Parallel Computing 106 (2021) 102793

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Callback-based completion notification using MPI Continuations

Joseph Schuchart [a,b,*], Philipp Samfass [c], Christoph Niethammer [b], José Gracia [b], George Bosilca [a]

[a] Innovative Computing Laboratory (ICL), University of Tennessee Knoxville (UTK), 1122 Volunteer Blvd, Knoxville, TN, 37996, USA
[b] High-Performance Computing Center Stuttgart (HLRS), Nobelstraße 19, 70597, Stuttgart, Germany
[c] Department of Informatics, Technical University of Munich, Boltzmannstr. 3, 85748, Garching (Munich), Germany

ARTICLE INFO

Keywords:
MPI
MPI Continuations
Task-based programming models
OpenMP
OmpSs
TAMPI
PaRSEC

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 APMs as correctly and efficiently handling MPI communication in different 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 present an extension to the previously described interface that allows for finer control of the behavior of the MPI Continuations interface. We then present some of our first experiences in using the interface in the context of different applications, including the NAS parallel benchmarks, the PaRSEC task-based runtime system, and a load-balancing scheme within an adaptive mesh refinement solver called ExaHyPE. We show that the interface, implemented inside Open MPI, enables low-latency, high-throughput completion notifications that outperform solutions implemented in the application space.

1. Background and motivation

Asynchronous (task-based) programming models are gaining more and more traction, promising to help users better utilize ubiquitous multi- and many-core systems by exposing all available concurrency to a runtime system. Applications are expressed in terms of work-packages (tasks) with well-defined inputs and outputs, which guide the runtime scheduler in determining the correct execution order of tasks. A wide range of task-based programming models have been developed, ranging from node-local approaches such as OpenMP [1] and OmpSs [2] to distributed systems such as HPX [3], StarPU [4], DASH [5], and PaRSEC [6]. All of them have in common that a set of tasks is scheduled for execution onto a set of worker threads based on constraints provided by the user, either in the form of dependencies or dataflow expressions.

At the same time, MPI is still the dominant interface for inter-process communication in parallel applications [7], providing blocking and non-blocking point-to-point and collective operations as well as I/O capabilities on top of elaborate datatype and process group abstractions [8]. Recent years have seen significant improvements to the way MPI implementations handle communication by multiple threads of execution in parallel [9,10].

However, task-based programming models pose new challenges to the way applications interact with MPI. While MPI provides non-blocking communications that can be used to hide communication latencies, it is left to the application layer to ensure that all necessary communication operations are eventually initiated to avoid deadlocks, that in-flight communication operations are eventually completed, and that the interactions between communicating tasks are correctly handled. Higher-level distributed tasking approaches such as PaRSEC or HPX commonly track the state of active communication operations and regularly test for completion before acting upon such a change in state. With node-local programming models such as OpenMP, this tracking of active communication operations is left to the application layer. Unfortunately, the simple approach of test-yield cycles inside a task does not provide optimal performance due to CPU cycles being wasted on testing and (in the case of OpenMP) may not even be portable [11].

MPI is thus currently not well equipped to support users and developers of asynchronous programming models, which (among other factors) has prompted some higher level runtimes to move away from MPI towards more low-level, asynchronous APIs [12].

Different approaches have been proposed to mitigate this burden, including Task-Aware MPI (TAMPI) [13] and a tight integration of fiber

* Corresponding author at: Innovative Computing Laboratory (ICL), University of Tennessee Knoxville (UTK), 1122 Volunteer Blvd, Knoxville, TN, 37996, USA.
E-mail addresses: schuchart@icl.utk.edu (J. Schuchart), samfass@in.tum.de (P. Samfass), niethammer@hlrs.de (C. Niethammer), gracia@hlrs.de (J. Gracia), bosilca@icl.utk.edu (G. Bosilca).

https://doi.org/10.1016/j.parco.2021.102793
Received 16 February 2021; Accepted 30 April 2021
Available online 17 May 2021

and tasking libraries with MPI implementations [14–16]. In a previous paper, we have argued that such an integration, while seemingly attractive due to its ease-of-use, fosters the development of a non-portable application eco-system [17]. In the same paper, we have proposed an extension to the MPI interface, called *MPI Continuations*, that is designed to loosely couple asynchronous programming models with MPI. In the current paper, we update and extend the description of this interface (Sections 2 and 3). We compare our approach to the MPI_T interface, a general-purpose API using callbacks for event notification that will be part of the upcoming MPI 4.0 standard (Section 4). We will demonstrate the integration of continuations with the PaRSEC runtime and with a adaptive load balancing scheme based on Intel TBB (Section 5).

2. MPI continuations: API overview

Continuations are a concept for structuring the execution of different parts of an application's code, dating back to research on Algol60 [18,19]. They have recently been proposed to the C++ standard in the form of `std::future::then` to coordinate asynchronous activities [20]. Continuations consist of a *body* (the code to execute) and a *context* (some state passed to the body) on which to operate.

A similar mechanism for MPI can be devised that allows a callback to be attached to an MPI request, which will be invoked once the operation represented by the request has completed. This establishes a notification scheme to be used by applications to timely react to the completion of operations, e.g., to wake up a blocked thread or release the dependencies of a detached task, all while relieving applications of managing MPI request objects themselves.

We propose a set of functions to set up continuations for active MPI operations, which may be invoked by application threads during calls to communication-related MPI functions or by an implementation-internal progress thread (if available) once all relevant operations are found to have completed. The body of a continuation may call any MPI library function and thus start new MPI operations in response to the completion of previous ones, e.g., in order to re-post a completed receive.

The MPI Continuations API consists of two parts. First, *continuation requests* are used to aggregate and progress continuations. Second, continuations are attached to active MPI operations and registered with continuation requests for tracking.

2.1. Continuation requests

Continuation requests (CR) are a form of persistent requests that are created through a call to `MPIX_Continue_init` and released eventually using `MPI_Request_free`. A CR tracks a set of active continuations that are registered with it. The set grows upon registration of a new continuation and shrinks once a continuation has been executed. A call to `MPI_Test` on a CR returns `flag == 1` if no active continuations are registered. Conversely, a call to `MPI_Wait` blocks until all registered continuations have completed. Further details are discussed in Section 3.

Continuation requests serve two main purposes. First, they aggregate continuations attached to active operations and enable testing and waiting for their completion. Second, by calling `MPI_Test` on a CR, applications can progress outstanding operations and continuations if no MPI-internal progress mechanism exists to process them asynchronously. See Section 3.4 for a discussion on issues related to progress.

The `info` argument to `MPIX_Continue_init` may be used to control certain aspects of the continuations. A list of proposed keys will be discussed in Section 3.5.

```c
/* The continuation callback function definition */
typedef void (MPIX_Continue_cb_function)(
  MPI_Status *statuses,
  void       *cb_data);

/* Create a continuation request using the
   provided info controls */
int MPIX_Continue_init(
  MPI_Request  *cont_req,
  MPI_Info      info);

/* Attach a continuation to an active operation represen-
 * ted by op_request. Upon completion of the operation,
 * the callback cb will be invoked and passed status and
 * cb_data as arguments. The status object will be set
 * before the continuation is invoked. If the operation
 * has completed already the continuation will not
 * be attached or invoked, flag will be set to 1, and
 * the status will be set before return. */
int MPIX_Continue(
  MPI_Request               *op_request,
  int                       *flag,
  MPIX_Continue_cb_function *cb,
  void                      *cb_data,
  MPI_Status                *status,
  MPI_Request                cont_req);

/* Similar to the above except that the continuation
 * is invoked once all op_requests have completed. */
int MPIX_Continueall(
  int                        count,
  MPI_Request                op_requests[],
  int                       *flag,
  MPIX_Continue_cb_function *cb,
  void                      *cb_data,
  MPI_Status                 statuses[],
  MPI_Request                cont_req);
```

Listing 1: MPI Continuation interface.

2.2. Registration of continuations

A continuation is attached to one or several active operations and registered with the *continuation request* for tracking. A call to `MPIX_Continue` attaches a continuation to a single operation request while the use of `MPIX_Continueall` sets up the continuation to be invoked once *all* operations represented by the provided requests have completed. As shown in Listing 1, the continuation is represented through a callback function with the signature of `MPIX_Continue_cb_function` (provided as function pointer `cb`) and a context (`cb_data`). Together with the pointer value provided for the `status` or `statuses` argument, the `cb_data` will be passed to `cb` upon invocation.

Upon return, all provided non-persistent requests are set to `MPI_REQUEST_NULL`, effectively releasing them back to MPI.[1] No copy of the requests should be used to cancel or test/wait for the completion of the associated operations. Consequently, only one continuation may be attached to an operation request. In contrast, persistent requests may still be canceled, tested, and waited for.

Similar to `MPI_Test`, `flag` shall be set to `1` if all operations are already complete. In that case, the continuation callback shall *not* be invoked by MPI, leaving it to the application to handle immediate completion (see Section 3.1). Otherwise, `flag` is set to zero.

