D R A F T
Document for a Standard Message-Passing Interface

Message Passing Interface Forum

August 22, 2023

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

Chapter 16

Completion Continuations

Some applications may need to handle large numbers of request handles or require fast reaction to the completion or cancellation of an MPI operation. The reaction to the completion of an operation can be expressed as a **continuation**. A continuation is a callback function provided by the application that is invoked by MPI once completion of the operation is detected.

Continuations are **attached** to either a single operation request or a set of operation requests and **registered** with a **continuation request**. A continuation request is a persistent request that is initialized and freed by the application and can be used to test or wait for the completion of all continuations registered to it. A continuation request completes once the callbacks of all continuations registered with it have completed execution. After initialization or completion, a continuation request has to be started to enable the execution of registered continuations. However, continuations can be registered with a valid continuation request at any time.

It is valid to attach a continuation to a continuation request to form a graph of continuations, i.e., to execute a continuation after one or more other continuations and operations have completed. However, the behavior is undefined if cycles are introduced into that graph. Thus, once a continuation has been attached to a continuation request no new continuations may be registered with that same continuation request.

Rationale. The above restriction prohibiting new continuations from being registered with a continuation request that has a continuation attached to it is meant to prevent cyclic dependencies between continuation requests. Otherwise, a deadlock is imminent if the completion of one continuation request is dependent on the completion of the other. (*End of rationale.*)

It is erroneous to pass any request to MPI_CONTINUE and MPI_CONTINUEALL in more than one position. Requests with attached continuations may not be used as input to MPI procedure calls, including to test or wait for the completion of the respective operation. Attaching more than one continuation to a request is erroneous.

When attaching a continuation to an operation a status object may be provided, which will be filled before the continuation callback function is invoked. The application may pass MPI_STATUS[ES]_IGNORE in lieu of a status object or status objects. If status objects are provided, it must be accessible until the callback function is invoked, i.e., the buffer may be freed inside the callback function.

The memory holding request handles must be accessible until the callback function is invoked, i.e., the memory may be freed inside the callback function. The application may pass the flag MPI_CONT_REQUESTS_FREE when attaching a continuation, signalling that the buffer holding the request handles will not be accessible after the continuation has been attached and requesting that non-persistent requests are freed immediately. However, this precludes handling of errors related to failed operations or continuations involving these requests. See Section 16.3.3 for details.

Upon succesful completion of the associated operations, non-persistent requests will be freed by MPI and set to MPI_REQUEST_NULL in the memory holding the request handles prior to the invocation of the callback function, unless MPI_CONT_REQUESTS_FREE was provided. Persistent requests will be inactive and may be restarted or freed during or after the execution of the callback function. An operation request with an attached continuation may be canceled, if allowed for the type of operation. Whether or not a request has been canceled can be determined using the associated status object.

The execution of a continuation callback function may occur on any application thread calling into MPI or be restricted to a thread testing or waiting on the associated continuation request (see Section 16.3.3). MPI procedures may be called from within the continuation callback function.

Advice to users. Applications should strive for short-running callback functions so as not to unnecessarily prolong the execution time of the MPI procedure from within which the callback function is executed. While not explicitly prohibited, the use of nonlocal MPI procedures inside the continuation callback function is discouraged and may cause deadlocks. (*End of advice to users.*)

Upon invocation, the callback function is passed an error code and the user-data pointer provided when the continuation was attached. The error code will be MPI_SUCCESS if all operations have completed succesfully. Upon completion, callback functions should return MPI_SUCCESS to signal success or an error code to signal failure. See Section 16.4 for details on error handling in the context of continuations.

Example 16.1 uses continuations to implement a scheme for offloading work to other processes.

