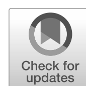
MPI Continuations And How To Invoke Them



Joseph Schuchart and George Bosilca

Abstract Asynchronous programming models (APM) are gaining more and more traction, allowing applications to expose the available concurrency to a runtime system tasked with coordinating the execution. While MPI has long provided support for multi-threaded communication and non-blocking operations, it falls short of adequately supporting the asynchrony of separate but dependent parts of an application coupled by the start and completion of a communication operation. Correctly and efficiently handling MPI communication in different APM models is still a challenge. We have previously proposed an extension to the MPI standard providing operation completion notifications using callbacks, so-called MPI Continuations. This interface is flexible enough to accommodate a wide range of different APMs. In this paper, we discuss different variations of the callback signature and how to best pass data from the code starting the communication operation to the code reacting to its completion. We establish three requirements (efficiency, usability, safety) and evaluate different variations against them. Finally, we find that the current choice is not the best design in terms of both efficiency and safety and propose a simpler, possibly more efficient and safe interface. We also show how the transfer of information into the continuation callback can be largely automated using C++ lambda captures.

1 Background

The Message Passing Interface (MPI) offers a host of nonblocking operations, which are started in a procedure call that immediately returns and provides a request handle representing the operation [4]. At the time of this writing, the only way to know whether an operation has completed is to poll for its completion, either by periodically testing the request or by blocking until its completion in a waiting procedure call. This

Joseph Schuchart and George Bosilca
Innovative Computing Laboratory (ICL), University of Tennessee Knoxville (UTK),
1122 Volunteer Blvd, Knoxville, TN 37996, U.S.A., e-mail: schuchart@icl.utk.edu

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2023

M. M. Resch et al. (eds.), *Sustained Simulation Performance 2021*,
https://doi.org/10.1007/978-3-031-18046-0_5

poses a significant challenge for applications utilizing asynchronous programming models such as OpenMP [9] or higher level distributed runtime systems managing communication through MPI because the requests have to be stored and (repeatedly) passed back into MPI to determine their status.

Over time, several approaches have been proposed that try to hide the synchronizing incurred by waiting for an MPI operation to complete, including TAMPI [6] and the integration of lightweight threads into MPI libraries [3]. These approaches attempt to block and switch the execution context until operations have completed. However, all such approaches are dependent on the support for specific threading implementations and thus not portable.

MPI Continuations, on the other hand, have been proposed as a way to minimize the request management overhead in applications or runtime systems by attaching a callback to a single or a set of continuations [7]. The callback will be executed once all of the operations the continuation was attached to have completed. The application or runtime system can then react to that change in state inside the callback, e.g., by enqueuing a new task or releasing resources associated with that operation. This approach has shown promising results in both OpenMP task-based applications as well as when integrated with the PaRSEC runtime system [8].

A similar approach, dubbed MPI Detach, has been proposed concurrently [5]. While conceptually similar to the MPI Continuations proposal, the callback interface proposed passes a status (or an array of statuses) into the continuations, which would require additional memory management by the MPI library.

In this work, we explore the design of the callback signature of MPI Continuations, focusing on usability, potential performance pitfalls, and safety concerns stemming from the necessary memory management. The rest of this paper is structured as follows: Section 2 provides a short overview over the current state of the continuations proposal. Section 3 discusses various requirements we impose on the design of the callback signature. Section 4 discusses various variations of the callback interface together with their benefits and drawbacks. Section 5 demonstrates the use of the continuations interface in the context of C++. Section 6 draws our conclusions from this exploration.

2 Current state

The Continuations proposal introduces two new concepts into MPI: Continuations and Continuation Requests (CRs). Continuations are a tuple of a callback function and a state on which the callback function operates. Similar concepts can be found in other instances employing the concept of continuations, e.g., continuations proposed for C++ futures in the form of `std::future<T>::then()` [1], which accepts a callable object (e.g., a lambda with it's capture context) that takes the value of type `T` of the future as its sole parameter. Here, the code in the lambda's body is the callback

function while its captured context and the value of type `T` are the state to operate on. The HPX and UPC++ programming systems relies heavily on continuations on C++ futures [2, 10].

2.1 Continuations

MPI Continuations are created using either `MPI_Continue` or `MPI_Continueall` which will **attach** a continuation to a single request or a set of requests, respectively. Since MPI currently only provides C and Fortran interfaces, automatic C++-style context captures cannot be directly supported. Thus, a user-provided data pointer is accepted that will be passed to the continuation callback. This data pointer represents the context of the continuation and is never dereferenced by MPI. It is thus of little relevance to the discussion in this work.

