3.11 Completion Continuations

Some applications may need to handle large numbers of requests or require fast reaction to changes in the state of an active request, i.e., the completion or cancellation of the operation. The reaction to the completion of an operation can be expressed as a *continuation*, i.e., a call-back function provided by the application that is invoked by MPI once completion or cancellation of the operation is detected. This alleviates the pressure on the application to repeatedly test large numbers of requests and allows for fast reaction to state changes.

Continuations are *attached* to either a single request or a set of requests and *registered* with a *continuation request*. A continuation request has to be initialized and freed by the application and can be used to test or wait for the completion of all registered continuations. Continuation requests themselves may have a continuation attached, which will be invoked once all continuations registered with it have been executed. Attaching a continuations to a non-persistent request returns ownership of that request back to MPI, i.e., the request may not be used to test or wait for the completion of the respective operation. Persistent requests, on the other hand, remain valid after attaching a continuation. However, the outcome of attaching more than one continuation to an active request is undefined.

By default, continuations may be executed by the MPI library at any time and on any thread after the completion of the associated operation or operations. This behavior can be controlled using info keys listed in Section 3.11.3.

When attaching a continuation to an operation request, a status object may be passed, which will be filled before the continuation is invoked. The pointer to the status object is then passed to the continuation. It is the application's responsibility to ensure that the status objects are accessible until the continuation is invoked. The application may also pass MPI_STATUS_IGNORE, in which case this value will also be passed to the continuation.

Example 3.20 uses continuations in order to avoid explicitly tracking buffers and to facilitate a simple throttling mechanism. Similar behavior can be achieved using `MPI_TESTSOME` and manually managing a set of active requests. However, continuations tie the request directly to an action to be executed upon completion of the respective operation, managed by MPI.

Example 3.20 One process sending messages to all others, using continuations to avoid tracking buffers and to limit the number of concurrently active send operations.

```
#include <stdlib.h>
#include <mpi.h>

#define NUM_VARS 1024
#define TAG 1001
#define MAX_ACTIVE_SEND 3
static int num_active_send = 0;

void completion_cb(MPI_Status *status, void *user_data)
{
  --num_active_send;
  free(user_data);
}
```

```
int main(int argc, char *argv[])
{
  int rank, size, flag;
  double *vars;
  MPI_Request op_request, cont_request;
  MPI_Comm comm = MPI_COMM_WORLD;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  MPI_Continue_init(MPI_INFO_NULL, &cont_request);

  if (rank == 0) {
    /* Send message to each peer, not exceeding MAX_ACTIVE_SEND */
    for (int i = 1; i < size; ++i) {
      /* don't exceed the limit */
      while(num_active_send >= MAX_ACTIVE_SEND) {
        MPI_Test(&flag, cont_request, MPI_STATUS_IGNORE);
      }
      ++num_active_send;
      vars = malloc(sizeof(double)*NUM_VARS);
      compute_vars_for(vars, i);
      MPI_Isend(vars, NUM_VARS, MPI_DOUBLE, i, TAG, comm, &op_request);
      /* Attach continuation that frees the buffer once complete */
      MPI_Continue(&op_request, &completion_cb, vars,
                   MPI_STATUS_IGNORE, cont_request);
    }
    /* Wait for remaining continuations to complete */
    MPI_Wait(&cont_request, MPI_STATUS_IGNORE);
  } else {
    vars = malloc(sizeof(double)*NUM_VARS);
    MPI_Recv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, MPI_STATUS_IGNORE);
    compute_vars_from(vars, 0);
    free(vars);
  }
  MPI_Request_free(&contreq);
  MPI_Finalize();
  return 0;
}
```

3.11.1 Continuation Requests

MPI_CONTINUE_INIT(info, cont_req)

IN	info	info argument (handle)
OUT	cont_req	continuation request (handle)

C binding

```
int MPI_Continue_init(MPI_Info info, MPI_Request *cont_req)
```

Fortran 2008 binding

```
MPI_Continue_init(info, cont_req, ierror)
    TYPE(MPI_Info), INTENT(IN) :: info
    TYPE(MPI_Request), INTENT(OUT) :: cont_req
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_CONTINUE_INIT(INFO, CONT_REQ, IERROR)
    INTEGER INFO, CONT_REQ, IERROR
```

This function creates a new continuation request that can be used to register continuations for active operation requests and test or wait for the completion of registered continuations. The info argument can be used to control parameters of the registered continuations. A list of available info keys can be found in Section 3.11.3.