Example 16.1. Library functions to offload work to other processes and receive the result. The example shows only the sender, which posts both the send of the work item and a receive for the result and attaches a single continuatiuon to both requests, which will be executed once both operations have completed and marks the work as completed. The continuation callback will eventually be executed once the continuation request is tested for completion in a call to the progress function. However, they may also be executed in an earlier call to MPI. By using continuations, the requests needed to track the corresponding send and receive operations do not have to be managed in application space. Instead, progressing the continuation request is sufficient to react to the completion of both operations.

```
#include <mpi.h>
/* work descriptor */
struct work_item {
  MPI_Request reqs[2];
  MPI_Status stats[2];
  work_t *msg;
  int size;
  int reply;
};

MPI_Request cont_request;
MPI_Comm comm;
```

```
int next_target();

void init()
{
  MPI_Continue_init(0, 0, MPI_INFO_NULL, &cont_request);
  MPI_Start(&cont_request);
  MPI_Comm_dup(MPI_COMM_WORLD, &comm);
}

void fini()
{
  MPI_Request_free(&cont_request);
  MPI_Comm_free(&comm);
}

/* callback invoked by MPI when send and receive operations are complete */
int complete_cb(int rc, void *user_data)
{
  assert(MPI_SUCCESS == rc); // default error handler would abort
  struct work_item *wd = (struct work_item *wd)user_data;
  int source = wd->stats[1].MPI_SOURCE;
  mark_completed(wd->msg, wd->reply, source);
  free(wd);
  return MPI_SUCCESS;
}

/* send work to a target process and post a receive for the result */
void send_work(work_t *msg, int size)
{
  struct work_item *wd = malloc(sizeof(struct work_item));
  wd->msg = msg;
  wd->size = size;
  int target = next_target();
  /* send the message */
  MPI_Isend(msg, size, MPI_BYTE, target, comm, &wd->reqs[0]);
  /* receive the reply */
  MPI_Irecv(&wd->reply, 1, MPI_INT, target, comm, &wd->reqs[1]);
  /* attach a continuation to both requests */
  MPI_Continueall(2, wd->reqs,
                  &complete_cb, wd,
                  /* flags = */0, wd->stats, &cont_request);
}

/* progress outstanding communication and continuations */
void progress()
{
  int flag;
  MPI_Test(&flag, &cont_request, MPI_STATUS_IGNORE);
  if (flag) {
    MPI_Start(&cont_request);
  }
}
```

16.1 Continuation Requests

MPI_CONTINUE_INIT(flags, max_poll, info, cont_req)

IN	flags	flags (integer)
IN	max_poll	maximum number of continuations to execute when testing, or 0 for no limit (non-negative integer)
IN	info	info argument (handle)
OUT	cont_req	continuation request (handle)

C binding

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

Fortran 2008 binding

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

Fortran binding

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

A call to this procedure creates a new continuation request in cont_req. The flags argument is used to control certain aspects of the continuation request. The following flag is predefined by MPI:

MPI_CONT_POLL_ONLY Marks the continuation request as poll-only, i.e., continuations registered with this continuation request are only executed when a thread tests or wait for the completion of the continuation request. If not provided, continuation callback functions may be executed by any thread calling into MPI.

Additional flags may be added in the future.

The max_poll argument controls the maximum number of continuations to execute when testing for the completion of this continuation request. If multiple continuation requests are tested for completion, the maximum number of continuations executed is the sum of all of their limits. A value of 0 signals that all available continuations may be executed when testing for the completion of the continuation request. Implementations are allowed to impose a limit if max_poll= 0 is set.

The info argument may be used to control the execution context of continuations:

"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".

A call to a test or wait procedure indicating completion of a continuation request does not free the request. A continuation request is released 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. Continuation requests may not be canceled.

16.2 Callback Function Signature

The continuation callback function has the type MPI_Continue_cb_function and upon invocation is passed an error code signalling the state of the associated operations as well as the pointer to additional data provided when attaching a continuation to the operation or set of operations. Unless MPI_CONT_INVOKE_FAILED is specified when attaching a continuation (see Section 16.3.3), the error code passed into the continuation will always be MPI_SUCCESS. The continuation callback should return MPI_SUCCESS if the continuation executed successfully, or an error code otherwise. In the latter case, the continuation will be marked as failed. The handling of failed continuations is discussed in Section 16.4.
The C callback function signature is:

```
typedef int MPI_Continue_cb_function(int error_code, void *user_data);
```

With the mpi_f08 module, the Fortran callback function signature is:

```
ABSTRACT INTERFACE
  SUBROUTINE MPI_Continue_cb_function(error_code, user_data, ierror)
    INTEGER :: error_code, ierror
    INTEGER(KIND=MPI_ADDRESS_KIND) :: user_data
```

With the mpi module and mpif.h, the Fortran callback functions signature is:

```
SUBROUTINE CONTINUE_CB_FUNCTION(ERROR_CODE, USER_DATA, IERROR)
    INTEGER ERROR_CODE, IERROR
    INTEGER(KIND=MPI_ADDRESS_KIND) USER_DATA
```