However, an operation in MPI is represented by a request and further information about the outcome of the operation can be gathered from status objects obtained for each request upon its completion (e.g., the tag and sender process rank in the case of a receive operation, or an error code in case of faults). In the case of `MPI_Wait`, a request is passed together with a pointer to a status object. The status object is optional and the application may pass `MPI_STATUS_IGNORE` instead, in which case no further information about the operation will be made available. In its current form, a pointer to a single status object or an array of status objects may be passed to `MPI_Continue` and `MPI_Continueall`, respectively, and the status objects will be set before the continuation is invoked. This pointer will then also be passed as an argument to the continuation.

2.2 Continuation Requests (CR)

Continuation Requests serve a dual purpose. First, they provide an abstract handle to a set of continuations **registered** with this CR, allowing the application to poll for the completion of all registered continuations and (by extension) the associated operations. Once all registered continuations have completed, a wait or test procedure call on that CR will signal its completion (by returning from wait or setting `flag = 1` in a test). Second, CRs provide a facility for progressing outstanding communication operations and to execute eligible continuations.

The relation between CRs, continuations, and operations is shown in Figure 1: multiple continuations may be registered with one continuation request but each continuation may only be registered with a single CR. The latter is a consequence of the fact that continuations are not accessible explicitly through a handle and their lifetime is managed entirely by MPI. Similarly, a continuation may be attached to multiple MPI operations at once, causing the callback to be executed once all of the are complete. However, each MPI operation may only be associated with one

continuation. The transitive closure of these relations is that a CR represents one or many MPI operations and that a successful test on a CR implies the completion of all MPI operations associated with continuations registered with that CR.

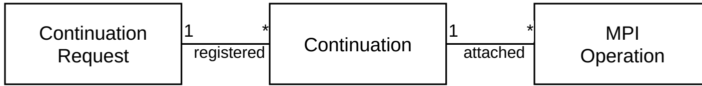


Fig. 1: Relations between Continuation Requests (CR), Continuations, and MPI Operations: multiple continuations can be registered with the same CR (left) and a Continuation can be attached to multiple operations (right). However, only continuation can be attached to any given MPI Operation.

2.3 Current API design

Listing 1 shows the current API as proposed. The ownership of non-persistent requests is returned to MPI and the respective entry in the array is set to `MPI_REQUEST_NULL`. The ownership of persistent requests is not changed. This behavior is similar to that of an optional array of status(es) (or `MPI_STATUS[ES]_IGNORE` otherwise) is passed to the function. The statuses will be set to the statuses of the completed MPI operations before the continuation callback is invoked and the pointer to the statuses provided by the user is passed as the first argument.

As a second argument, the `user_data` pointer is passed to the callback. This pointer may reference any state the continuation may require for its execution.

In addition to requests, statuses, the callback function pointer, and the user-provided state, the two functions listed in Listing 1 also accept a set of OR-combined `flags` that control different aspects of the continuation. Among these flags is `MPI_CONT_IMMEDIATE` to control whether the continuation may be executed immediately if all operations have completed already. If that flag is not set, the continuation will be enqueued for later execution, e.g., when waiting on the continuation request. However, the details of these flags are still fluid and beyond the scope of this paper and not relevant for the ensuing discussion.

As a last argument, the *continuation request* described in Section 2.2 is passed to the attaching functions.

Figure 2 shows the flow of ownership in the current API design. The call to `MPI_Isend` allocates a request object and passes its ownership back to the caller (who *borrows* it), who is then responsible for releasing that request in a call to `MPI_Test` or `MPI_Wait`. If the request is passed to `MPI_Continue`, its ownership is transferred back to the MPI library, who is then responsible for releasing the associated internal resources. If a status argument other than `MPI_STATUS_IGNORE` is provided, the ownership of the status buffer is transferred to MPI and the application should not modify the buffer before the continuation is invoked, which implies the

```c
typedef void (MPI_Continue_cb_function)(MPI_Status *statuses,
                                        void *user_data);
```

(a) Callback signature.

```c
int MPI_Continue(
  MPI_Request *op_req,
  MPI_Continue_cb_funtion *cb,
  void *user_data,
  int flags,
  MPI_Status *status,
  MPI_Request cont_req);
```

(b) Attaching to single operation.

```c
int MPI_Continueall(
  int count,
  MPI_Request op_req[],
  MPI_Continue_cb_funtion *cb,
  void *user_data,
  int flags,
  MPI_Status statuses[],
  MPI_Request cont_req);
```

(c) Attaching to multiple operations.

