any order.

13.3.1 Queue Creation and Destruction

MPI_QUEUE_INIT(queue, type, external)

OUT	queue	handle to the created queue object (handle)
IN	type	type of the queue object (integer)
IN	external	pointer to the associated opaque external ordering-context object (choice)

C binding

```
int MPI_Queue_init(MPI_Queue *queue, int type, void *external)
```

Fortran 2008 binding

```
MPI_Queue_init(queue, type, external, ierror)
    TYPE(MPI_Queue), INTENT(OUT) :: queue
    INTEGER, INTENT(IN) :: type
    TYPE(*), DIMENSION(..), INTENT(IN) :: external
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_QUEUE_INIT(QUEUE, TYPE, EXTERNAL, IERROR)
    INTEGER QUEUE, TYPE, IERROR
    <type> EXTERNAL(*)
```

MPI_QUEUE_INIT creates a **queue object** and returns a handle to it in queue. A *queue object* is the concrete handle through which an MPI program refers to an abstract *queue*. The type argument identifies the external asynchronous programming system with which the *queue* is associated; with the exception of MPI_QUEUE_TYPE_DEFAULT, valid type values and their meanings are defined by the implementation for each supported external programming system. It is erroneous to pass a type value not supported by the implementation. The external argument is a pointer to the execution context object in that programming system (e.g., a GPU stream handle) with which the operations enqueued on this *queue* are ordered. The interpretation of external is determined by type.

MPI_QUEUE_INIT is a *local* operation.

A *queue object* can be *freed* using the following MPI procedure.

MPI_QUEUE_FREE(queue)

INOUT	queue	address of queue to free (handle)

C binding

```
int MPI_Queue_free(MPI_Queue *queue)
```

Fortran 2008 binding

```
MPI_Queue_free(queue, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_QUEUE_FREE(QUEUE, IERROR)
    INTEGER QUEUE, IERROR
```

MPI_QUEUE_FREE marks the queue object referenced by queue for deallocation and releases associated resources. It is erroneous to free a *queue* that has enqueued operations that have not yet been completed (i.e., a *queue* that is not empty). After the call returns, queue is set to MPI_QUEUE_NULL.

MPI_QUEUE_FREE is a *local* operation.

13.3.2 Enqueuing Operations

MPI_ENQUEUE_START(queue, request)

INOUT	queue	address of queue on which to enqueue start (handle)
INOUT	request	inactive matched persistent request to start (handle)

C binding

```
int MPI_Enqueue_start(MPI_Queue *queue, MPI_Request *request)
```

Fortran 2008 binding

```
MPI_Enqueue_start(queue, request, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    TYPE(MPI_Request), INTENT(INOUT) :: request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ENQUEUE_START(QUEUE, REQUEST, IERROR)
    INTEGER QUEUE, REQUEST, IERROR
```

MPI_ENQUEUE_START enqueues the start of the *matched* persistent communication request on queue. The queue is updated to record the enqueued start operation.

The request argument shall be a handle to a *matched persistent request*, as returned by MPI_MATCH or MPI_MATCHALL, or a *partitioned persistent request* (which is initialized as *matched*, e.g., via MPI_PSEND_INIT). It is erroneous to pass a *nonblocking* request, a generalized request, or an unmatched *persistent request* as request; in this case

MPI_ENQUEUE_START returns an error and the request is not enqueued. In order to enqueue the start of request, the request shall either be *inactive* or have been started using MPI_ENQUEUE_START and have a corresponding MPI_ENQUEUE_WAIT already enqueued on the same *queue*.

Once MPI_ENQUEUE_START returns, it is erroneous to call any MPI operations on request prior to calling MPI_ENQUEUE_WAIT. A request whose start has been enqueued on a *queue* may only be waited on using the same *queue*; it is erroneous to enqueue a wait for request on a different *queue* or to directly call MPI_WAIT on the request.

MPI_ENQUEUE_START is a *local* operation.