Unless `MPI_STATUS_IGNORE` or `MPI_STATUSES_IGNORE` is provided, the status object(s) will be set before the continuation is invoked (or before returning from `MPIX_Continue[all]` in case all operations

[1] This has been a deliberate design decision. Otherwise, the release of a request object inside the continuation or the MPI library would have to be synchronized with operations on it outside of the continuation. We thus avoid this source of errors.

```
1  bool poll_mpi(MPI_Request *cont_req) {
2    int flag; /* result stored in flag ignored here */
3    MPI_Test(&flag, cont_req, MPI_STATUS_IGNORE);
4    return false; /* we should be called again */
5  }
6  void continue_cb(MPI_Status *status, void *task) {
7    /* release the task's dependencies */
8    nanos6_decrease_task_event_counter(task, 1);
9  }
10 void solve(int NT, int num_fields,
11             field_t *fields[num_fields]) {
12   /* create continuation request */
13   MPI_Request cont_req;
14   MPIX_Continue_init(&cont_req, MPI_INFO_NULL);
15   /* register polling service */
16   nanos6_register_polling_service(
17       "MPI", &poll_mpi, &cont_req);
18
19   for (int timestep = 0; timestep < NT; timestep++) {
20     for (int i = 0; i < num_fields; ++i) {
21       #pragma oss task depend(out: fields[i])
22       {
23         /* unpack recv buffer, compute and pack
24          * send buffer */
25         integrate_halo(fields[i]);
26         solve(fields[i]);
27         save_boundary(fields[i]);
28
29         /* start send and recv */
30         MPI_Request reqs[2];
31         MPI_Isend(fields[i]->sendbuf, ..., &reqs[0]);
32         MPI_Irecv(fields[i]->recvbuf, ..., &reqs[1]);
33
34         /* detach task if requests are active */
35         void* task = nanos6_get_current_event_counter()↪
               ;
36         nanos6_increase_current_task_event_counter(
37             task, 1);
38         /* attach continuation */
39         int flag;
40         MPIX_Continueall(
41             2, reqs, &flag, &continue_cb, task,
42             MPI_STATUSES_IGNORE, cont_req);
43         if (flag)
44           nanos6_decrease_task_event_counter(task, 1);
45   } } }
46   /* wait for all tasks to complete and tear down */
47   #pragma oss taskwait
48   nanos6_unregister_polling_service(
49       "MPI", &poll_mpi, NULL);
50   MPI_Request_free(&cont_req);
51 }
```

Listing 2: Simplified example using MPI Continuations in an iterative solver. Tasks are created for each field per timestep. Communication is initiated and tasks are detached. As soon as the communication completes, the task's dependencies are released and the field's next iteration may be scheduled.

have completed already). The status objects should be allocated by the application in a memory location that remains valid until the callback has been invoked, e.g., on the stack of a blocked thread or fiber or in memory allocated on the heap that is released inside the continuation callback.

2.3. Usage in a simple example

The example provided in Listing 2 uses MPI Continuations to release detached tasks in an OmpSs-2 [21] application once communication operations have completed. Looping over all fields of a local domain in each timestep, one task is created per field, which carries an output dependency on its `field` object (Line 21). Inside each task, the received boundary data from the previous timestep is incorporated and a solver is applied to the field (potentially with nested tasks) before the local boundary is packed for sending (Lines 25–27).

Both the send and receive operations are initiated and a continuation is attached using `MPIX_Continueall` (Lines 30–44). OmpSs-2 offers direct access to API functions of the underlying `nanos6` runtime. In this case, the event counter for the current task is retrieved, increased, and passed as the context of the continuation. If all operation completed immediately, the event counter is decremented again.[2] Otherwise, the task will run to completion but its dependencies will not be released before the event counter is decremented in the continuation callback `continue_cb` (Lines 6–9).

In order to ensure that all continuations are eventually completed, a *polling service* is registered with the OmpSs-2 runtime (Line 16). This polling service is regularly invoked by the runtime and – in the function `poll_mpi` in Lines 1–5 – calls `MPI_Test` on the continuation request.[3] This will invoke all available continuations and eventually cause all tasks of the next timestep to become available for execution once the respective send and receive operations of the previous timestep have completed.

With only approximately 15 lines of code (including setup and tear-down), it is possible to integrate MPI communication with a task-parallel application to fully overlap communication and computation. If adopted into the MPI standard, this approach is fully portable and allows an application to focus on application-level concerns while leaving MPI-level concerns such as request management to MPI. By making the interaction between non-blocking MPI operations and the task scheduler explicit, this approach ensures that both MPI and the task programming system support all required operations (with the exception of strong progress guarantees, as discussed in Section 3.4). Moreover, we believe that integrating such an interface into MPI enables more efficient implementations as it allows direct access to the request structure instead of handling opaque request objects.

3. MPI continuations: Details

A scheme similar to the one proposed here could be implemented outside of the MPI library with an interface similar to TAMPI. However, a main advantage of the integration with MPI is that the continuations can be invoked as soon as *any* thread calls into MPI and determines the associated operations to be complete. In more complex applications, this allows for the invocation of continuations set up by one part of the application during an MPI call issued in another part of the application, potentially reducing the time-to-release of blocked tasks and thus preventing thread starvation.

3.1. Restrictions

As stated earlier, continuations may be executed by application threads while executing communication functions in MPI. Exceptions are `MPIX_Continue[all]`, which may be called from within a critical region protected by a mutex it would then attempt to acquire again inside the continuation. Listing 3 provides an example where the call to `MPIX_Continue` happens in a critical region guarded by a mutex, which is necessary to prevent signals from being lost due to the `unblock` callback being executed on another thread before the thread executing `block` started waiting in `pthread_cond_wait`.

While not prohibited, the use of blocking operations inside continuations should be avoided as it may cause a call to a nonblocking MPI procedure to block on an unrelated operation. No other continuation may be invoked in MPI calls made from within a continuation to avoid stack overflows due to deep nesting of continuation calls.

[2] The event counter has to be incremented first as the OmpSs-2 specification mandates that "the user is responsible for not fulfilling events that the target task has still not bound". [21, §4.4].

[3] Note that OpenMP does not currently provide such an interface, which requires the user to create a *progress task* or spawn a *progress thread* that yield after testing.

```
void block(thread_state_t *ts, MPI_Request *req) {
    int flag;
    pthread_mutex_lock(&ts->mtx);
    MPIX_Continue(&req, &flag, &unblock, ts, ↪
        MPI_STATUS_IGNORE, cont_req);
    if (!flag) pthread_cond_wait(&ts->cond, &ts->mtx);
    pthread_mutex_unlock(&ts->mtx);
}
int unblock(MPI_Status *status, void *data) {
    thread_state_t *ts = (thread_state_t *)data;
    pthread_mutex_lock(&ts->mtx);
    pthread_cond_signal(&ts->cond);
    pthread_mutex_unlock(&ts->mtx);
}
```

Listing 3: Code to block and unblock a POSIX thread. The `thread_state_t` structure contains a thread-specific mutex and conditional variable. Locking the mutex in `unblock` is necessary to prevent signals from getting lost.

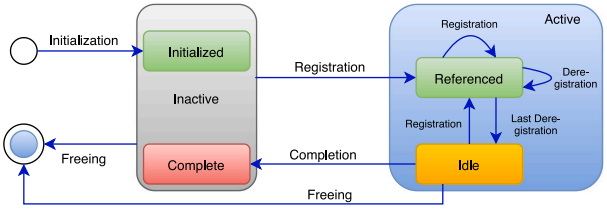


Fig. 1. State diagram of continuation requests.

By default, continuations may not be invoked from within signal handlers inside the MPI implementation as that would restrict the application to using only async-signal-safe operations. MPI implementation relying on signals to interact with the network hardware should defer the execution to the next invocation of an MPI procedure by any of the application threads or hand over to a progress thread. However, the application may signal that a callback is async-signal safe, as discussed in Section 3.5.

3.2. State transitions

Unlike existing request types in MPI, continuation requests (CR) do not represent a single operation but a set of operations. Extending the semantics of persistent requests [22], Fig. 1 depicts the state diagram of CRs. That state may change with every new registration or completion of a continuation. A newly *Initialized* or otherwise *Inactive* CR becomes *Active Referenced* when a continuation is registered. It remains *Active Referenced* if additional continuations are registered. Upon completion of continuations, they are deregistered from the CR. The CR becomes *Active Idle* upon the deregistration of the last active continuation. An *Active Idle* CR may become either *Active Referenced* again if a new continuation is registered or *Complete* if a *Completion* function such as `MPI_Test` is called on it. It is possible to call `MPI_Request_free` on an active CR, in which case the CR cannot be used to register additional continuations and will be released as soon as all previously registered continuations have completed.

A continuation may be attached to a CR itself and registered with a separate CR, allowing applications to build complex patterns of notifications. The new continuation will then be executed once all continuations registered with the first CR have completed.

3.3. Thread-safety requirements

Multiple threads may safely register continuations with the same continuation request in parallel but only one thread may test or wait for its completion at any given time. This allows for continuations to be registered concurrently without requiring mutual exclusion at the application level.