Continuation requests may be passed to MPI_TEST and MPI_WAIT to test or wait for the completion of the execution of all registered continuations. A call to a test or wait function indicating completion of a continuation request does not free the request. A continuation request is complete until the first continuation is registered, at which point the request is marked as incomplete until the last continuation has completed its execution.

A continuation request may be freed through a call to MPI_REQUEST_FREE, which marks the request for deallocation. The request will be deallocated once all registered continuations have been executed.

TODO: which field will be set in the status when waiting for a cont request?

3.11.2 Attaching Continuations

Continuations are attached to requests representing initiated operations using either MPI_CONTINUE or MPI_CONTINUEALL.

The continuation callback function has the type MPI_Continue_cb_function and upon invocation is passed the pointer to the status object(s) and the user-provided pointer to additional data.

```
typedef void MPI_Continue_cb_function(MPI_Status *array_of_statuses,
             void *user_data);
```

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Continue_cb_function(array_of_statuses, user_data, ierror)
    TYPE(MPI_Status) :: array_of_statuses(*)
    INTEGER(KIND=MPI_ADDRESS_KIND) :: user_data
    INTEGER, OPTIONAL :: ierror
```

```
SUBROUTINE MPI_CONTINUE_CB_FUNCTION(ARRAY_OF_STATUSES, USER_DATA, IERROR)
    INTEGER ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) USER_DATA
```

The statuses in the array_of_statuses will be set before the continuation callback is invoked. The application is responsible for allocating and releasing the memory backing the array_of_statuses and that memory shall be accessible for the time between attaching the continuation and the invocation of the continuation. If MPI_STATUS_IGNORE or MPI_STATUSES_IGNORE was passed during registration, then these values will be passed to the callback function invocation instead.

TODO: is it safe to conflate MPI_STATUS_IGNORE and MPI_STATUSES_IGNORE in the callback? Do we need two different callbacks for continue and continueall?

TODO: by passing the status to the callback we may be giving the impression that this is memory managed magically managed by MPI. The user could just as well pass the status in the user_data if needed...

MPI_CONTINUE(op_request, cb, cb_data, status, cont_request)

INOUT	op_request	operation request (handle)
IN	cb	callback to be invoked once the operation is complete (function)
IN	cb_data	pointer to a user-controlled buffer
IN	status	status object (array of status)
IN	cont_request	continuation request (handle)

C binding

```
int MPI_Continue(MPI_Request *op_request, MPI_Continue_cb_function cb,
             void *cb_data, MPI_Status *status, MPI_Request cont_request)
```

Fortran 2008 binding