Listing 1: API for attaching a continuation to a single or multiple MPI operations.

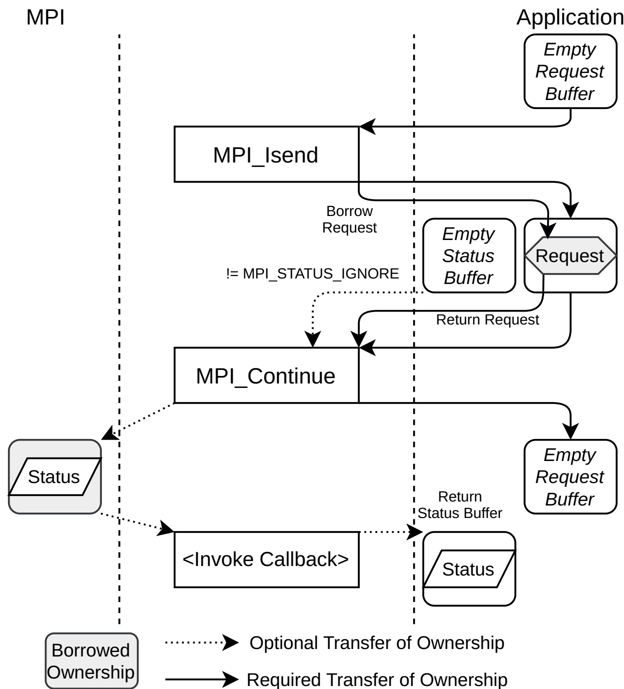


Fig. 2: Flow of ownership if passing a user-provided array of statuses to the continuation.

transfer of ownership of that buffer back to the application. While transient in nature, ownership of the request buffer is transferred into `MPI_Isend` and implicitly returned at the end of the call. We have included these transient ownership transfers for the sake of completeness.

We note that if `MPI_STATUS_IGNORE` is provided instead of a status buffer the only object(s) whose ownership is transferred are the requests. In that case, no borrowed ownership remain after the call to `MPI_Continue`.

3 Callback interface requirements

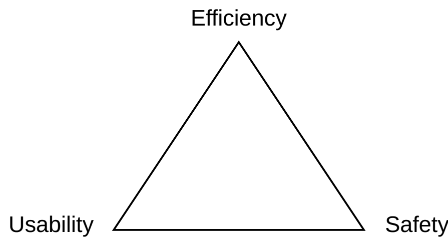


Fig. 3: Usability, efficiency, and safety are often detrimental in the design of APIs, requiring carefully balancing of these three requirements.

We outline three main requirements for the continuations API that we believe should be fundamental to the design of the continuations API. As shown in Figure 3, requirements for a safe, efficient, and easily usable API are often detrimental and need careful balancing.

3.1 Efficiency

The complexity of polling for the completion of requests using existing mechanisms such as `MPI_Testall` and `MPI_Waitall` involves checking the status of each request and progressing communication if required. MPI implementations have been carefully optimized to avoid dynamic memory management in such critical execution paths.

The cost of attaching a continuation to a set of requests and managing its execution should be equally low. In particular, requiring memory allocations that are not strictly necessary and copying objects (e.g., requests and statuses) should be avoided wherever possible. Ideally, no dynamic memory management would be required on

the part of the application, at least in the simplest of use-cases. Similarly, requiring the allocation of buffers inside MPI to hold requests or statuses in the design of the API would negatively impact performance even for simple cases.

3.2 Safety

While APIs for the C language rarely can eliminate all possible mistakes made by programmers, good API design aims at minimizing complexities and reducing the probability of such mistakes. In the context of asynchronous execution APIs such as continuations, likely sources of errors are accessing memory in the callback that points to the stack of the function that started the operation and attached the continuation, e.g., trying to access the request or status objects. Ideally, the MPI Continuations API helps users avoid the pitfalls of memory lifetime issues by eliminating disambiguities about object lifetime and ownership.