16.3 Attaching Continuations

16.3.1 Attaching to a Single Request

MPI_CONTINUE(op_request, cb, cb_data, flags, 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	flags	flags controlling aspects of the continuation (integer)
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, int flags, MPI_Status *status,
              MPI_Request cont_request)
```

Fortran 2008 binding

```
MPI_Continue(op_request, cb, cb_data, flags, 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
    INTEGER, INTENT(IN) :: flags
    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, FLAGS, STATUS, CONT_REQUEST, IERROR)
    INTEGER OP_REQUEST, FLAGS, 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 represented by the request op_request and registers it with the continuation request cont_request. The callback function cb will be invoked after the MPI implementation finds the operation to be complete. Upon invocation, the cb_data pointer will be passed to the callback function. The request cont_request must be a continuation request created through a call to MPI_CONTINUE_INIT.

The flags argument is either 0 or an OR-combination of one or more of the flags described in Section 16.3.3.

If the operation represented by op_request is complete at the time of the call to MPI_CONTINUE, the implementation may invoke the callback function cb immediately, unless the MPI_CONT_DEFER_COMPLETE flag is provided. No other continuation callback functions may be invoked during a call to this procedure.

The memory location holding the request must be accessible until the continuation callback is invoked and the request will be set to MPI_REQUEST_NULL before the invocation. However, if MPI_CONT_REQUESTS_FREE was set in flags the memory containing the request handles may be reused after the call to MPI_CONTINUE returns and will not be accessed by MPI afterwards.

Unless MPI_STATUS_IGNORE is passed as status, the status object pointed to by status must be accessible until the continuation callback is invoked. If a status object is provided, it will be filled before the callback function is invoked.

16.3.2 Attaching to Multiple Requests

MPI_CONTINUEALL(count, array_of_op_requests, cb, cb_data, flags, 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	flags	flags controlling aspects of the continuation (integer)
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, int flags,
             MPI_Status array_of_statuses[], MPI_Request cont_request)
```

Fortran 2008 binding

```
MPI_Continueall(count, array_of_op_requests, cb, cb_data, flags,
             array_of_statuses, cont_request, ierror)
    INTEGER, INTENT(IN) :: count, flags
    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, FLAGS,
             ARRAY_OF_STATUSES, CONT_REQUEST, IERROR)
    INTEGER COUNT, ARRAY_OF_OP_REQUESTS(*), FLAGS,
             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 continuation callback will be invoked once all count operations in the list array_of_op_requests have completed. If MPI_STATUSES_IGNORE is not passed for array_of_statuses, the array should be of length count and the statuses will be set before the continuation is invoked. Unless MPI_STATUSES_IGNORE is provided, the memory containing the statuses must remain accessible until the continuation is invoked. Unless the MPI_CONT_REQUESTS_FREE flag is set, the memory containing the request handles must remain accessible until the continuation is invoked.

16.3.3 Flags For Attaching Continuations

MPI_CONT_DEFER_COMPLETE: If provided, MPI will not invoke the continuation callback function immediately even if the operation (or all operations) is complete already while the continuation is beiung attached. The continuation will instead be executed at a later point in time.

MPI_CONT_REQUESTS_FREE: The memory holding the request handles will be freed and should not be accessed by MPI after the return from the procedure call. MPI should thus free any non-persistent request handles and set them to MPI_REQUEST_NULL before the call returns. If this flag is set, the lack of access to the request handles prevents MPI from handling errors occurring in any of the involved operations. The behavior is undefined if an error occurs on an associated operation and a non-aborting error handler has been installed.

> *Advice to users.* Applications that do not wish to handle errors may use this flag to simplify the handling of requests. (*End of advice to users.*)

MPI_CONT_INVOKE_FAILED: The continuation is invoked even if an error is detected for one or more of the associated operations. In that case, the error code for the operation (in the case of MPI_CONTINUE) or MPI_ERR_IN_STATUS is passed as the error code to the continuation callback. If the continuation subsequently returns MPI_SUCCESS the continuation will not be marked as failed. See Section 16.4 for details.

16.4 Error Handling

A continuation may fail for two reasons. First, if the continuation callback returns a value other than MPI_SUCCESS the continuation will be marked as failed and the returned error code as well as MPI_COMM_SELF will be associated with the continuation. Second, if any of the operations the continuation is attached to fails and MPI_CONT_INVOKE_FAILED has not been provided. In that case, the continuation will be marked as failed and the corresponding error and MPI object of the first failed operation will be associated with the continuation. The status for the request of any failed operations will contain the error code for these operation. By default, the continuation will not be executed if any of its operations have failed.