3.4. Progress requirements

The availability of continuations is only one piece of the puzzle to reliably and portably integrate MPI with task-based applications. The issue of progress in MPI is long-standing and while some implementations provide internal mechanisms for progressing communication (e.g., progress threads) this is not behavior users can rely on in all circumstances. Thus, applications are still required to call into MPI to progress outstanding communication operations and – with the help of the proposed interface – invoke available continuation callbacks. This proposal does not change the status quo in that regard.

It is thus left to the application layer to make sure that communication is progressing. This can be achieved by regularly testing the continuation request, which could be done either through a recurring task, a polling service registered with the task runtime system (as shown for OmpSs-2 in Listing 2), or by relying on a progress thread inside MPI. The interface presented here is flexible enough to allow for its use in a wide range of circumstances and it defers to the application layer to choose the right strategy based on the available capabilities.

3.5. Controlling behavior through the `info` object

The `MPIX_Continue_init` function takes an `info` object that may be used to control certain aspects of the continuation request and the registered callbacks, respectively. We propose the following info keys for controlling continuation request *behavior*, which have proven useful during our experiments:

`mpi_continue_poll_only` Continuations registered with a CR created with this key set to `true` will only be executed once the CR is tested for completion, e.g., using `MPI_Test`. This may be useful for "heavy" callbacks or callbacks that may block the executing thread in order to not disturb the execution of other parts of the application, e.g., by blocking threads used by another library. This info key provides control over the point at which callbacks are executed. The default is `false`.

`mpi_continue_enqueue_complete` If this info key is set, then a call to `MPIX_Continue[all]` will always return `flag = 0` even if the operation has completed immediately. The continuation will then be enqueued for later execution. This option may be useful for applications expecting large numbers of incoming messages with only limited time for handling them, instead choosing to defer their reception to a later point. Without this key, the caller would have to have a mechanism for deferring the handling of completed operations itself. 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 application may then choose to limit the number of continuations to be handled at once in order to guarantee progress on outgoing messages as well. The default is `-1`, meaning unlimited. Setting both `mpi_continue_max_poll = 0` and `mpi_continue_poll_only = true` is erroneous as it would cause no continuation registered with this continuation request to ever be executed.

In addition to the info keys described above, we also propose two info keys controlling the *context* in which the implementation may execute the registered continuations:

```c
void completion_cb(MPI_Status *status, void *data) {
  int is_cancelled;
  MPI_Test_cancelled(status, &is_cancelled);
  if (is_cancelled) {
    return;
  }
  /* perform regular actions of the callback otherwise */
  ...
}
```

Listing 4: Callback checking for cancellation of the request it is attached to.

`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. Some applications may rely on thread-local data being initialized outside of the continuations callback or use callbacks that are not thread-safe, in which case the use of `any` would lead to correctness issues. This key has no effect on implementations that do not use a dedicated progress thread.

`mpi_continue_async_signal_safe` If this Boolean 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`.

The list of info keys discussed above reflects the different dimensions in which applications may want to control the behavior of the continuations interface. The `info` mechanism in MPI provides an ideal mechanism for controlling these different aspects, allowing users to adapt its behavior to their needs.

3.6. Request cancellation

MPI provides the ability to cancel outstanding MPI operations using the `MPI_Cancel` procedure. While cancellation of send operations has been deprecated, it is still possible (and at times useful) to cancel requests representing receive operations, e.g., to clean-up pre-posted receive operations at the end of an application run. Thus, applications need to be able to cope with canceled requests in continuation callbacks. This can be achieved by allocating and passing status objects in calls to `MPI_Continue[all]` and check for cancellation in the callback function, as demonstrated in Listing 4. This alleviates the need for explicit callbacks notifying about the cancellation of requests.

4. Comparison with the MPI_T interface

The MPI_T API will provide an interface for callback-based event notification in the upcoming version 4 of the MPI standard [23]. This interface is meant for performance analysis tools to be notified about events on arbitrary MPI objects, e.g., communicators or window. It would thus be possible to track the completion of operations using events on the respective communicators. However, this would require the application (or some library) to keep a mapping between requests and the asynchronous activity related to them, which would be tedious and error-prone.

Since the interface is not limited to coarse-grain MPI objects such as windows and communicators, this interface could also be utilized to register completion events on individual requests. However, this approach has several caveats that will be discussed below.

4.1. Overhead

In order to register a callback with individual operation requests using MPI_T events, the user has to use three distinct function calls. In a first step, an event handle is allocated for the specific request object using `MPI_T_EVENT_HANDLE_ALLOC`. This binds a certain event (chosen from a set of events provided by the MPI implementation) to the request object and provides an event handle. In a second step, a callback is registered with that event-registration object by calling `MPI_T_EVENT_REGISTER_CALLBACK`. Eventually, the event handle has to be released using `MPI_T_EVENT_HANDLE_FREE`. For non-persistent requests, this has to be done during every completion callback to avoid memory leaks caused by dangling event handles.

Altogether, these three API calls are likely to incur a significant overhead for the registration of callbacks. Unfortunately, the authors do not yet have access to a working implementation of the MPI_T callback interface to test this hypothesis, which remains as future work.

4.2. Reliability

In its current form, the MPI_T interface poses two major correctness challenges for it to be used for callback-based completion notification. First, the current definition allows implementations to drop events if it finds that too many events have occurred. While this may be acceptable for performance analysis (e.g., in the context of profiling tools) it is highly problematic for applications whose correctness relies on callbacks being delivered for each operation request. Even though it is possible to be notified about the number of dropped events, the information on the particular completion will be lost.

Second, an inherent race condition exists in this interface since an operation may have completed between the time it was issued and the time the event callback has been registered. It is unclear if the implementation would still be required to deliver the event to the application. At the same time, the restrictions on immediate execution of continuations laid out in Section 3.1 would have to be obeyed.

4.3. Portability and usability

Currently, the MPI_T interface does not provide any predefined events, making it impossible to rely on certain events for correctness. While in the future the MPI standard may choose to mandate support for events for operation request completion, its detection would still be potentially tedious due to the weak binding using string comparisons. In contrast, the interface proposed in this work provides a well-defined interface that applications can rely on.

However, mandating the existence of a specific event on requests with rather restrictive semantics seems to run counter to the spirit of the MPI_T interface, which is to provide freedom to implementors to define events and their specific semantics.

Moreover, the MPI_T interface does not provide the full capabilities of the continuations interface proposed in this work. For example, in contrast to `MPIX_Continueall` it is not possible to register callbacks for groups of requests. Moreover, the MPI_T interface does not provide a way to explicitly progress outstanding events and test or wait for outstanding registered callbacks, which our proposal accomplishes through continuation requests.

Last but not least, the MPI_T interface includes a hierarchy of safety levels for callbacks, which comprises `MPI_T_CB_REQUIRE_NONE` (no restrictions on the set of MPI functions that may be called inside the callback), `MPI_T_CB_REQUIRE_MPI_RESTRICTED` (set of allowed MPI functions restricted to MPI_T), `MPI_T_CB_REQUIRE_THREAD_SAFE` (like the previous level and the callback must be thread-safe), and `MPI_T_CB_REQUIRE_ASYNC_SIGNAL_SAFE` (the callback must be async-signal safe). This hierarchy, while useful for tools using the MPI_T interface, does not provide sufficient control for applications. Most importantly, this hierarchy does not allow for the definition of callbacks

Table 1
Software configuration.

Software	Version	Configuration/Remarks
Open MPI	`git-0dc2325`	`--with-ucx=...`
UCX	1.9.0	`--enable-mt`
GCC	10.2.0	*site installation*
OmpSs-2	2020.11.1	built as Clang plugin
Clang	`git-523fd9d03f`	includes fix for issue found in dependency handling
TAMPI	`git-c3bd734`	*default*

that are thread-safe and may invoke arbitrary MPI functions, which may be crucial for massively multi-threaded task-based applications employing this interface for task management purposes. In contrast, the interface proposed in this paper allows for fine-grain control of these two (orthogonal) aspects through info key–value pairs, as described in Section 3.5.

Overall, we believe that while the MPI_T interface uses a *similar* mechanism for the notification of arbitrary events, it is not suitable for the purpose of efficient operation completion notification as it only offers a subset of features the interface proposed here provides.

5. Evaluation

We will evaluate the use of continuations using the NPB BT-MZ benchmark (extended from earlier measurements presented in [17]), through an integration in PaRSEC, and an integration with a dynamic load-balancing scheme in ExaHyPE, an adaptive mesh refinement (AMR) package built on top of MPI and Intel Threading Building Blocks (TBB) [24].

All measurements were conducted on a Hewlett Packard Enterprise (HPE) Apollo system called *Hawk*, which is installed at the Center for High Performance Computing Stuttgart (HLRS) in Stuttgart, Germany.[4] The system is comprised of dual-socket nodes equipped with 128 GB of RAM and AMD EPYC 7742 CPUs with a nominal frequency of 2.25 GHz. The nodes are connected through a enhanced 9D torus using the Mellanox InfiniBand HDR200 interconnect. The configurations of the used software are listed in Table 1.