3.3 Usability

While a clean interface with little or no potential pitfalls certainly contributes to the usability of an interface, some simplifications in the API may require additional steps to achieve complex setups, e.g., management of additional memory (with a potential impact on performance) or set up of custom data structures and the resulting additional code that has to be written and maintained. On the other hand, a complex callback design providing a rich set of information (request handles, status objects, datatypes, message element counts) directly to the callback function may reduce the work on the part of the application since all relevant information is provided directly. However, most application may not need the provided information in their callbacks, resulting in overhead in memory space and time that does not yield any benefits for these applications.

4 Callback interface variations

We will discuss a set of variations in the design of the current API described in Section 2.3, using a simple example

Using the current API, Listing 2 provides an example of attaching a continuation to a nonblocking receive operation. All the continuation does is to enqueue a task that will process the message and release the buffer. The buffer is not processed directly in order to keep the duration of the callback as short as possible and to potentially defer the processing of the message to another thread. There is no use of the `status` provided when attaching the continuation and the message buffer is passed directly

```
1  /* Continuation request, initialized elsewhere */
2  MPI_Request cont_req;
3
4  void complete_cb(MPI_Status *status, void *buffer) {
5    enqueue_processing_task(buffer);
6  }
7
8  void start_receive(void *buffer, int from, int size){
9    MPI_Request op_req;
10   MPI_Irecv(buffer, size, MPI_BYTE, from, /*tag=*/101,
11             MPI_COMM_WORLD, &op_req);
12   MPI_Continue(&op_req, &complete_cb, buffer, 0,
13                MPI_STATUS_IGNORE, cont_req);
14 }
```

Listing 2: Simple example of a continuation attached to a nonblocking receive.

```
1  /* Continuation request, initialized elsewhere */
2  MPI_Request cont_req;
3
4  void complete_cb(MPI_Status *status, void *buffer) {
5    int msg_size;
6    MPI_Get_count(status,MPI_BYTE, &msg_size);
7    enqueue_processing_task(buffer, msg_size);
8    free(status);
9  }
10
11 void start_receive(void *buffer, int from, int buffer_size){
12   MPI_Request op_req;
13   MPI_Status *status = malloc(sizeof(MPI_Status));
14   MPI_Irecv(buffer, buffer_size, MPI_BYTE, from, /*tag=*/101,
15             MPI_COMM_WORLD, &op_req);
16   MPI_Continue(&op_req, &complete_cb, buffer, 0,
17                status, cont_req);
18 }
```

Listing 3: Simple example of a continuation attached to a nonblocking receive operation, querying the status of the operation.

on to the continuation. The value provided for the `status` parameter of the callback will be `MPI_STATUS_IGNORE`. No dynamic has to be allocated in this example. We note that the `cont_req` used in this and the following examples would have been initialized at an earlier point.

Listing 3 provides a variation of this example where `start_receive` posts a receive for a message with a maximum size and uses the status of the operation to query the size of the message actually received. The status is allocated on the heap (using `malloc` in Line 13) to ensure that the memory remains valid until the continuation has executed. The allocated status is subsequently freed in Line 8.

A more complex example employing a persistent receive operation is provided in Listing 4. When attaching the continuation, a status is passed that will be set before the callback is invoked. Like before, the status is allocated on the heap. Instead of

```
1  /* Wrapper around data needed in the callback */
2  typedef struct callback_data_t {
3    MPI_Request op_req; /* persistent operation request */
4    void *msg;          /* message buffer */
5  } callback_data_t;
6
7  void complete_cb(MPI_Status *status, void *user_data) {
8    int cancelled;
9    int msg_size;
10   MPI_Test_cancelled(status, &cancelled);
11   if (cancelled) { /* nothing to be done */
12     free(user_data);
13     free(status);
14     return;
15   }
16   MPI_Get_count(status,MPI_BYTE, &msg_size);
17   /* copy the message and restart the receive */
18   callback_data_t* cb_data = (callback_data_t*)user_data;
19   copy_msg_and_enqueue_task(cb_data->msg, msg_size);
20   MPI_Start(&op_req->op_req);
21   MPI_Continue(&op_req->op_req, &complete_cb,
22                user_data, 0, status, cont_req);
23 }
24
25 MPI_Request
26 start_recurring_receive(void *buffer, int from, int size) {
27   /* Allocate the callback data */
28   callback_data_t *cbdata = malloc(sizeof(callback_data_t));
29   /* Allocate the status object */
30   MPI_Status *status = malloc(sizeof(MPI_Status));
31   cbdata->msg = buffer;
32   MPI_Recv_init(buffer, size, MPI_BYTE, from, /*tag=*/101,
33                 MPI_COMM_WORLD, &cbdata->op_req);
34   /* start the operation and attach continuation */
35   MPI_Start(&cbdata->op_req);
36   MPI_Continue(&op_req->op_req, &complete_cb,
37                cbdata, 0, status, cont_req);
38   return op_req->op_req;
39 }
40
41 void stop_recurring_receive(MPI_Request op_req) {
42   MPI_Cancel(&op_req);
43   MPI_Request_free(&op_req);
44 }
```