However, if MPI_CONT_INVOKE_FAILED has been provided then the continuation callback will be invoked and either the error code of the failed operation (for MPI_CONTINUE) or MPI_ERR_IN_STATUS (for MPI_CONTINUEALL) will be passed as its first argument and the provided status objects (if any) will contain the error code for the respective operation. If the application is able to handle that error inside the continuation callback, it can subsequently return MPI_SUCCESS from the continuation callback to prevent the error from being propagated outside of the continuation.

The test or wait on a continuation request with a registered failed continuation shall return for that request the error of the first continuation detected as failed. The appropriate error handler will be invoked on the error and MPI object associated with that continuation. If that error handler does not abort the application, the set of failed continuations can be queried from the continuation request using MPI_CONTINUE_GET_FAILED.

MPI_CONTINUE_GET_FAILED(cont_request, count, cb_data)

IN	cont_request	continuation request (handle)
INOUT	count	list length (non-negative integer)
OUT	cb_data	address of a buffer of count pointer to callback data (choice)

C binding

```
int MPI_Continue_get_failed(MPI_Request cont_request, int *count,
             void *cb_data)
```

Fortran 2008 binding

```
MPI_Continue_get_failed(cont_request, count, cb_data, ierror)
    TYPE(MPI_Request), INTENT(IN) :: cont_request
    INTEGER, INTENT(INOUT) :: count
    TYPE(*), DIMENSION(..) :: cb_data
    INTEGER, OPTIONAL, INTENT(OUT) :: ierror
```

Fortran binding

```
MPI_CONTINUE_GET_FAILED(CONT_REQUEST, COUNT, CB_DATA, IERROR)
    INTEGER CONT_REQUEST, COUNT, IERROR
    <type> CB_DATA(*)
```

This function returns at most count pointers to callback data of failed continuations in the buffer pointed to by cb_data. The argument cb_data should be an array of pointers to callback data of length at least count.

Rationale. The use of a formal parameter cb_data of type void* (rather than void**) avoids the messy type casting that would be needed if the callback data pointer are declared with a type other than void*. (*End of rationale.*)

Upon return, count will be set to the actual number of callback data pointers stored in the first count positions of the cb_data array. If the value of count upon return is the same as its input value then there may be more failed continuations to query. Conversely, if count upon return is smaller than the input value then all failed continuations have been queried. The callback data pointer for any given continuation shall not be returned twice, unless $N > 1$ continuations share the same pointer, in which case that pointer value shall be returned N times.

Querying failed continuations does not change the state of the continuation request, i.e., the continuation request has to be started again to enable the execution of continuations and new continuations may be registered with a continuation request that has failed continuations left to query.

16.5 Continuations and Threads

If the implementation supports MPI_THREAD_MULTIPLE, it is safe to register continuations with the same continuation request from within multiple threads. This allows multiple threads within an application to attach continuations to operations without requiring mutual exclusion. However, starting as well as testing and waiting for its completion must be limited to a single thread at a time.

If multiple threads exist within an application, any thread calling into MPI may execute continuation callbacks registered with any continuation request. In order to limit the execution of continuation callback fiunctions to a single thread at a time, the MPI_CONT_POLL_ONLY flag may be passed to MPI_CONTINUE_INIT. Consequently, the callbacks of continuations registered with this continuation request will only be executed by a thread testing or waiting for the completion of the continuation request.

16.6 Examples

Example 16.2. Using a continuations on a persistent receive request, restarting the request and attaching a new continuation after processing an incoming message. This example 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;

int completion_cb(int rc, void *user_data)
{
  process_vars(user_data);
  num_recvs += 1;
  if (num_recvs < world_size-1) {
    MPI_Start(&recv_request);
    /* attach the continuation */
    MPI_Continue(&recv_request,  &completion_cb, user_data,
                 0, &status, cont_request);
  }
  return MPI_SUCCESS;
}