MPI_ENQUEUE_STARTALL(queue, count, array of requests)

INOUT	queue	address of queue on which to enqueue startall (handle)
IN	count	list length (nonnegative integer)
INOUT	array of requests	array of inactive matched persistent requests to start (array of handles)

C binding

```
int MPI_Enqueue_startall(MPI_Queue *queue, int count, MPI_Request array of
             requests[])
```

Fortran 2008 binding

```
MPI_Enqueue_startall(queue, count, array of requests, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array of requests(count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ENQUEUE_STARTALL(QUEUE, COUNT, ARRAY OF REQUESTS, IERROR)
    INTEGER QUEUE, COUNT, ARRAY OF REQUESTS(*), IERROR
```

MPI_ENQUEUE_STARTALL has the same effect as the execution of MPI_ENQUEUE_START for each element of array of requests in some arbitrary order. Each request in array of requests shall satisfy the preconditions required by MPI_ENQUEUE_START. If any request in the array does not satisfy these preconditions, no request in the array is enqueued and an error is returned. MPI_ENQUEUE_STARTALL with an array of length one is equivalent to MPI_ENQUEUE_START.

MPI_ENQUEUE_STARTALL is a *local* operation.

MPI_ENQUEUE_WAIT(queue, request, status)

INOUT	queue	address of queue on which to enqueue wait (handle)
INOUT	request	matched persistent request whose start is pending on queue (handle)
OUT	status	status object for the completed operation (status)

C binding

```
int MPI_Enqueue_wait(MPI_Queue *queue, MPI_Request *request,
             MPI_Status *status)
```

Fortran 2008 binding

```
MPI_Enqueue_wait(queue, request, status, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    TYPE(MPI_Request), INTENT(INOUT) :: request
    TYPE(MPI_Status) :: status
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ENQUEUE_WAIT(QUEUE, REQUEST, STATUS, IERROR)
    INTEGER QUEUE, REQUEST, STATUS(MPI_STATUS_SIZE), IERROR
```

MPI_ENQUEUE_WAIT enqueues a wait for the completion of the communication operation identified by request on queue. Queue execution is blocked until request has *completed*.

The request argument shall be a handle to a *matched persistent request* whose start has been enqueued on queue via MPI_ENQUEUE_START. It is erroneous to enqueue a wait for request on a different *queue* than the one on which its start was enqueued, or to enqueue a wait for a request whose start has not yet been enqueued.

When the enqueued wait *completes*, request is marked *inactive* and status is set to reflect the completed operation, with the same content as described for MPI_WAIT. If status is MPI_STATUS_IGNORE, no status information is stored. Because the enqueued request is processed asynchronously, the contents of the status argument and the state of the request object are undefined until the enqueued wait has completed. An MPI program may ensure the completion of the enqueued wait by calling MPI_QUEUE_FENCE on the queue or through another mechanism specific to associated asynchronous execution system. Once the caller has ensured that the enqueued wait has completed, it may safely examine the status argument or call MPI_REQUEST_FREE to free the request.

MPI_ENQUEUE_WAIT is a *local* operation.

MPI_ENQUEUE_WAITALL(queue, count, array of requests, array of statuses)

INOUT	queue	address of queue on which to enqueue waitall (handle)
IN	count	list length (nonnegative integer)
INOUT	array of requests	array of matched persistent requests whose starts are pending on queue (array of handles)
OUT	array of statuses	array of status objects for the completed operations (array of status)

C binding

```
int MPI_Enqueue_waitall(MPI_Queue *queue, int count, MPI_Request array of
             requests[], MPI_Status array of statuses[])
```

Fortran 2008 binding

```
MPI_Enqueue_waitall(queue, count, array of requests, array of statuses, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    INTEGER, INTENT(IN) :: count
```

```
    TYPE(MPI_Request), INTENT(INOUT) :: array of requests(count)
    TYPE(MPI_Status) :: array of statuses(count)
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_ENQUEUE_WAITALL(QUEUE, COUNT, ARRAY OF REQUESTS, ARRAY OF STATUSES, IERROR)
    INTEGER QUEUE, COUNT, ARRAY OF REQUESTS(*),
              ARRAY OF STATUSES(MPI_STATUS_SIZE, count), IERROR
```