Listing 4: A more complex example attaching a continuation to a persistent receive. An incoming message will be copied and a task processing it enqueued. The persistent receive is then restarted before the continuation is attached anew. Eventually, the persistent receive will be canceled, which will be detected inside the continuation in Lines 8–14.

just passing the message pointer, this time a structure of type callback_data_t is allocated that wraps the pointer to the message and the persistent operation request, both of which are accessed inside the continuation callback. In contrast to C++ lambda captures, such capturing has to be done manually in C.

The start-attach cycle is broken once the persistent receive is cancelled (Lines 41–44) and the check of the status in Line 10 detects the cancellation. The heap memory is released and no task is enqueued to process the message (Lines 12 – 13).

It is not hard to see that the current variant is neither fool-proof nor the most efficient solution. A subtle change to the way the status is allocated can lead to disastrous consequences. If instead of allocating the status on the heap the status was allocated on the stack (as is commonly the case when calling MPI_Test), the memory pointed to by status in the callback is invalid as it points to the stack of start_recurring_receive that is no longer active. Changing the code of Listing 4 accordingly, yields

```
MPI_Status status;
...
MPI_Continue(&op_req->op_req, &complete_cb,
             cbdata, 0, &status, cont_req);
```

Unfortunately, this rather subtle bug is not easy to spot and in practice would likely slip through a code review. It is not unnatural for users to believe that the status argument of the callback points to a status object provided by MPI, instead of simply being the status pointer provided while attaching the continuation. Thus, this is a potential source of grave errors that (as all bugs related to memory management) would be hard to debug.

In terms of efficiency, it is questionable why the status should be allocated separately. It would indeed be more efficient to allocate the status as part of the callback_data_t structure. However, since a pointer to that structure is already passed to the callback, passing a separate pointer to the callback function seems superfluous. In essence, the status pointer has to be stored and passed twice. Consequently, the current interface breaks with some of the requirements laid out in Section 3, both potentially impairing safety and efficiency.

4.1 Passing requests and user data

Instead of passing the pointer to the status object, it might be tempting to provide request (or array of requests) to the callback and query their status using MPI_Request_get_status. After all, unlike the status argument the (array of) request(s) is a non-optional parameter to MPI_Continue and MPI_Continueall. This would remove the status from the continuations interface altogether and avoid the potential access out-of-scope stack memory from within the continuation callback.

In principle, two sub-variants of this approach are possible.

4.1.1 MPI-provided Request Buffer

The first sub-variant is to allocate an MPI-internal buffer and copy the request or requests passed to `MPI_Continue` or `MPI_Continueall` into it. The ownership of non-persistent requests would still be transferred back to MPI and their handle be replaced by `MPI_REQUEST_NULL` in order to make them inaccessible outside of the continuation callback. This buffer of copied request handles would then be passed into the continuation callback and (together with all non-persistent requests) destroyed after the continuation completes. The flow of ownership is depicted in Figure 4.