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(0, 0, 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,
                 0, &status, cont_request);
    /* wait for all messages to be received */
    MPI_Start(&cont_request);
    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 16.3. 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 will be called periodically by the library's user to progress the execution of continuations.

```
#include <mpi.h>

static MPI_Request cont_request = MPI_REQUEST_NULL;
static int cont_init_count = 0;
struct callback_data {
  MPI_Request recv_request;
  MPI_Status status;
  void *buffer;
};
typedef struct callback_data callback_data_t;

int completion_cb(int rc, void *user_data)
{
  int cancelled;
  callback_data_t *cd = (callback_data_t)user_data;
  /* test whether the receive was cancelled, process otherwise */
  MPI_Test_cancelled(cd->status, &cancelled);
  if (cancelled) {
    MPI_Request_free(&cd->recv_request);
  } else {
    process_msg(cd->buffer);
    MPI_Start(&cd->recv_request);
    /* attach continuation */
    MPI_Continue(&cb_data->recv_request, &completion_cb, buffer,
                 0, &cd->status, cont_request);
```

```c
  }
  return MPI_SUCCESS;
}

/* initialize a receive with a given tag and register a continuation */
void init_recv(void *buffer, int num_bytes, int tag)
{
  /* initialize continuation request and start a receive */
  callback_data_t *cd = malloc(sizeof(*cd));
  cd->buffer = buffer;
  if (cont_request == MPI_REQUEST_NULL) {
    MPI_Continue_init(0, 0, MPI_INFO_NULL, &cont_request);
    MPI_Start(&cont_request);
    cont_init_count++;
  }
  MPI_Recv_init(cd->buffer, num_bytes, MPI_BYTE, MPI_ANY_SOURCE, tag,
                comm, &cb_data->recv_request);
  MPI_Start(&cd->recv_request);
  MPI_Continue(&cb_data->recv_request, &completion_cb, buffer,
               0, &cd->status, cont_request);
}

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

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);
  --cont_init_count;
  if (cont_init_count == 0) {
    MPI_Request_free(&cont_request);
  }
}
```

Example 16.4. Using continuations to handle detached OpenMP tasks communicating through MPI. An additional background thread is needed to ensure progress on outstanding continuations. For both continuations, the flag MPI_CONT_REQUESTS_FREE is used because the request variable is located on the stack of the thread executing the task and will go out of scope once the task completes. The detach clause on the receive task marks the task as detached, i.e., although the task completes its dependencies will not be fulfilled until the omp_fulfill_event OpenMP procedure is called on the event. This mechanism can be used to encapsulate MPI communication in OpenMP tasks without blocking the thread executing the task.

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

#define NUM_VARS 1024
#define TAG 1001

int send_completion_cb(int rc, void *user_data)
{
  free(user_data);
  return MPI_SUCCESS;
}

int recv_completion_cb(int rc, void *user_data)
{
  omp_fulfill_event((omp_event_t) user_data);
  return MPI_SUCCESS;
}

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);
    if (flag) {
      MPI_Start(&cont_request);
    }
    usleep(100);
  }
  return NULL:
}

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

  MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
  MPI_Comm comm = MPI_COMM_WORLD;
  MPI_Comm_size(comm, &world_size);
  MPI_Comm_rank(comm, &rank);
  MPI_Continue_init(MPI_INFO_NULL, &cont_request);
  MPI_Start(&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) {
      #pragma omp taskloop
      for (int i = 1; i < world_size; ++i) {
        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_CONT_REQUESTS_FREE,
                     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, 0,
                     MPI_CONT_REQUESTS_FREE,
                     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;
}
```

Example 16.5. A modified progress function of Example 16.1 querying failed continuations. A failed target is removed from the list of targets and an unsuccessful work item is resubmitted to another process. This requires a non-aborting error handling (e.g., MPI_ERRORS_RETURN) to be set on the used communicator.

```
/* progress outstanding communication and continuations */
```

```
void offload_progress()
{
  int ret, flag;
  ret = MPI_Test(&flag, &cont_request, MPI_STATUS_IGNORE);
  if (MPI_SUCCESS != ret) {
    /* some continuations have failed, query which ones */
    struct work_item *wds[16];
    int count = 16;
    while (16 == count) {
      MPI_Continue_get_failed(cont_request, &count, wds);
      for (int i = 0; i < count; ++i) {
        MPI_Status status = wds[i]->status[1];
        if (MPI_ERR_PENDING != status.MPI_ERROR) {
          /* mark the target as unavailable */
          remove_target(status.MPI_SOURCE);
          /* resubmit work to another target */
          send_work(wds[i]->work, wds[i]->size);
          free(wds[i]);
        }
      }
    }
    /* all failed continuations have been handled,
      * restart the continuation request */
    MPI_Start(&cont_request);
  } else if (flag) {
    MPI_Start(&cont_request);
  }
}
```

Index