MPI Detach - Asynchronous Local Completion

Joachim Protze
RWTH Aachen University, ITC
Aachen, Germany
protze@itc.rwth-aachen.de

Marc-André Hermanns
RWTH Aachen University, ITC
Aachen, Germany
hermanns@itc.rwth-aachen.de

Ali Demiralp
RWTH Aachen University, VCI
Aachen, Germany
demiralp@vr.rwth-aachen.de

Matthias S. Müller
RWTH Aachen University, ITC
Aachen, Germany
mueller@itc.rwth-aachen.de

Torsten Kuhlen
RWTH Aachen University, VCI
Aachen, Germany
kuhlen@vr.rwth-aachen.de

ABSTRACT

When aiming for large scale parallel computing, waiting time due to network latency, synchronization, and load imbalance are the primary opponents of high parallel efficiency. A common approach to hide latency with computation is the use of non-blocking communication. In the presence of a consistent load imbalance, synchronization cost is just the visible symptom of the load imbalance. Tasking approaches as in OpenMP, TBB, OmpSs, or C++²⁰ coroutines promise to expose a higher degree of concurrency, which can be distributed on available execution units and significantly increase load balance. Available MPI non-blocking functionality does not integrate seamlessly into such tasking parallelization. In this work, we present a slim extension of the MPI interface to allow seamless integration of non-blocking communication with available concepts of asynchronous execution in OpenMP and C++.

ACM Reference Format:

Joachim Protze, Marc-André Hermanns, Ali Demiralp, Matthias S. Müller, and Torsten Kuhlen. 2020. MPI Detach - Asynchronous Local Completion. In *27th European MPI Users' Group Meeting (EuroMPI/USA '20)*, September 21–24, 2020, Austin, TX, USA. ACM, New York, NY, USA, 10 pages. <https://doi.org/10.1145/3416315.3416323>

1 INTRODUCTION

Next to theory and experimentation, simulation has long established its place as the third pillar of science. This has lead to an ever-growing increase in demand for computational power in high-performance computing (HPC). To provide this computational power, HPC systems comprise up to thousands of separate compute nodes in a single distributed-memory system. The de-facto standard programming model for distributed-memory systems in a scientific and HPC context is the Message Passing Interface (MPI) [5]. Here, processes exchange explicit messages to coordinate work and distribute data.

While conceptually targeting distributed-memory platforms with its explicit exchange of messages, MPI can be employed on

distributed-memory systems, shared-memory systems, and hybrids of the two. This means multiple processes can exchange messages transparently, regardless of whether they execute on the same compute node or not. While the available execution units on a shared-memory domain were limited, applications could use shared-memory as a “fast interconnect” between MPI processes without the need to employ separate (orthogonal) programming models for each type of memory domain. With the rise of large multicore processors that today comprise tens to hundreds of cores in a single shared-memory domain with a decreasing memory-per-core ratio, application developers more and more adopt shared-memory programming into their MPI applications to overcome some of the limitations of classic multi-processing, such as data replication.

While MPI provides an interface to shared-memory programming using remote-memory access, the prevalent programming model for shared-memory programming remains multi-threading using threading interfaces, such as OpenMP, POSIX threads, and C++ threads. MPI is largely thread-agnostic in that it acknowledges the existence of threads and how to ensure a thread-safe use, but not much beyond that. Proposals to introduce Endpoints [3] or Finepoints [6] highlight the increasing interest in providing better support and performance for multi-threaded MPI applications.

OpenMP [13] has a long tradition in HPC. OpenMP worksharing has long been the preferred easy-to-use parallelization construct in bulk-synchronous applications that iteratively alternate communication phases (using MPI) and computation phases (using OpenMP). This mode of operation fits very well in the past, enabling the MPI communication to be mostly separate from the thread parallelization. With the introduction of tasking in OpenMP 3.0 in 2008, task-level programming became more and more important for modern application codes, with its reliance on tracking data dependencies and efficient scheduling through a runtime rather than the programmer, trying to minimize waiting time in irregular applications. This can dissolve the strict separation of communication and computation phases, creating a need for tasks performing MPI communication. The use of OpenMP `taskyield` [18] has proven as not reliable for MPI communication in OpenMP tasks because both OpenMP and MPI provide too weak guarantees. The framework Chameleon [10] already takes the next step and supports the migration of tasks to other MPI ranks in order to improve the load balance further.