All data points represent the mean of at least five repetitions, with the standard deviation plotted as error bars in the graph.

5.1. Implementation

We have implemented a proof-of-concept (PoC) of continuations within Open MPI.[5] We have provided micro-benchmarks as part of the initial paper [17] and refer interested readers to it for details. Since then, we have added support for the info keys described in Section 3.5, which should only have a marginal impact on the overhead of continuation registration and execution. However, the integration into the MPI library has proven useful due to the direct access to the request object structures as we otherwise would have to build structures around the opaque request handles, potentially incurring additional overhead for their management.

5.2. NPB BT-MZ

In this section, we demonstrate the use of MPI Continuations in the context of the NAS Parallel Benchmark application BT-MZ, a multi-zone block tri-diagonal solver for the unsteady, compressible Navier Stokes equations on a three-dimensional mesh with two-dimensional domain decomposition [26]. Zones of different sizes are distributed across processes using a static load-balancing scheme, with the difference in zone sizes being up to 20× between the smallest and largest zones [27]. In each timestep and for each local zone, five computational steps are involved: forming the right-hand side, solving block-diagonal systems in each dimension `x`, `y`, and `z`, and updating the solution. We use two representative problem sizes – classes D and E – for which the input configurations are listed in Table 2. The reference implementation in Fortran uses OpenMP work-sharing constructs to parallelize nested loops during the updates to all local zones before collecting and exchanging all boundary data with neighboring ranks at the end of each timestep. OpenMP parallelization is done over the outermost loop, which in most cases is the smallest dimension `z`, with the notable exception of the solution in the `z` dimension itself.

Table 2
Problem sizes in the NPB BT-MZ benchmark [25].

Class	Zones	Iterations	Grid size ($x \times y \times z$)	Memory
D	32 × 32	250	1632 × 1216 × 34	12.8 GB
E	64 × 64	250	4224 × 3456 × 92	250 GB

We have ported the Fortran reference implementation to C++ and implemented two variations using task-based concurrency.[6] The first variation uses tasks to overlap the computation of zones within a timestep in a fork-join approach in between boundary updates, replacing OpenMP parallel loops with task-generating loops, with their execution coordinated using dependencies. This already allows for exploiting fine-grain concurrency within a single timestep.

The second variant extends this to also perform the boundary exchange inside tasks, including the necessary MPI communication, effectively minimizing the coupling between zones to the contact point dependencies. This variant requires some support from the tasking library to properly handle communicating tasks. We use TAMPI in combination with OmpSs-2 as well as detached tasks in OpenMP available from Clang in its current developer repository. We have attempted to use `taskyield` within OpenMP tasks using the Clang implementation, which resulted in deadlocks due to the restricted semantics of task-yield in OpenMP [11]. The implementation using detached tasks spawns a progress thread that tests the continuation request before calling `usleep` to yield the processor.

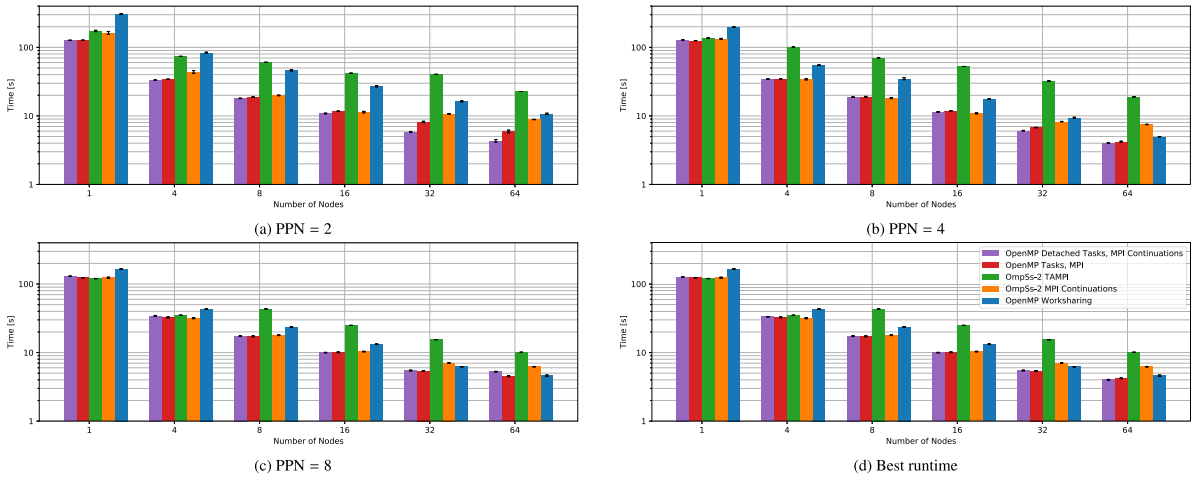
Given the limited concurrency in the OpenMP work-sharing loops, we present results achieved using 2, 4, or 8 processes per nodes (PPN; 64, 32, and 16 threads per process, respectively). All codes have been compiled with the flags `-O2 -march=znver2 -mcmodel=medium` using the Clang compiler. Process binding has been achieved using Open MPI's `--map-by node:PE=$numthreads --bind-to core` arguments to `mpirun`.

Class D. Fig. 2 depicts the results for class D using 2, 4, and 8 processes per node, as well as the best runtime from each of the configurations. Most notably, the OmpSs-2 variant using TAMPI yield significantly higher runtimes than the OpenMP variants, an effect that was reproduced multiple times for this problem size. It is, however, notable that the OmpSs-2 variant using continuations yields lower runtimes than the variant using TAMPI, esp. for small values of PPN. Similarly, the data

[4] Detail can be found at https://www.hlrs.de/systems/hpe-apollo-hawk/. Last accessed February 11, 2021.

[5] The PoC implementation is available at https://github.com/devreal/ompi/tree/mpi-continue-master. Last accessed February 11, 2021.

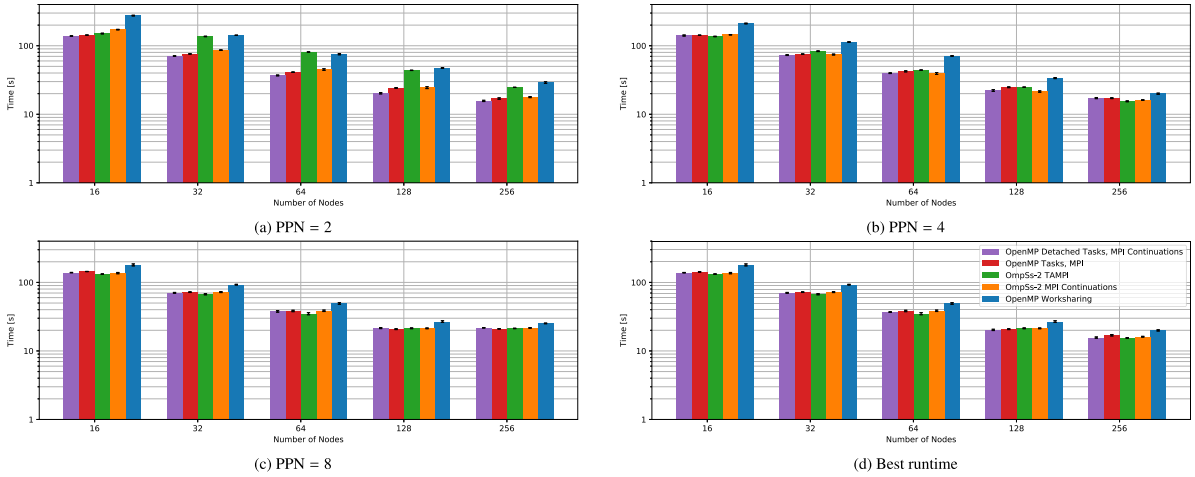
[6] The different variants of NPB BT-MZ discussed here are available at https://github.com/devreal/npb3.3.



(a) PPN = 2 (b) PPN = 4

(c) PPN = 8 (d) Best runtime

Fig. 2. NPB BT-MZ class D benchmark results.



(a) PPN = 2 (b) PPN = 4

(c) PPN = 8 (d) Best runtime

Fig. 3. NPB BT-MZ class E benchmark results.

shows that using continuations with OpenMP tasks yields better results for low PPN, esp. at the upper end of the process range (Fig. 2(a)). For larger PPN the OpenMP variant using a task-based fork-join model achieves the same performance for higher PPN as the continuation-based variant with smaller PPN. It should be noted, however, that at the upper end of the process range with PPN = 8, each process only receives two zones on average, diminishing the benefit of overlapping computations of multiple zones, esp. if the two zones share a boundary.

Class E. Fig. 3 shows the results for runs using class E, i.e., with 4× more and larger zones than class D (Table 2). Similar to class D, the scaling of the OmpSs-2 variant using TAMPI is inhibited for PPN = 2, in contrast to the other task-based variants. This suggests that the handling of requests inside TAMPI and the required inter-thread synchronization harms performance especially for high numbers of threads. However, this time for higher PPN the scaling is significantly better with TAMPI, at times slightly outperforming the OmpSs-2 variant using continuations (e.g., 32 and 64 nodes with PPN = 8). The OpenMP variant using continuations again outperforms the fork-join variant for PPN = 2 and is mostly on par for other configurations.