```
MPI_Continue(op_request, cb, cb_data, status, cont_request, ierror)
    TYPE(MPI_Request), INTENT(INOUT) :: op_request
    MPI_Continue_cb_function, INTENT(IN) :: cb
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: cb_data
    TYPE(MPI_Status), INTENT(IN), ASYNCHRONOUS :: status(1)
    TYPE(MPI_Request), INTENT(IN) :: cont_request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_CONTINUE(OP_REQUEST, CB, CB_DATA, STATUS, CONT_REQUEST, IERROR)
    INTEGER OP_REQUEST, STATUS(MPI_STATUS_SIZE, 1), CONT_REQUEST, IERROR
    MPI_Continue_cb_function CB
    INTEGER(KIND=MPI_ADDRESS_KIND) CB_DATA
```

This function attaches a continuation to the operation request op_request and registering it with the continuation request cont_request. The callback function cb will be invoked after the MPI implementation finds the operation represented by op_request to be complete. Upon invocation, the status pointer will be passed to the callback function together with

the cb_data pointer. The cont_request must be a continuation request created through a call to MPI_CONTINUE_INIT.

If the operation represented by op_request is complete at the time of the call to MPI_CONTINUE, the implementation may invoke the cb immediately, unless the "mpi_continue_enqueue_complete" info key is set to "true" during the creation of the cont_request.

Unless MPI_STATUS_IGNORE is passed as status, the status object pointed to must be accessible until the callback is invoked.

For non-persistent requests, the op_request is set to MPI_REQUEST_NULL before the call returns. The outcome of passing a non-persistent operation request to another MPI function after it was passed to MPI_CONTINUE is undefined.

A persistent operation request will not be set to MPI_REQUEST_NULL before returning from the call and may be passed to other MPI procedures. Upon invocation of the attached continuation, the persistent request will be inactive and can be started inside the continuation. The outcome of attaching more than one continuation to an operation request is undefined.

MPI_CONTINUEALL(count, array_of_op_requests, cb, cb_data, array_of_statuses, cont_request)

IN	count	list length (non-negative integer)
INOUT	array_of_op_requests	array of requests (array of handles)
IN	cb	callback to be invoked once the operation is complete (function)
IN	cb_data	pointer to a user-controlled buffer
IN	array_of_statuses	array of status objects (array of status)
IN	cont_request	continuation request (handle)

C binding

```
int MPI_Continueall(int count, MPI_Request array_of_op_requests[],
              MPI_Continue_cb_function cb, void *cb_data,
              MPI_Status array_of_statuses[], MPI_Request cont_request)
```

Fortran 2008 binding

```
MPI_Continueall(count, array_of_op_requests, cb, cb_data,
              array_of_statuses, cont_request, ierror)
    INTEGER, INTENT(IN) :: count
    TYPE(MPI_Request), INTENT(INOUT) :: array_of_op_requests(count)
    MPI_Continue_cb_function, INTENT(IN) :: cb
    INTEGER(KIND=MPI_ADDRESS_KIND), INTENT(IN) :: cb_data
    TYPE(MPI_Status), INTENT(IN), ASYNCHRONOUS :: array_of_statuses(*)
    TYPE(MPI_Request), INTENT(IN) :: cont_request
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_CONTINUEALL(COUNT, ARRAY_OF_OP_REQUESTS, CB, CB_DATA,
              ARRAY_OF_STATUSES, CONT_REQUEST, IERROR)
```