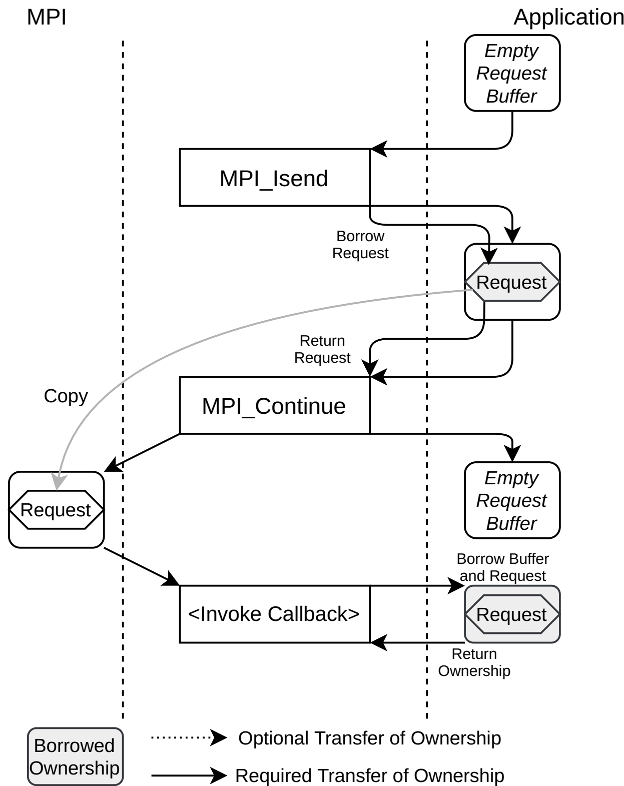


Fig. 4: Flow of ownership if passing an array of copied request handles to the continuation.

A significant drawback of this approach is the required copying of requests and additional dynamic memory management inside the MPI library since the number of requests to which a continuation is attached is not known *a priori*.

4.1.2 User-provided Request Buffer

Instead of allocating a buffer inside the MPI implementation, the API could also directly pass on the pointer to the request(s) provided to `MPI_Continue` or `MPI_Continueall`. As stated in Section 2.3, the ownership of non-persistent requests is returned to MPI. In this case, if the application wanted to access the status of an operation, it would have to request that the request handles are retained. This could be accomplished by introducing and passing a flag such as `MPI_CONT_RETAIN`, requesting that even non-persistent requests are retained until the continuation is invoked. The flow of ownership in this case is depicted in Figure 5.

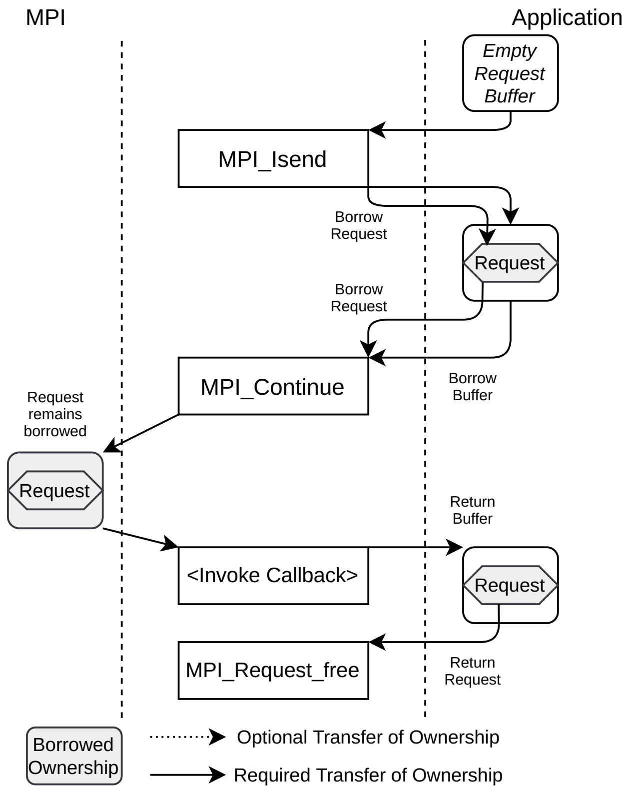


Fig. 5: Flow of ownership if passing the user-provided array of requests to the continuation.

Since ownership would remain with the application, it is necessary to either implicitly (at the end of the continuation) or explicitly return ownership at the end of the callback by invoking `MPI_Request_free` on each non-persistent request. In the interest of efficiency (and symmetry with test and wait functions), the addition of `MPI_Request_freeall` should be considered in this case. For continuations that

do not inquire the status of the operation, another flag should be introduced that prevents the implementation from storing and passing on the pointer to the request. This is especially important for requests that were located on the stack, as is the case in Listings 2 and 3. However, a new handle will have to be introduced to pass as an invalid pointer to a request handle.[1]

It becomes apparent that passing the request instead of the status into the continuation breaks with the requirements outlined in Section 3. Either the MPI library is required to allocate internal memory for each continuation, or the issue of pointers potentially pointing to invalid (stack) memory is shifted from the status object to the request objects. On top of that, the added complexity of properly managing the lifetime of requests through flags and additional release of requests opens the door for additional errors in user code and potentially impairs usability.

4.2 Passing only user data

To avoid the potential mistakes in the lifetime management of statuses and requests and the potential efficiency issues outlined in the previous sections, the state passed to the continuation should be reduced to a single pointer. This removes any ambiguity regarding the ownership of the status and request objects and avoids any additional memory allocations.