Two factors may contribute to this observation. First, with higher PPN, even with only 64 nodes there are merely eight zones per rank on average, with the static load balancing scheme potentially assigning only one or two zones to some of the ranks, diminishing the potential for overlapping computation and communication. Second, higher PPN also reduce the ability of MPI to offload communication to the network interface card and requiring more communication to be performed using node-local memory copies, requiring more work by the compute threads.

These two factors may explain the fact that the OpenMP variant using continuations performs best with small PPN values. This in turn suggests, that the interface proposed in this paper can help improve the performance of task-based applications using MPI within tasks as lower PPN values may reduce the surface-to-volume ratio in some application and provide for low-overhead communication management even with higher numbers of threads.

5.3. PaRSEC

The PaRSEC tasking runtime [6] provides an integrated task execution and communication environment, which tracks ownerships and moves data between nodes. Originally proposed as a runtime for DPLASMA (a library of linear algebra algorithms and successor of ScaLAPACK) [28,29], PaRSEC has evolved into a generalized runtime with different frontends for applications that can be expressed as directed acyclic graphs. At its core is a scheduler using POSIX threads to execute tasks and a global dependency tracking and communication system. Communication itself is performed by a dedicated communication thread, allowing MPI to be used with `MPI_THREAD_FUNNELED` to avoid potential thread-synchronization overheads.

The communication thread thus handles both outgoing and incoming communication. In its latest incarnation (on which this work is

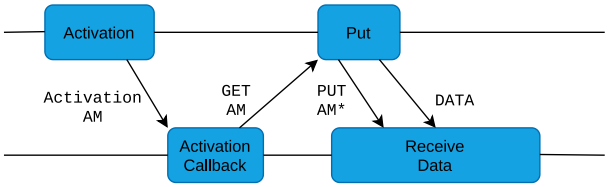


Fig. 4. The PaRSEC communication pattern for activating a dependency: an active message (AM) is sent to trigger the activation at the remote rank, which then sends an AM to request the data that flows along the dependency. With the two-sided MPI communication backend, the data communication itself involves an (internal) AM (marked with *) to trigger the receive matching the send. The data communication happens without blocking the communication thread.

based), the interface has been redesigned to resemble a one-sided communication API providing `put`, `get`, and active-message capabilities, independent of the underlying communication backend. In particular, the MPI backend emulates one-sided communication using two-sided send/recv by exchanging internal active messages (AM) that start a receive (`put`) or send (`get`) at the remote side before issuing the corresponding local operation. On top of that, AMs are used to send activation messages for dependencies that have been resolved by completing a task. The communication pattern involved in activating a dependency and transferring the data using an emulated `put` is depicted in Fig. 4.

In the reference implementation, a small number of persistent receives are posted for each registered AM tag. In a tight loop, the communication thread sends outgoing messages and processes incoming messages.

Incoming activation messages require the communication thread to release dependent tasks and potentially trigger new communication operations to retrieve data required as input to the activated tasks. This may require significant work on the part of the communication thread, in which time no other active message can be processed.

As depicted in Fig. 5, the communication thread manages a fixed size array of requests that are passed to `MPI_Testsome` to test for completion of communication operations, including send and receives posted for AMs and data messages. This set of *active requests* is kept small deliberately to mitigate the overhead of request checking in `MPI_Testsome` and the subsequent reorganizing required once requests have completed. This set is accompanied by a list of *pending requests* that are posted send and receive operations and are eventually moved into the set of active requests once any active data message has completed.

While keeping the set of active requests small ensures efficient testing, it may lead to delays in completion detection: while a recently posted operation may have already completed, it may not be recognized as such until its request is moved into the set of active requests.

The communication thread also limits the number of concurrent outgoing data messages in order to prioritize the limited bandwidth on a first-come–first-serve basis (message prioritization based on urgency is done at higher levels inside PaRSEC). This simple throttling mechanism prevents heavy contention and ensures that messages can be sent with a large fraction of the available bandwidth instead of hundreds of messages being transferred with a small share of the bandwidth, which would delay all messages.

5.3.1. Implementation using continuations

The use of continuations in the context of the PaRSEC communication engine promised to provide faster reaction times to incoming active messages and completed transfers, by executing the action associated with an active message or completed communication. We thus register continuations for all requests, essentially eliminating the set of *pending requests* and the use of `MPI_Testsome`. Any time the MPI implementation finds operations to complete, the attached continuation

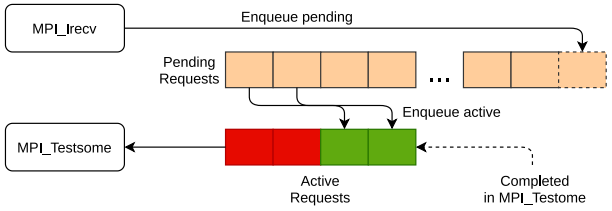


Fig. 5. Request management in the PaRSEC communication thread.

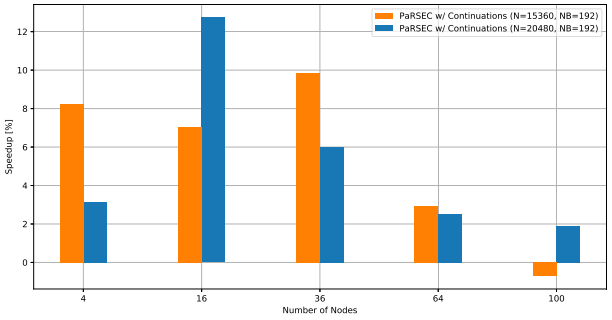


Fig. 6. Speedup of the QR factorization in DPLASMA using continuations over PaRSEC's reference implementation with different per-node matrix and tile sizes.

would be enqueued for execution. By default, continuations for *any* of the operations would be eligible for immediate execution.

However, especially the activation AM callbacks may be expensive, as they may in turn trigger new communication operations. Moreover, the communication thread has to potentially handle a large number of active messages in bursts, effectively prolonging the time until the it is able to handle outgoing messages again. Such bursts lead to poor performance as outgoing activations will be delayed, starving ranks depending on them.

We thus created different continuation requests for different classes of operations (incoming active messages as well as incoming and outgoing data messages) and – on the one used for AMs – set the info key `mpi_continue_poll_only` discussed in Section 3.5 to `true` in order to restrict the execution of incoming active messages to the point when the communication thread calls `MPI_Test` on the respective continuation request. An exception are AMs used to emulate the one-sided communication described above, which are known to be of short duration since they only encompass posting a non-blocking send or receive. We also set `mpi_continue_enqueue_complete` to `true` for all AMs to enqueue continuations even if the receive is complete immediately, preventing the communication thread from having to process large number of incoming AMs in burst situations.

Similarly, we are restricting the execution of continuations of incoming data messages, as they may incur work in the scheduler (releasing tasks) or additional communication (forwarding messages in hierarchical collective communication operations). However, the completion of an outgoing data message is typically a short operation and can thus be executed immediately, freeing up a slot for the next outgoing data message.

5.3.2. QR factorization

The weak-scaling benchmark results for the QR factorization are depicted in Fig. 6, using matrix sizes of 15 k^2 and 20 k^2 elements per node and a tile size of 192. We found that the usage of continuations is most effective on smaller matrix sizes and relatively small tile sizes as that requires a fast exchange of both activations and data as worker thread starvation may occur otherwise. Here, we observe more than 12% speedup on 16 nodes (N = 20 k^2) and 10% speedup on 36 nodes (N = 15 k^2), while for larger numbers of nodes the effect diminishes.

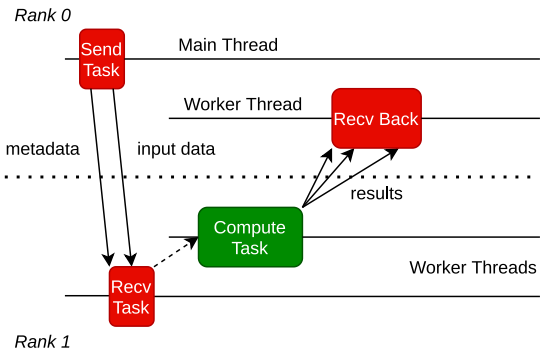


Fig. 7. Offloading a task in ExaHyPE.

For larger tile sizes, we have observed neither a positive nor negative impact on the performance, suggesting that these configurations are less sensitive to latency and thus do not benefit from faster responses.

It is left as future work to further explore the use of continuations in other use-cases of PaRSEC and to analyze PaRSEC's communication requirement at higher node counts.

5.4. ExaHyPE

The ExaHyPE project has recently proposed a novel approach for load-balancing task-based adaptive mesh refinement (AMR) applications by offloading tasks from ranks with high loads to ranks with lower loads [24]. Load balancing decision are made based on instrumenting MPI waiting times in at runtime. They serve as input to a diffusive load balancing scheme in which purely local offloading decisions are made to mitigate the impact of overloaded processes, leading to improved load balance. The project uses Intel Threading Building Blocks (TBB) to manage shared memory parallelism on top of MPI for distributed memory parallelism. Due to its domain decomposition of the underlying space-tree created by tri-partitioning, ExaHyPE exhibits severe load imbalances at configuration other than 28 or 731 ranks.