MPI_ENQUEUE_WAITALL has the same effect as the execution of MPI_ENQUEUE_WAIT for each element of array of requests in some arbitrary order. Each request in array of requests shall satisfy the preconditions required by MPI_ENQUEUE_WAIT. Upon completion of each enqueued wait, array of statuses[i] is set to reflect the completed operation for array of requests[i], with the same content as described for MPI_WAIT. If array of statuses is MPI_STATUSES_IGNORE, no status information is stored. MPI_ENQUEUE_WAITALL with an array of length one is equivalent to MPI_ENQUEUE_WAIT.

MPI_ENQUEUE_WAITALL is a *local* operation.

13.3.3 Queue Fencing Operations

MPI_QUEUE_FENCE(queue)

INOUT	queue	address of queue on which to wait for all enqueued operations to complete (handle)

C binding

```
int MPI_Queue_fence(MPI_Queue *queue)
```

Fortran 2008 binding

```
MPI_Queue_fence(queue, ierror)
    TYPE(MPI_Queue), INTENT(INOUT) :: queue
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_QUEUE_FENCE(QUEUE, IERROR)
    INTEGER QUEUE, IERROR
```

MPI_QUEUE_FENCE blocks the calling thread until all operations previously enqueued on queue have *completed*. This call synchronizes the calling thread with the external execution context associated with queue and allows the calling thread to determine when all enqueued communication operations have finished. After MPI_QUEUE_FENCE returns, all requests whose waits were enqueued on queue are *inactive* and their associated communication buffers may be safely reused.

MPI_QUEUE_FENCE is a *nonlocal* operation.

13.3.4 Queue Types

The MPI standard defines only a single predefined queue type: MPI_QUEUE_TYPE_DEFAULT. When this *queue* type is used, the associated external execution environment handle is ignored. When using MPI_QUEUE_TYPE_DEFAULT, calls to

MPI_ENQUEUE_START and MPI_ENQUEUE_STARTALL are semantically equivalent to calling, respectively, MPI_START and MPI_STARTALL. Calls to MPI_ENQUEUE_WAIT and MPI_ENQUEUE_WAITALL applied to a *queue* with type MPI_QUEUE_TYPE_DEFAULT immediately begin completion processing of the corresponding communication operations but do not block the calling thread. A call to MPI_QUEUE_FENCE on a *queue* with type MPI_QUEUE_TYPE_DEFAULT blocks until all enqueued start and wait operations on that *queue* have completed.

13.4 Concurrency and Progress

Ordering within a *queue*. Start and wait operations enqueued on a *queue* execute in the order in which they were enqueued. Specifically, an enqueued start does not initiate the underlying communication until all previously enqueued start operations on the same *queue* have initiated and all previously enqueued wait operations have completed; similarly, an enqueued wait does not complete until both the associated communication request has completed and all previously enqueued waits on the same *queue* have completed. Start operations and wait operations enqueued simultaneously (using, e.g., MPI_ENQUEUE_STARTALL or MPI_ENQUEUE_WAITALL), may initiate or complete the specified requests in any order. How enqueued start and wait operations are ordered with respect to non-MPI operations on the associated asynchronous execution context is separately defined by the relevant asynchronous programming system.

Concurrency between queues. Operations enqueued on different *queue*s may proceed concurrently. The *queue* abstraction is local; *queue*s on the same or different MPI processes have no inherent ordering relationship with respect to one another. An MPI program that requires ordering between operations on two different *queue*s shall introduce explicit synchronization, for example by calling MPI_QUEUE_FENCE on both *queue*s.