The code of Listing 4 adapted to only passing the user pointer is provided in Listing 5. It should be noted that while this interface removes potential issues around memory management (and thus provides improved safety and efficiency) a slightly higher burden is put on users in that all state of the continuation has to be collected in a single structure. However, we believe that this is a cost that is worth paying in exchange for the reduced potential of memory management mistakes and efficiency issues.

In order to further reduce the risk of using out-of-scope stack variables in continuations (e.g., from users allocating `callback_data_t` on the stack), the MPI Continuations interface would again have to copy the contents of the user-provided buffer into an internal buffer and pass that buffer to the callback. However, as stated earlier, this may compromise efficiency and safety and does not guarantee that nested pointers do not point to variables on the out-of-scope stack.

With this interface, the simple code in Listing 2 will remain the same, except that the `MPI_Status*` argument to the callback disappears. No dynamic memory allocation would be required in this case. The code in Listing 3 will have to allocate a structure containing the pointer to the message buffer and the status. The allocation is thus shifted from the status object to the user-provided data pointer.

[1] MPI does not typically employ the `NULL` pointer but instead defines special values for all invalid handles.

```
1  typedef struct callback_data_t {
2    MPI_Request op_req; /* persistent operation request */
3    MPI_Status status; /* status of the operation */
4    void *msg; /* the message to be received */
5  } callback_data_t;
6
7  void complete_cb(void *user_data) {
8    callback_data_t* cbdata = (callback_data_t*)user_data;
9    int cancelled;
10   MPI_Test_cancelled(&cbdata->status, &cancelled);
11   if (cancelled) { /* nothing to be done */
12     free(cbdata);
13     return;
14   }
15   enqueue_process_process(cbdata->msg);
16   MPI_Start(&op_req->op_req);
17   MPI_Continue(&op_req->op_req, &complete_cb,
18                cbdata, 0, status, cont_req);
19 }
20
21 MPI_Request
22 start_recurring_receive(void *buffer, int from, int size) {
23   callback_data_t *cbdata = malloc(sizeof(callback_data_t));
24   cbdata->msg = buffer;
25   MPI_Recv_init(buffer, size, MPI_BYTE, from, /*tag=*/101,
26                 MPI_COMM_WORLD, &cbdata->op_req);
27   MPI_Start(&cbdata->op_req);
28   MPI_Continue(&op_req->op_req, &complete_cb,
29                cbdata, 0, &cbdata->status, cont_req);
30   return cbdata->op_req;
31 }
```

Listing 5: The example of Listing 4 passing the user data pointer as the only state to the continuation callback.

5 C++ lambda capture

A variation of the code of Listing 5 using C++ lambda captures is listed in Listing 6. In this case, the compiler captures all data necessary inside the lambda defined in Lines 30 – 35. Unfortunately, the `status` of the operation cannot be automatically captured by the lambda because it's values is known only once the callback is invoked. Thus, the status is encapsulated inside a wrapper `cb_t` that makes it accessible both inside and outside the lambda.

All other variables are captured *by value* (including the operation request) and stored as part of the `fn` member of `cb_t`. The lambda is marked as `mutable` because both `MPI_Start` and `MPI_Continue` take a non-const pointer to it. The static `invoke` member function of `cb_t` (Lines 10 – 19) will be called by MPI, which then checks for cancellation and invokes the lambda, passing a reference to itself, allowing the lambda to reattach the continuation using the same object.

```
1  /* Callback wrapper typed on the callable's type */
2  template<typename Fn>
3  struct cb_t {
4    /* Status must be accessible outside the callable */
5    MPI_Status status;
6    Fn fn;
7    cb_t(Fn&& fn) : fn(std::forward<Fn>(fn)) {}
8    /* static function invoked from MPI,
9       dispatching to the provided callable */
10   static void invoke(void*data) {
11     cb_t* cb = static_cast<cb_t*>(data);
12     int cancelled;
13     MPI_Test_cancelled(&cb.status, &cancelled);
14     if (cancelled) {
15       delete cb; /* cleanup the wrapper */
16       return;
17     }
18     cb->fn(*cb);
19   }
20 };
21
22 MPI_Request
23 start_recurring_receive(void *buffer, int from, int size) {
24   MPI_Request op_req;
25   MPI_Recv_init(buffer, size, MPI_BYTE, from, /*tag=*/101,
26                 MPI_COMM_WORLD, &op_req);
27   MPI_Start(&op_req);
28   auto cb = new cb_t(
29     /* Marked mutable to pass op_req as non-const. */
30     [=](auto& cb) mutable {
31       process(buffer);
32       MPI_Start(&op_req);
33       MPI_Continue(&op_req, &cb.invoke,
34                    &cb, 0, &cb.status, cont_req);
35     });
36   MPI_Continue(&op_req, &cb->invoke,
37                cb, 0, &cb->status, cont_req);
38   return op_req;
39 }
```

Listing 6: The example of Listing 4 using C++ lambda capture. The `status` must be accessible inside and outside the lambda expression and thus cannot be captured. Instead it is held in a wrapper object through which the lambda is invoked.