Tasks are being offloaded by processes identified as critical ranks to other processes based on the amount of time waited for a kick-off message to start the next iteration. The decision on whether and where to offload a task is made at the time the task is discovered by the main thread during the traversal of the grid. If local threads are idle, tasks will be scheduled for local execution, i.e., they are not subject to being sent to another process. If, on the other hand, all local threads are busy and some additional tasks are available locally, the task will be sent to another rank, whose load was determined to be lower than the local load. Fig. 7 depicts the flow of an offloaded task. Offloading a task consists in sending a short message containing the task metadata (e.g., timestep size and a timestamp) followed by a second message containing the task's input data, which are sent in two distinct messages as packing them into one would incur significant overhead.

A small number of pre-posted persistent receives for metadata are used to identify incoming tasks. Once such a receive completes, a receive for the task input data is initiated and upon completion the task is enqueued for execution by any of the worker threads. As soon as a worker thread has completed the execution of an offloaded task, it initiates sending back the results, which consists of three messages. At the offloading source, the non-blocking receives are posted in the callback signaling the completion of the send requests, which is useful for keeping the number of active requests low.

While it would be possible for the main thread to post the receives for the task results immediately after sending the task input data, we found that the extra time spent on posting the three receive operations induces significant overhead for the main thread (which is on the critical path) and thus slows down the overall execution.

Table 3
Lines of code required for submitting and progressing requests in ExaHyPE with and without MPI Continuations.

Operation	Reference	Continuations	Δ
Submitting requests	55	36	−19
Progressing requests	133	17	−116

In case the offloading target cannot return the result for the task in time, a so-called *emergency* is triggered and the number of tasks offloaded to that rank is reduced. If multiple emergencies occur, the rank will not receive any tasks for a number of timesteps and the algorithm tries to shift the load to other ranks.

All communication happens using non-blocking send and receive operations, which are managed by a central *offloading manager* that invokes a callback upon detecting their completion, essentially implementing a form of continuations in the application space. Since offloading a task involves multiple messages in both directions, it is necessary to manage groups of requests whose combined completion triggers the invocation of a single callback. The offloading manager thus uses multiple parallel data structures to manage the mapping between MPI requests and callbacks.

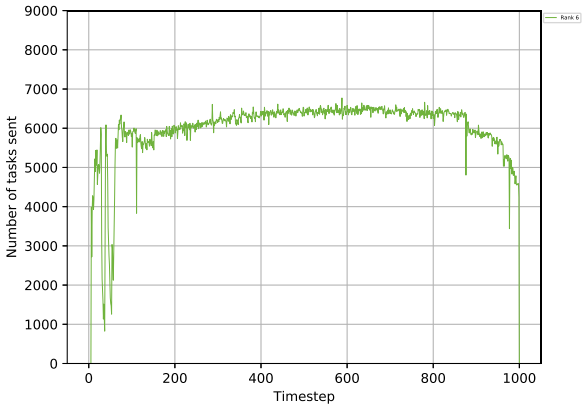
Worker threads will poll the offloading manager for completed communication operations, passing a subset of the active requests to `MPI_Testsome` (similar to PaRSEC). However, it is not clear what constitutes an optimal number of requests to pass to the test function. On the one hand, too little requests might cause some tasks that were offloaded to slow ranks and complete too late to block the detection of tasks returned from fast ranks, thus leading to unnecessarily large numbers of emergencies. On the other hand, a large number of requests might incur significant overhead inside `MPI_Testsome` during the linear walk through the request array to check all request for completion. In order to ensure fast detection of completed tasks, the reference implementation opted for eagerly testing as many requests as possible while trying to avoid repeated reallocation of the request array.

Overall, this scheme is a natural fit for MPI Continuations.

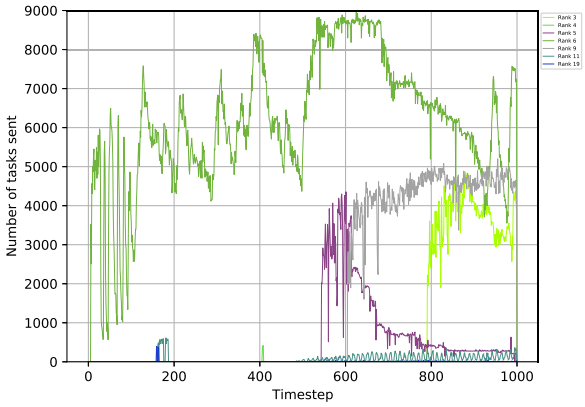
5.4.1. Implementation using continuations

In our implementation using MPI Continuations, we were able to replace much of the complexity of managing groups of requests with a single call to `MPIX_Continueall`, reducing the amount of code necessary to implement the registration of callbacks with groups of requests, as shown in Table 3. An even more significant reduction in code can be found in the progression of registered requests. Where the reference implementation has to assemble a continuous array of requests to pass to `MPI_Testsome` and handle the returned array of indices, the use of continuations only requires invoking `MPI_Test` on the continuation request. In addition to the reduction in lines of code, we also removed the data structures that were dedicated to managing the required mapping between requests, groups, callbacks, and callback arguments in the application space (not shown in Table 3). Overall, the use of the continuations interface has reduced the overall complexity of a part of code that is central to the load balancing scheme described above.

Since requests do not have to managed in the application space, we have implemented a scheme that shifts the callback registration for task send requests to worker threads, allowing the use of a single continuation for send and receive requests. This scheme has not yielded any improvements in the reference implementation, presumably due to the larger number of simultaneously active requests to be handled by the offloading manager.

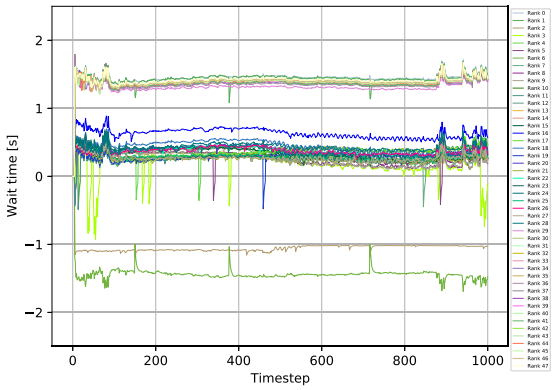


(a) Reference implementation.

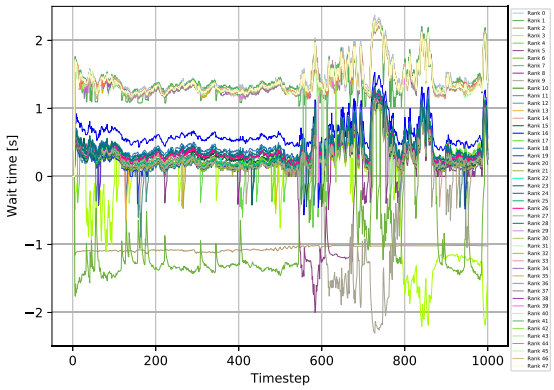


(b) Using continuations.

Fig. 8. Number of offloaded tasks over 1000 iterations. Each line represents a single offloading source. Using continuations results in higher numbers of tasks being offloaded and a shift in the critical rank.



(a) Reference implementation.



(b) Using continuations.

Fig. 9. Wait times per timestep over 1000 iterations on 48 ranks. Ranks with positive times wait for other ranks to complete the timestep. Ranks with negative wait times are being waited on. Using continuations yields lower runtimes on the critical path (lower half) until other ranks start offloading tasks, leading to disturbances.

5.4.2. Results

In our benchmark, we use ExaHyPE's ADER-DG implementation to solve the compressible Navier Stokes equations for the simulation of clouds [30]. For small problem sizes, we have not observed significant differences between the reference implementation and the implementation using continuations. However, Fig. 8 shows the number of tasks offloaded over a run of 1000 timesteps on a grid of 81^3 points on 24 ranks with 32 threads each. Fig. 8(a) shows that in the reference implementation, only one rank is offloading tasks as that rank remains on the critical path, reaching a peak at approximately 6500 tasks. In contrast, Fig. 8(b) shows that when using continuations, the same rank is able to offload just short of 9000 tasks (+35%), at which point other ranks are detected as being critical by the algorithm, causing them to offload tasks as well.

Fig. 9 shows the corresponding wait times for each iteration. A positive wait time indicates that the rank is waiting for one or more other ranks (and thus could become an offloading target) while a negative wait time is used to indicate that the corresponding rank is being waited on (and thus should offload tasks). In the reference implementation (Fig. 9(a)), the rank on the critical path is being waited on for approximately 1.45 s for the majority of timesteps. With continuations, on the other hand, the wait time for the critical rank in the first 550 iterations is at 1.3 s, which constitutes a 10% speedup (Fig. 9(b)). However, as other ranks are detected as critical, a disturbance occurs because the offloading targets that have so far only processed tasks of the initial critical rank are now served tasks from other ranks as well, rendering them unable to process tasks in a timely manner. The algorithm is able to identify this condition and adjust the number of tasks offloaded to recover from the overload. Due to the diffusive nature of the algorithm, many more iterations will be required to achieve a stable state. However, it is only through the use of continuations that enough tasks can be offloaded in this scenario to trigger the change in the critical rank.