```
INTEGER COUNT, ARRAY_OF_OP_REQUESTS(*),
          ARRAY_OF_STATUSES(MPI_STATUS_SIZE, *), CONT_REQUEST, IERROR
MPI_Continue_cb_function CB
INTEGER(KIND=MPI_ADDRESS_KIND) CB_DATA
```

Similar to MPI_CONTINUE, this function is used to attach a continuation callback to a set of operation requests. The callback will be invoked once all operations in the set array_of_op_requests have completed. If MPI_STATUSES_IGNORE is not passed for array_of_statuses, the statuses will be set before the continuation is invoked and the provided pointer passed to the callback. Otherwise, MPI_STATUSES_IGNORE is passed to the callback function. The rules regarding persistent and non-persistent requests described for MPI_CONTINUE also apply here.

3.11.3 Predefined Info Keys

The behavior of continuations and continuation requests can be controlled using the following info keys:

"mpi_continue_poll_only" Continuations registered with a continuation request created with this key set to "true" will only be executed once the associated continuation request is tested or waited for completion. The default is "false".

"mpi_continue_enqueue_complete" If this info key is set to "true", then a continuation will not be executed during a call to MPI_CONTINUE or MPI_CONTINUEALL even if the associated operation(s) are complete immediately. Instead the continuation is enqueued for later execution, e.g., when polling on the associated continuation request. The default is "false".

"mpi_continue_max_poll" This key sets the maximum number of continuations that should be invoked once the continuation request is tested. This may be useful in cases where limited time should be spent on processing continuations at once, e.g., an application-level communication thread responsible for handling both incoming and outgoing messages. The default is "-1", meaning unlimited. Setting both "mpi_continue_max_poll" to "0" and "mpi_continue_poll_only" to "true" is erroneous as it would cause no continuation registered with this continuation request to ever be executed.

Advice to users. MPI has no knowledge or control over what actions the continuation it invokes will perform. Thus, MPI cannot decide on its own how many or at what time to execute continuations. Continuations that take long to execute may block the calling thread and prevent progress of other aspects of the application. It may thus be necessary for the application to balance the need for quick reaction to MPI operation completions with reactivity in other (non-communication related) aspects of the application. (*End of advice to users.*)

The execution context of continuations can be controlled using the following info keys:

"mpi_continue_thread" This key may be set to one of the following two values: "application" and "any". The "application" value indicates that continuations may only be executed by threads controlled by the application, i.e., any application thread that calls into MPI. This is the default. The value "any" indicates that continuations may be executed

by *any* thread, including MPI-internal progress threads if available. This key has no effect on implementations that do not use an internal progress thread.

Rationale. Some applications may rely on thread-local data being initialized outside of the continuation or use callbacks that are not thread-safe, in which case the use of "any" would lead to correctness issues. (*End of rationale.*)

"mpi_continue_async_signal_safe" If the value is set to "true", the application provides a hint to the implementation that the continuations are async-signal safe and thus may be invoked from within a signal handler. This limits the capabilities of the callback, excluding calls back into the MPI library and other unsafe operations. The default is "false".

3.11.4 Examples

Example 3.21 Using a continuation on persistent receive request, restarting the request after processing an incoming message. This examples uses MPI_WAIT to wait for the completion of all continuations registered with the continuation request and thus to wait for all messages to be processed.

```
#include <stdlib.h>
#include <mpi.h>

#define NUM_VARS 1024
#define TAG 1001

static MPI_Request recv_request, cont_request;
static volatile int num_recvs = 0;
static int world_size;

void completion_cb(MPI_Status *status, void *user_data)
{
  process_vars(user_data);
  if (num_recvs++ < world_size-1) {
    MPI_Start(&recv_request);
    MPI_Continue(&recv_request, &completion_cb, vars,
                 status, cont_request);
  }
}

int main(int argc, char *argv[])
{
  int rank;
  MPI_Status status;
  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &world_size);
  MPI_Comm_rank(comm, &rank);
```

```
  double *vars = malloc(sizeof(double)*NUM_VARS);
  if (rank == 0) {
    MPI_Continue_init(MPI_INFO_NULL, &cont_request);
    MPI_Recv_init(vars, NUM_VARS, MPI_DOUBLE, MPI_ANY_SOURCE, TAG,
                  comm, &recv_request);
    MPI_Start(&recv_request);
    MPI_Continue(&recv_request, &completion_cb, vars,
                 &status, cont_request);
    /* wait for all messages to be received */
    MPI_Wait(&cont_request, MPI_STATUS_IGNORE);
    MPI_Request_free(&recv_request);
    MPI_Request_free(&cont_request);
  } else {
    create_vars(vars);
    MPI_Send(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm);
  }

  free(vars);
  MPI_Finalize();
  return 0;
}
```

Example 3.22 Using continuations to react to an arbitrary number of messages (sender not shown) in a library and checking for cancellation of the receive request inside the continuation. The progress function should be called periodically by the library's user.