MPI provides different threading levels (see Section 2.1), but for irregular tasking applications, this may lead to a requirement of the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroMPI/USA '20, September 21–24, 2020, Austin, TX, USA

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-8880-1/20/09...\$15.00

<https://doi.org/10.1145/3416315.3416323>

highest level (**MPI_THREAD_MULTIPLE**) negatively impacting performance. TAMPI, a task-aware MPI implementation [16, 17], marks a first step towards integration of MPI and tasking parallelism. While this work shows the potential of this kind of programming, it assumes the MPI implementation to be explicitly aware of a tasking paradigm.

In our work, we propose a slim extension of the MPI interface, which integrates into existing MPI interfaces to provide functionality very similar to TAMPI. However, our approach keeps MPI and tasking separate while encouraging the interoperability of the two paradigms. We will show that even with the task-agnostic interface, we can write clear code and improve performance. Furthermore, this interface is also applicable to use cases beyond OpenMP/OmpSs tasks, such as C++ futures.

This paper presents the following contributions: (1) interface extensions to MPI that enable users to transfer control over request completion to the MPI library, while providing a callback-driven notification of completion back to the user; (2) a prototype implementation of the proposed interface that is independent of the MPI implementation used; and (3) a modern C++ interface for the proposed detach interface using Boost.MPI and Boost.ASIO.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction into MPI non-blocking communication requests. Section 3 provides a motivational overview of desired interpretations of MPI non-blocking communication with other interfaces, such as OpenMP and modern C++. Section 4 proposes interface extensions to MPI to hand over requests to the MPI library for asynchronous local completion, before describing our prototype implementation in Section 5. We present measurements on modified codes that compare our implementation to other approaches in benchmarks and a ray-tracing application in Section 6. Finally, Section 7 concludes this paper and provides an outlook on future work.

2 THE MESSAGE PASSING INTERFACE

This section recaps the relevant features of MPI regarding thread support, non-blocking procedures, and progress to motivate the proposed interface extensions.

2.1 Thread support

MPI provides several levels of thread support, where requirements by the user and guarantees by the MPI runtime system are communicated during initialization: (1) **MPI_THREAD_SINGLE**, (2) **MPI_THREAD_FUNNELED**, (3) **MPI_THREAD_SERIALIZED**, and (4) **MPI_THREAD_MULTIPLE**.

In **MPI_THREAD_SINGLE** only a single user thread will execute. In **MPI_THREAD_FUNNELED** multiple user threads may execute, but only the thread that called **MPI_Init_thread** will call further MPI functions. In **MPI_THREAD_SERIALIZED** multiple user threads may issue calls to the MPI library, but never concurrently. In **MPI_THREAD_MULTIPLE** multiple user threads may issue calls to the MPI library, with no restrictions on concurrency of those calls.

The user negotiates the actual thread support level by providing the required thread support level to **MPI_Init_thread**. The MPI library returns the actual supported thread support level, which may differ from the required value provided by the user. The user is

responsible for ensuring that all interactions with the MPI library are supported in the actual thread support level after initialization.

Threads are not separately addressable in MPI communication. Assuming the highest thread support level, this means a message addressed to an MPI process can be received by any thread in this process.

2.2 Requests

MPI provides blocking and non-blocking communication schemes. Blocking communication functions will return when the respective part of a message exchange is completed locally. Non-blocking communication functions may return before the respective part of a message exchange is completed locally, and one or more further calls to the MPI library are needed to complete the communication operation locally.

To allow for users to query and ensure the completion of specific communication operations initiated in non-blocking communication, MPI provides opaque handles to so-called *requests*.

MPI provides two flavors of such requests: (1) *standard requests*¹ that are created ad-hoc during message initiation and discarded during completion; and (2) *persistent requests*, where message parameters are defined separately before message initiation and which can be reused. Persistent requests may become active when the communication is initiated and inactive when the associated communication operation is complete locally.