We have deliberately avoided the use `std::function` in order to provide the compiler with the opportunity to inline the code in the lambda, further reducing the overhead of the call. While the use of `std::function` would remove the template parameter `Fn` from `cb_t`, it introduces a second indirect call in the continuation (in addition to the indirect call to `invoke` inside the MPI library).

We note that the templating of `cb_t` makes it easily composable and reusable, allowing it to be used with different lambdas throughout the an application. While not entirely void of complexity, the use of C++ lambda captures with MPI Continuations removes the hassle of manually transferring data from the callsite into the callback through custom structures, as is required when using C.

6 Conclusions

In this paper, we have discussed several variants of the MPI Continuations API and how state relevant to the execution of the continuation callback can be captured and passed to the callback. We set out a three requirements (usability, efficiency, and safety) that at times are at odds with each other. We found that the interface currently proposed encourages inefficient (by separately allocating the status object(s)) and potentially unsafe (by passing stack-based variables) code. We believe that an interface that requires the aggregation of all variables accessed inside the continuation callback into a single structure yields a safer and potentially more efficient API design. We have also shown that by employing modern C++ lambda captures, this task can be mostly automated. Based on our findings, we will adapt the interface in our MPI Continuations proposal to only pass a single state pointer and to avoid potential confusions about lifetime and ownership of status objects present in the current proposal.

References

1. N. Gustafsson, A. Laksberg, H. Sutter and S. Mithani. N3857: Improvements to `std::future`<T> and Related APIs. Tech. Rep. N3857 (2014).
2. H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio and D. Fey. HPX: A Task Based Programming Model in a Global Address Space. In: *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pp. 6:1–6:11, ACM (2014). DOI 10.1145/2676870.2676883
3. H. Lu, S. Seo and P. Balaji. MPI+ULT: Overlapping Communication and Computation with User-Level Threads. In: *2015 IEEE 17th International Conference on High Performance Computing and Communications, 2015 IEEE 7th International Symposium on Cyberspace Safety and Security, and 2015 IEEE 12th International Conference on Embedded Software and Systems* (2015). DOI 10.1109/HPCC-CSS-ICESS.2015.82
4. MPI: A Message-Passing Interface Standard, Version 4.0. Tech. rep. (2021). URL https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf
5. J. Protze, M.A. Hermanns, A. Demiralp, M.S. Müller and T. Kuhlen. MPI Detach – Asynchronous Local Completion. In: *27th European MPI Users' Group Meeting*, EuroMPI/USA '20, Association for Computing Machinery (2020). DOI 10.1145/3416315.3416323
6. K. Sala, X. Teruel, J.M. Perez, A.J. Peña, V. Beltran and J. Labarta. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Computing* **85**, 153–166 (2019). DOI 10.1016/j.parco.2018.12.008

7. J. Schuchart, C. Niethammer and J. Gracia. Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations. In: *27th European MPI Users' Group Meeting* EuroMPI/USA '20, pp. 39–50, Association for Computing Machinery (2020). DOI 10.1145/3416315.3416320
8. J. Schuchart, P. Samfass, C. Niethammer, J. Gracia and G. Bosilca. *Parallel Computing* **106**, 102793 (2021). https://doi.org/10.1016/j.parco.2021.102793. URL https://www.sciencedirect.com/science/article/pii/S0167819121000466
9. J. Schuchart, K. Tsugane, J. Gracia and M. Sato. The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In: *Evolving OpenMP for Evolving Architectures* (Springer International Publishing, 2018), pp. 3–17. DOI 10.1007/978-3-319-98521-3_1. Awarded Best Paper
10. Y. Zheng, A. Kamil, M.B. Driscoll, H. Shan and K. Yelick. UPC++: A PGAS Extension for C++. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 1105–1114 (2014). DOI 10.1109/IPDPS.2014.115