```
#include <mpi.h>

#define TAG 1001

static MPI_Request recv_request, cont_request;
static MPI_Status status;

void completion_cb(MPI_Status *status, void *user_data)
{
  int cancelled;
  /* test whether the receive was cancelled, process otherwise */
  MPI_Test_cancelled(status, &cancelled);
  if (cancelled) {
    MPI_Request_free(&recv_request);
  } else {
    process_msg(user_data);
    MPI_Start(&recv_request);
    MPI_Continue(&recv_request, &send_completion_cb, vars,
                 status, cont_request);
  }
```

```
}

void init_recv(void *buffer, int num_bytes)
{
  /* initialize continuation request and start a receive */
  MPI_Continue_init(MPI_INFO_NULL, &cont_request);
  MPI_Recv_init(vars, num_bytes, MPI_BYTE, MPI_ANY_SOURCE, TAG,
                comm, &recv_request);
  MPI_Start(&recv_request);
  MPI_Continue(&recv_request, &completion_cb, buffer,
               &status, cont_request);
}

void progress()
{
  int flag; // ignored
  /* progress outstanding continuations */
  MPI_Test(&cont_request, &flag, MPI_STATUS_IGNORE);
}

void end_recv()
{
  /* cancel the request and wait for the last continuation to complete */
  MPI_Cancel(&recv_request);
  MPI_Wait(&cont_request, MPI_STATUS_IGNORE);
  MPI_Request_free(&cont_request);
}
```

Example 3.23 Using continuations to handle detached OpenMP tasks communicating through MPI. An additional background thread is needed to ensure progress on outstanding continuations.

```
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <omp.h>
#include <mpi.h>

#define NUM_VARS 1024
#define TAG 1001

void send_completion_cb(MPI_Status *status, void *user_data)
{
  free(user_data);
}

void recv_completion_cb(MPI_Status *status, void *user_data)
```

```
{
  omp_fulfill_event((omp_event_t) user_data);
}

static volatile int need_progress = 1;
void* progress_thread(void *arg)
{
  int flag;
  MPI_Request *cont_request = (MPI_Request*)arg;
  while (need_progress) {
    MPI_Test(cont_request, &flag, MPI_STATUS_IGNORE);
    usleep(100);
  }
  return NULL:
}

int main(int argc, char *argv[])
{
  int rank, size;
  MPI_Request op_request, cont_request;
  omp_event_t event;

  MPI_Init(&argc, &argv);
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &size);
  MPI_Comm_rank(comm, &rank);
  MPI_Continue_init(MPI_INFO_NULL, &cont_request);
  /* thread that progresses outstanding continuations */
  pthread_t thread;
  pthread_create(&thread, NULL, &progress_thread, &cont_request);

  #pragma omp parallel master
  {
    if (rank == 0) {
      for (int i = 1; i < size; ++i) {
        #pragma omp task
        {
          double *vars = malloc(sizeof(double)*NUM_VARS);
          compute_vars_for(vars, i);
          MPI_Isend(vars, NUM_VARS, MPI_DOUBLE, i, TAG, comm, &op_request);
          /* attach continuation that frees the buffer once complete */
          MPI_Continue(&op_request, &send_completion_cb, vars,
                       MPI_STATUS_IGNORE, cont_request);
        }
      }
    } else {
      /* task that receives values */
```

```
      double *vars;
      #pragma omp task depend(out: vars) detach(event)
      {
        MPI_Request op_request;
        vars = malloc(sizeof(double)*NUM_VARS);
        MPI_Irecv(vars, NUM_VARS, MPI_DOUBLE, 0, TAG, comm, &op_request);
        MPI_Continue(&op_request, &recv_completion_cb, event,
                     MPI_STATUS_IGNORE, cont_request);
      }
      /* task processing values, executed once the receiving task's
         dependencies are released */
      #pragma omp task depend(in: vars)
      {
        compute_vars_from(vars, 0);
        free(vars);
      }
    }
  }

  need_progress = 0;
  pthread_join(thread, NULL);

  MPI_Request_free(&cont_request);
  MPI_Finalize();
  return 0;
}
```