Request handles are the interface to ongoing operations that have to be completed using extra completion functions. Both request types—standard and persistent—can be completed using the same set of completion functions. These are either wait or test functions: **MPI_{Wait|Test}{any|some|all}**. Wait functions will block until the corresponding operation completed locally. Test functions will return a flag indicating the completion status of the corresponding operation. If the flag indicates completion of the corresponding operation, the request is considered in the same state as if an equivalent wait function would have been called instead. It is erroneous for a request to appear in two or more concurrent calls to wait or test functions other than **MPI_Test** [5, p. 485].

2.3 Progress and fairness

When we talk about progress in communication, we mean the process of bringing a message from application memory onto the network or at the receiver side from the network to the application buffer. Depending on available hardware capabilities and utilization, the communication library might actively need to control the transmission of data.

Compared to other HPC communication interfaces, such as OpenSHMEM [1], MPI provides a weak progress guarantee. For point-to-point communication, MPI guarantees that once a matching pair of send and receive is initiated, one of them will (eventually) complete. For collective communication, MPI states that once initiated, a non-blocking collective operation *may* progress independently of any computation or other communication at participating processes, but it is not required to progress. This provides much freedom of

¹We name these requests *standard* to easily differentiate them from *persistent* requests where needed. The MPI standard itself does not specifically name them other than *requests*.

implementation for a progress engine, for example, through RDMA hardware support, progress threads, or even progressing pending communication whenever the user calls into the MPI library.

While enabling the implementation on a broad range of platforms, the latter approach may not provide communication-computation overlap possibly expected by the user. When a pending communication cannot progress, although matching communication calls are posted, the application may suffer from waiting time [8] when the communication partners do not call into the MPI library for long periods. Hence, some application codes, in the hope of driving pending communications in the absence of a strong progress engine, repeatedly call into the MPI library, to ensure regular progress for their application.

Furthermore, while MPI guarantees message order for consecutive messages, it does not provide any fairness guarantee in the handling of communication from different sources. This means a pending send operation may stay open even though the receiving process posts matching receives that would match this open send if other send operations from different sources match these receives.

This extends to logically concurrent messages of the same process with the same message envelope when multiple threads in an MPI process initiate non-blocking communication concurrently.

3 INTEROPERABILITY

As a motivation for the proposed interface, we sketch two interoperability use cases. The applicability of the interface is not limited to these use cases.

3.1 Interoperability with OpenMP tasks

As a first interoperability use case, we sketch the interoperability of MPI with OpenMP tasks. We start with a simple tasking code, where we have work depending on incoming data and other work that can be processed independently, e.g., on local data.

Source Code 1: Receive data to be processed in function **work_with**. **other_work** could execute while waiting.

```
#pragma omp task depend(out: data)
MPI_Recv(data, ...);
#pragma omp task depend(in: data)
work_with(data);
#pragma omp task
other_work(other_data);
```

The execution of Source Code 1 will block in **MPI_Recv**, so that no other task can execute while waiting on the data.

Based on OpenMP 4.5, Schuchart et al. [18] suggested to use an OpenMP **taskyield** based approach as sketched in Source Code 2. In the worst case, this approach can result in deadlock due to the limited guarantees OpenMP provides for **taskyield**.

Source Code 2: Introducing non-blocking communication and **taskyield** can help to overlap communication with independent computation.

```
#pragma omp task depend(out: data)
{
    MPI_Request req;
    int received = 0;
```

```
    MPI_Irecv(data, ..., &req);
    MPI_Test(&req, received, ...);
    while(!received){
        #pragma omp taskyield
        MPI_Test(&req, received, ...);
    }
}
#pragma omp task depend(in: data)
work_with(data);
#pragma omp task
other_work(other_data);
```

With OpenMP 5.0 [13], the new feature of detached tasks was introduced. Instead of manually delaying the completion of a task – and therefore leaving the dependency open – the task can finish execution, while task completion depends on a call to **omp_fulfill_event**. Only if the task completes, the dependencies are released and depending tasks can start execution.