> *Rationale.* The serial-execution-context guarantee within a *queue*, combined with the independence between *queue*s, allows implementations to assign dedicated network and hardware resources to each *queue* and to omit locking on those resources within the execution of a single *queue*. This is the primary performance motivation for the *queue* abstraction. (*End of rationale.*)

Thread safety. A *queue* object shall not be accessed concurrently by more than one thread at a time. It is the responsibility of the calling program to ensure that all accesses to a given *queue* object (including MPI_ENQUEUE_START, MPI_ENQUEUE_WAIT, MPI_ENQUEUE_STARTALL, MPI_ENQUEUE_WAITALL, and MPI_QUEUE_FENCE) are serialized. This requirement is consistent with the serial-execution-context semantics of a *queue*; users of an asynchronous programming system that dispatches operations from multiple threads to a single *queue* shall serialize those dispatches, as is required by the programming system's own serial-context invariant.

MPI programs that use *queue* objects may use any MPI thread model. Operations started by external execution contexts shall respect the provided MPI thread model.

Progress of enqueued operations. An MPI implementation is required to guarantee

progress on operations enqueued on a *queue*. Progress on a *queue* is guaranteed during any call to MPI_QUEUE_FENCE, and during any *blocked* MPI procedure call or MPI test procedure on the same MPI process (see Section 2.9). Additional mechanisms by which an implementation may guarantee progress include a dedicated progress thread, offloading to a network interface controller, or coordination with the external execution context associated with the *queue*.

Progress on a *queue* is guaranteed to be provided independently of any enqueued operations on other *queue*s. An MPI process that has enqueued operations on a *queue* but does not subsequently call MPI_QUEUE_FENCE or any other MPI procedure may not be guaranteed progress; programs that require communication to complete without calling into MPI on the CPU shall use MPI_QUEUE_FENCE or rely on implementation-specific guarantees provided by the associated external execution context.

> *Rationale.* For some external execution contexts, such as GPU streams, the execution context itself may drive progress without CPU involvement. The standard does not mandate this capability because it depends on implementation and hardware support, but conforming implementations are encouraged to provide it for contexts where it is feasible. (*End of rationale.*)

Ordering with respect to standard MPI operations. *Matched persistent request*s used with the enqueueing operations in this chapter may also be accessed through standard MPI operations before their start has been enqueued on a *queue* (for example, using MPI_START or MPI_STARTALL). Once an enqueued start has been issued for a request, no standard MPI operations may be called on that request until the corresponding enqueued wait has completed (i.e., after the request has been returned to the *inactive* state by the *queue*). It is erroneous to call MPI_WAIT, MPI_TEST, MPI_CANCEL, or any other standard MPI operation on a request while its enqueued start is pending.

Enqueued communication operations interact with standard MPI matching rules in the same way as equivalent operations issued by the calling thread. A *matched* persistent send request enqueued on a *queue* satisfies any matching receive and is subject to the same message-ordering rules as a send issued by any other means.

Deadlock. Enqueued operations are executed asynchronously with respect to the calling thread, introducing potential sources of deadlock into programs and implementations. In particular:

- An enqueued wait on a *queue* blocks *queue* execution until the corresponding communication has completed. If the matching operation on the remote MPI process is never started, the enqueued wait never completes. MPI already requires that programs ensure that matching sends and receives are started; this requirement applies equally to enqueued operations.
- An external execution context may impose additional constraints on when enqueued operations are eligible for execution. For example, a GPU scheduler that is not given sufficient resources to schedule a stream containing enqueued communication operations may deadlock. Programs should ensure that exeternal execution contexts associated with MPI *queue*s are given appropriate scheduling priority.

13.5 Examples

The following example illustrates the use of queued communication to implement a ring exchange across a one-dimensional ring of MPI processes. The example is organized in two phases. In the setup phase (done once before the iteration loop), a *queue* is created and bound to an external execution context, persistent send and receive requests are created for each neighbor, and all requests are *matched* via MPI_MATCHALL to move the matching protocol off the per-iteration critical path. In the iteration phase, starts for the receive and send requests are enqueued on the *queue*, followed by an enqueued wait for all requests. After all iterations, MPI_QUEUE_FENCE blocks the calling thread until all enqueued operations have completed.