6. Related work

Multiple efforts have been made in the past to improve specific aspects of the interaction between multi-threaded programming models and MPI [31–33].

Extended generalized requests have been proposed to encapsulate operations within requests and allow the MPI implementation to progress them through a callback [34]. Similar to the interface proposed here, extended generalized requests employs *inversion of control* (IoC) [35]: application-level code is called by the MPI library in inversion of the usual call relation. However, these requests are still polling-based and cannot be used for completion notification.

IoC is also used in the upcoming MPI_T events interface, allowing applications and tools to be notified through callbacks about events from within the MPI library [23]. However, the availability of events is implementation-specific and callbacks can only be registered for *event types*, as opposed to specific operations. The MPI_T interface has been used to structure the interaction between MPI and task-based programming models, still requiring the mapping between MPI requests and task-information in the application layer [36].

Continuations are not a new concept in the context of low-level communication libraries: UCX uses callback notification functions when initiating non-blocking operations [37]. In contrast, other interfaces

such as libfabric [38], Portals [39], and uGNI [40] rely on completion queues from which notifications are popped. While this would be a viable concept in MPI as well, we believe that continuations provide greater flexibility at the application level. Other low-level communication abstractions such as LCI [41] provide lightweight completion notification by setting variables directly accessible by the application.

Another alternative would be a system similar to *scheduler activations* in OS kernels [42], where MPI would signal to some application-level entity that the current thread or fiber would block inside a call. This has been proposed for the OpenShmem standard [43]. However, its use is limited to tasking models that have strong rescheduling guarantees and would not work with OpenMP detached tasks.

Hardware-triggered operations are being used to react to events in the network with low latency and to offload computation to the network interface card [44–46]. However, in-network computing is focused on operations on the data stream and likely cannot be utilized to execute arbitrary code in user-space, e.g., call into tasking libraries. Exploring the boundaries of integrating these systems with the interface proposed here remains as future work.

The integration of tasking and fiber libraries into MPI has been proposed as a replacement for POSIX thread support to enable the use of blocking MPI calls inside tasks and fibers [15]. We have discussed this in detail in [17]. Other proposals provide runtime-specific wrapper libraries around MPI, transforming blocking MPI calls to non-blocking calls and testing for completion before releasing a fiber or task [13,16, 47]. These proposals fall short of providing a portable interface that can be used with arbitrary asynchronous programming models.

A proposal for completion callbacks for MPI request objects has been discussed in the context of the MPI Forum over a decade ago and rejected due to its broad focus [48]. In contrast, the proposal outlined here avoids some of the pitfalls such as callback contention, ownership of requests, and unclear progress semantics.

A proposal made concurrently to ours has been made in the form of `MPI_Detach` [49]. While similar in its goals and its general interface (registering completion callbacks for one or more requests), it differs in subtle but important ways. Most importantly, the API does not provide immediate completion and uses a general progress procedure to process outstanding completion callbacks without the testing and waiting capability provided by continuation requests. However, the existence of a concurrent proposal underscores the need for MPI to provide an interface for its integration with task-based programming models.

The interface proposed here is especially well-suited for coupling MPI with node-local asynchronous programming models such as OpenMP [1], OmpSs-2 [21], and Intel TBB [50] or in combination with cooperatively scheduled fiber libraries such as Argobots [51] or Qthreads [52] to ease the MPI communication management burden users are facing.

7. Conclusions

In order to tackle the challenges of asynchronous activities communicating through MPI, we have proposed an extension to the MPI standard that allows users to register continuations with active operations that will be invoked once the MPI library determines the operations to be complete. In this paper, we have extended this interface with fine-grained controls meant to steer the behavior of the MPI implementation to accommodate the needs of applications by using the info key mechanism of MPI. Moreover, we have discussed early experiences in using this interface in combination with four different task-based runtime systems, namely OpenMP detached tasks, OmpSs-2, Intel TBB, and the PaRSEC custom scheduler.

The overhead of our proof-of-concept implementation is sufficiently low to not impact existing applications and its use shows similar or improved performance over existing approaches. Overall, MPI continuations may provide a powerful tool for reducing the latency in thread-parallel message passing applications while reducing the programmer's burden on writing and maintaining complex software for managing MPI requests in the application space.

The interface has been designed with higher-level abstractions such as C++ futures for MPI in mind but it remains as future work to provide an actual implementation on top of MPI Continuations. However, we are confident that our design will be useful in the design of an asynchronous C++ MPI interface.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.parco.2021.102793.

Acknowledgments

This research was in parts funded by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) in the SmartDASH project. The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under the ChEESE project, grant agreement No. 823844. This material is based upon work supported by the National Science Foundation under Grant No. #1664142 and the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the US Department of Energy Office of Science and the National Nuclear Security Administration.

References

[1] OpenMP architecture review board, OpenMP application programming interface, version 5.0, 2018, URL https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf. (Accessed 24 April 2020).

[2] A. Duran, A. Eduard, R.M. Badia, J. Larbarta, L. Martinell, X. Martorell, J. Plana, OmpSs: A proposal for programming heterogeneous multi-core architectures, Parallel Process. Lett. 21 (02) (2011) 173–193, http://dx.doi.org/10.1142/S0129626411000151.

[3] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, 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, ACM, 2014, pp. 6:1–6:11, http://dx.doi.org/10.1145/2676870.2676883.

[4] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, StarPU: A unified platform for task scheduling on heterogeneous multicore architectures, Concurr. Comput.: Pract. Exp. 23 (2) (2011) 187–198, http://dx.doi.org/10.1002/cpe.1631.

[5] J. Schuchart, J. Gracia, Global task data-dependencies in PGAS applications, in: High Performance Computing, Springer International Publishing, 2019, http://dx.doi.org/10.1007/978-3-030-20656-7_16.

[6] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, J. Dongarra, DAGuE: A generic distributed DAG engine for high performance computing, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, pp. 1151–1158, http://dx.doi.org/10.1109/IPDPS.2011.281.

[7] D.E. Bernholdt, S. Boehm, G. Bosilca, M.G. Venkata, R.E. Grant, T. Naughton, H.P. Pritchard, M. Schulz, G.R. Vallee, A survey of MPI usage in the US exascale computing project, Concurr. Comput.: Pract. Exp. (2018) http://dx.doi.org/10.1002/cpe.4851.

[8] MPI: A Message-Passing Interface Standard. Version 3.1, Tech. Rep., 2015, URL http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf. (Accessed 24 April 2020).

[9] T. Patinyasakdikul, D. Eberius, G. Bosilca, N. Hjelm, Give MPI threading a fair chance: A study of multithreaded MPI designs, in: IEEE International Conference on Cluster Computing, CLUSTER, 2019, pp. 1–11, http://dx.doi.org/10.1109/CLUSTER.2019.8891015.

[10] N. Hjelm, M.G.F. Dosanjh, R.E. Grant, T. Groves, P. Bridges, D. Arnold, Improving MPI multi-threaded RMA communication performance, in: Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018, ACM, 2018, pp. 58:1–58:11, http://dx.doi.org/10.1145/3225058.3225114.

[11] J. Schuchart, K. Tsugane, J. Gracia, 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, http://dx.doi.org/10.1007/978-3-319-98521-3_1, awarded Best Paper.

[12] G. Daiß, P. Amini, J. Biddiscombe, P. Diehl, J. Frank, K. Huck, H. Kaiser, D. Marcello, D. Pfander, D. Pfüger, From piz daint to the stars: Simulation of stellar mergers using high-level abstractions, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, 2019, http://dx.doi.org/10.1145/3295500.3356221.

[13] K. Sala, X. Teruel, J.M. Perez, A.J. Peña, V. Beltran, J. Labarta, Integrating blocking and non-blocking MPI primitives with task-based programming models, Parallel Comput. 85 (2019) 153–166, http://dx.doi.org/10.1016/j.parco.2018.12.008.

[14] G. Mercier, F. Trahay, D. Buntinas, E. Brunet, NewMadeleine: An efficient support for high-performance networks in MPICH2, in: Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS, 2009, http://dx.doi.org/10.1109/IPDPS.2009.5161003.

[15] H. Lu, S. Seo, 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, http://dx.doi.org/10.1109/HPCC-CSS-ICESS.2015.82.

[16] K. Sala, J. Bellón, P. Farré, X. Teruel, J.M. Perez, A.J. Peña, D. Holmes, V. Beltran, J. Labarta, Improving the interoperability between MPI and task-based programming models, in: Proceedings of the 25th European MPI Users' Group Meeting, EuroMPI'18, Association for Computing Machinery, New York, NY, USA, 2018, http://dx.doi.org/10.1145/3236367.3236382.

[17] J. Schuchart, C. Niethammer, 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, Association for Computing Machinery, 2020, pp. 39–50, http://dx.doi.org/10.1145/3416315.3416320.

[18] D.P. Friedman, C.T. Haynes, E. Kohlbecker, Programming with continuations, in: Program Transformation and Programming Environments, Springer Berlin Heidelberg, 1984, http://dx.doi.org/10.1007/978-3-642-46490-4_23.

[19] J.C. Reynolds, The discoveries of continuations, LISP Symb. Comput. (1993) http://dx.doi.org/10.1007/BF01019459.

[20] N. Gustafsson, A. Laksberg, H. Sutter, S. Mithani, N3857: Improvements to `Std::Future<T>` and Related APIs, Tech. Rep. N3857, 2014.

[21] Barcelona Supercomputing Center, OmpSs-2 Specification, Tech. Rep., 2020, URL https://pm.bsc.es/ftp/ompss-2/doc/spec/OmpSs-2-Specification.pdf. (Accessed 12 May 2020).

[22] P.V. Bangalore, R. Rabenseifner, D.J. Holmes, J. Jaeger, G. Mercier, C. Blaas-Schenner, A. Skjellum, Exposition, clarification, and expansion of MPI semantic terms and conventions: Is a nonblocking MPI function permitted to block? in: Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19, Association for Computing Machinery, New York, NY, USA, 2019, http://dx.doi.org/10.1145/3343211.3343213.

[23] M.-A. Hermanns, N.T. Hjelm, M. Knobloch, K. Mohror, M. Schulz, The MPI_T events interface: An early evaluation and overview of the interface, Parallel Comput. (2019) http://dx.doi.org/10.1016/j.parco.2018.12.006.

[24] P. Samfass, T. Weinzierl, D.E. Charrier, M. Bader, Lightweight task offloading exploiting MPI wait times for parallel adaptive mesh refinement, Concurr. Comput.: Pract. Exper. (2020) http://dx.doi.org/10.1002/cpe.5916.

[25] NASA advanced supercomputing division, problem sizes and parameters in NAS parallel benchmarks, URL https://www.nas.nasa.gov/publications/npb_problem_sizes.html.

[26] R.F.V. der Wijngaart, H. Jin, NAS Parallel Benchmarks, Multi-Zone Versions, Tech. Rep. NAS-03-010, NASA Advanced Supercomputing (NAS) Division, 2003.

[27] H. Jin, R.F. Van der Wijngaart, Performance characteristics of the multi-zone NAS parallel benchmarks, J. Parallel Distrib. Comput. 66 (5) (2006) 674–685, http://dx.doi.org/10.1016/j.jpdc.2005.06.016.

[28] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, J. Dongarra, Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA, in: 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum, 2011, http://dx.doi.org/10.1109/IPDPS.2011.299.

[29] J. Choi, J.J. Dongarra, R. Pozo, D.W. Walker, ScaLAPACK: a scalable linear algebra library for distributed memory concurrent computers, in: [Proceedings 1992] the Fourth Symposium on the Frontiers of Massively Parallel Computation, 1992, pp. 120–127, http://dx.doi.org/10.1109/FMPC.1992.234898.

[30] L. Krenz, L. Rannabauer, M. Bader, A high-order discontinuous Galerkin solver with dynamic adaptive mesh refinement to simulate cloud formation processes, in: Parallel Processing and Applied Mathematics, Springer International Publishing, 2020, http://dx.doi.org/10.1007/978-3-030-43229-4_27.

[31] T. Hoefler, G. Bronevetsky, B. Barrett, B.R. de Supinski, A. Lumsdaine, Efficient MPI support for advanced hybrid programming models, in: Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, 2010, http://dx.doi.org/10.1007/978-3-642-15646-5_6.

[32] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling MPI interoperability through flexible communication endpoints, in: Proceedings of the 20th European MPI Users' Group Meeting, EuroMPI '13, Association for Computing Machinery, 2013, http://dx.doi.org/10.1145/2488551.2488553.

[33] R.E. Grant, M.G.F. Dosanjh, M.J. Levenhagen, R. Brightwell, A. Skjellum, Finepoints: Partitioned multithreaded MPI communication, in: High Performance Computing, Springer International Publishing, 2019, http://dx.doi.org/10.1007/978-3-030-20656-7_17.

[34] R. Latham, W. Gropp, R. Ross, R. Thakur, Extending the MPI-2 generalized request interface, in: Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface, PVM/MPI'07, Springer-Verlag, 2007.

[35] M. Mattsson, Object-Oriented Frameworks - a Survey of Methodological Issues (Licentiate thesis), 1996, URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.1424&rep=rep1&type=pdf.

[36] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, A. Bhatele, Optimizing computation-communication overlap in asynchronous task-based programs, in: Proceedings of the ACM International Conference on Supercomputing, ICS '19, Association for Computing Machinery, 2019, http://dx.doi.org/10.1145/3330345.3330379.

[37] Unified communication framework consortium, UCX: Unified communication X API standard v1.6, unified communication framework consortium, 2019, URL https://github.com/openucx/ucx/wiki/api-doc/v1.6/ucx-v1.6.pdf.

[38] OpenFabrics interfaces working group, high performance network programming with OFI, 2017, URL https://github.com/ofiwg/ofi-guide/blob/master/OFIGuide.md. (Accessed 14 May 2020).

[39] B.W. Barrett, R. Brightwell, R.E. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, T. Hoefler, A.B. Maccabe, T. Hudson, The portals 4.2 network programming interface, Tech. Rep. SAND2018-12790, Sandia National Laboratories, 2018.

[40] H. Pritchard, I. Gorodetsky, D. Buntinas, A uGNI-based MPICH2 nemesis network module for the cray XE, in: Recent Advances in the Message Passing Interface, Springer Berlin Heidelberg, 2011, http://dx.doi.org/10.1007/978-3-642-24449-0_14.

[41] H. Dang, R. Dathathri, G. Gill, A. Brooks, N. Dryden, A. Lenharth, L. Hoang, K. Pingali, M. Snir, A lightweight communication runtime for distributed graph analytics, in: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS, 2018, http://dx.doi.org/10.1109/IPDPS.2018.00107.

[42] T.E. Anderson, B.N. Bershad, E.D. Lazowska, H.M. Levy, Scheduler activations: Effective kernel support for the user-level management of parallelism, in: Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles, SOSP '91, Association for Computing Machinery, 1991, http://dx.doi.org/10.1145/121132.121151.

[43] M. Wasi-ur Rahman, D. Ozog, J. Dinan, Simplifying communication overlap in OpenSHMEM through integrated user-level thread scheduling, in: High Performance Computing, Springer International Publishing, 2020, http://dx.doi.org/10.1007/978-3-030-50743-5_25.

[44] T. Hoefler, S. Di Girolamo, K. Taranov, R.E. Grant, R. Brightwell, sPIN: High-performance streaming processing in the network, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '17, ACM, 2017, pp. 59:1–59:16, http://dx.doi.org/10.1145/3126908.3126970.

[45] W. Schonbein, R.E. Grant, M.G.F. Dosanjh, D. Arnold, Inca: In-network compute assistance, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, Association for Computing Machinery, 2019, http://dx.doi.org/10.1145/3295500.3356153.

[46] N.S. Islam, G. Zheng, S. Sur, A. Langer, M. Garzaran, Minimizing the usage of hardware counters for collective communication using triggered operations, in: Proceedings of the 26th European MPI Users' Group Meeting, EuroMPI '19, Association for Computing Machinery, 2019, http://dx.doi.org/10.1145/3343211.3343222.

[47] D.T. Stark, R.F. Barrett, R.E. Grant, S.L. Olivier, K.T. Pedretti, C.T. Vaughan, Early experiences co-scheduling work and communication tasks for hybrid mpi+x applications, in: Proceedings of the 2014 Workshop on Exascale MPI, ExaMPI '14, IEEE Press, 2014, http://dx.doi.org/10.1109/ExaMPI.2014.6.

[48] MPI Collectives Working Group, Request completion callback function, 2008, MPI Forum Discussion, archived at https://github.com/mpi-forum/mpi-forum-historic/issues/26. (Accessed 6 July 2020).

[49] J. Protze, M.-A. Hermanns, A. Demiralp, M.S. Müller, T. Kuhlen, MPI detach – asynchronous local completion, in: 27th European MPI Users' Group Meeting, EuroMPI/USA '20, Association for Computing Machinery, 2020, http://dx.doi.org/10.1145/3416315.3416323.

[50] J. Reinders, Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism, O'Reilly & Associates, 2007.

[51] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castelló, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L.V. Kalé, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, P. Beckman, Argobots: A lightweight low-level threading and tasking framework, IEEE Trans. Parallel Distrib. Syst. (2018) http://dx.doi.org/10.1109/TPDS.2017.2766062.

[52] K.B. Wheeler, R.C. Murphy, D. Thain, Qthreads: An API for programming with millions of lightweight threads, in: 2008 IEEE International Symposium on Parallel and Distributed Processing, 2008, http://dx.doi.org/10.1109/IPDPS.2008.4536359.