Example 13.1. Stream-triggered data exchange in a one-dimensional ring of MPI processes. Code that interfaces with an external execution environment would include a context for that environment as part of *queue* creation and use an execution environment-specific call to schedule computation at the commented points relative to the MPI communication calls.

```
#include "mpi.h"
#define N    1024
#define NITER 100

int main(int argc, char *argv[])
{
    int rank, size, left, right;
    double send_left[N], send_right[N];
    double recv_left[N], recv_right[N];
    MPI_Queue queue;
    MPI_Request reqs[4];
    MPI_Status  statuses[4];

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    /* Determine neighbors in a 1-D ring topology */
    left  = (rank - 1 + size) % size;
    right = (rank + 1) % size;

    /* Initialize a queue. A system-specific type and external execution
     * context can be used if available. */
    MPI_Queue_init(&queue, MPI_QUEUE_TYPE_DEFAULT, NULL);

    /* Create persistent send and receive requests for ring exchange */
    MPI_Recv_init(recv_left,  N, MPI_DOUBLE, left,  0,
                  MPI_COMM_WORLD, &reqs[0]);
    MPI_Recv_init(recv_right, N, MPI_DOUBLE, right, 0,
                  MPI_COMM_WORLD, &reqs[1]);
    MPI_Send_init(send_left,  N, MPI_DOUBLE, left,  0,
                  MPI_COMM_WORLD, &reqs[2]);
    MPI_Send_init(send_right, N, MPI_DOUBLE, right, 0,
                  MPI_COMM_WORLD, &reqs[3]);
```

```
    /* Match all requests once, removing per-iteration tag matching and
       buffer negotiation from the communication critical path */
    MPI_Matchall(4, reqs);

    /* Enqueue the data exchange operations each iteration */
    for (int iter = 0; iter < NITER; iter++) {
        MPI_Enqueue_startall(queue, 2, &reqs[0]);

        /* Schedule iteration work needed prior to sending
        * (e.g., packing send buffers) here */
        MPI_Enqueue_startall(queue, 2, &reqs[2]);

        /* Schedule work overlapped with communication here */

        /* Enqueue waits for all sends and receives to complete */
        MPI_Enqueue_waitall(queue, 4, reqs, statuses);

        /* Schedule work that depends on the results of the receive
         * (e.g., unpacking receive buffers) here */
    }

    /* Block until all enqueued communication operations complete */
    MPI_Queue_fence(queue);

    /* Free requests and the queue */
    for (int i = 0; i < 4; i++)
        MPI_Request_free(&reqs[i]);
    MPI_Queue_free(&queue);

    MPI_Finalize();
    return 0;
}
```

In this example, the setup phase creates four *persistent requests* (two receives and two sends) and matches them all with a single call to MPI_MATCHALL. This call resolves the pairing between sends and receives on both processes and exchanges the information needed for data transfer, so that subsequent enqueued starts do not require per-operation tag matching. Matching is a *nonlocal* operation: both the sending and receiving processes shall call a matching operation for the match to complete.

In the iteration loop, MPI_ENQUEUE_STARTALL is called first for the receive requests and then for the send requests. Both calls are *local* operations that return immediately after recording the enqueued starts; neither call blocks the calling thread or blocks execution of the *queue*. MPI_ENQUEUE_WAITALL enqueues waits for all four requests and is likewise a *local* operation. Execution of the enqueued starts and waits on the *queue* proceeds asynchronously, ordered with respect to any computational operations enqueued on the same external execution context.

After the loop, MPI_QUEUE_FENCE synchronizes the calling thread with the *queue*, blocking until all enqueued operations have completed. Once MPI_QUEUE_FENCE returns, all requests are *inactive* and their communication buffers may be safely reused or freed.