Source Code 3: Allow the communication task to finish, but only complete when the MPI communication is completed

```
omp_event_handle_t ev_handle;
#pragma omp task detach(ev_handle) depend(out: data)
{
    MPI_Request req;
    MPI_Irecv(data, ..., &req);
    MPIX_Foo(&req, omp_fulfill_event, ev_handle);
}
#pragma omp task depend(in: data)
work_with(data);
#pragma omp task
other_work(other_data);
```

Source Code 3 shows a possible integration of the new OpenMP feature with MPI non-blocking communication. The new MPI function **MPIX_Foo** takes the request handle and starts monitoring the request for completion of the communication in the background. Once the request is completed, the implementation will call **omp_fulfill_event(ev_handle)**. This will resolve the out dependency to **data**, so that the depending task can execute **work_with(data)**. After the call to **MPIX_Foo**, the task has no use for the request handle, so the function should consume the handle and set it to **MPI_REQUEST_NULL**. Section 4.1 will make a concrete proposal for this function.

Other than Source Code 2, this code does not regularly call into the MPI runtime library. As a result, we can still run into a deadlock, because MPI might not make local progress and therefore **omp_fulfill_event** is never called. We will address this issue by an additionally proposed interface in Section 4.2.

3.2 Interoperability with C++ futures

Despite the official C++ bindings being removed in the MPI 3.0 specification [5], C++ continues to be a common host language for distributed computing. Applications incorporate MPI either by directly using the C interface or through wrappers such as Boost.MPI² and MPP [14]. The latter improves ease-of-use by

²Boost Library Documentation: <https://www.boost.org/doc/libs/>

providing an object-oriented layer on top of the MPI C interface, yet their support for asynchronous operations (i.e., **MPI_I...**) are limited.

Considering the recent extensions to the C++ standard template library (STL) [19], specifically the **std::future<T>** and **std::promise<T>** [9] interfaces which enable transmission of values (and exceptions) across asynchronous operations, it becomes possible to expose an MPI request as a future whose value is updated opaquely from the user as shown in Source Code 4:

Source Code 4: Wrapping the completion of an MPI request into a C++ future

```
std::future<mpi::status> as_future(mpi::request& r)
{
    auto* p = new std::promise<mpi::status>();
    auto f = promise->get_future();
    mpix::foo(&r, [p] (const mpi::status& s)
    {
        p->set_value(s);
        delete p;
    });
    return std::move(f);
}
```

Note how the lifetime of the promise is bound to the completion of the request. The **mpix::foo** is an analogue of **MPIX_Foo** from Section 3.1; it is responsible for querying the request in a non-blocking manner and calling the given callback on completion. The users may then **.wait()** the returned future which is equivalent to synchronous **MPI_Wait** or more interestingly chain them using continuations (i.e. **.then(...)**) which have been proposed in [7] and recently added to the standard:

Source Code 5

```
auto f1 = as_future(comm->isend(...));
auto f2 = f1.then([] (auto& f1)
{
    return as_future(comm->irecv(...));
});
auto f3 = f2.then([] (auto& f2)
{
    return as_future(comm->ibroadcast(...));
});
...
```

Here, **.then(...)** introduces an intermediate callback, providing the value of the future to its successor. Continuations enable complex sequences of MPI communications to be expressed in a flat manner, in which the resolution of each future triggers the next, reminiscent of the pipeline pattern. Community feedback to the MPI forum shows interest in integrating MPI into native C++ workflow³, which is basically the syntax in Source Code 5.

4 MPI INTERFACE EXTENSIONS

The interfaces proposed in this section follow two objectives: (1) the detach interface allows to use non-blocking communication as

locally asynchronous communication; (2) the progress interface allows triggering progress in MPI from another parallel runtime.

4.1 Detaching from non-blocking completion

As the fundamental concept of the detach interface, the application registers a callback for notification of local non-blocking request completion. The combined execution of a detach routine, such as **MPIX_Detach**, and the provided callback has the same semantics as a call to **MPI_Wait** or a call to **MPI_Test** returning **flag=true**, completing the request locally. Similar to wait and test calls, a detach call will consider a null or inactive request as immediately completed. In such case, the callback might be called immediately.

For consistent behavior, it is essential to ensure that the user does not interfere with the completion of any detached MPI requests—i.e., complete a detached request manually. Therefore, a detach call will consume *standard request* handles passed to it and set them to **MPI_REQUEST_NULL** on return, thus taking over full ownership of the requests. This behavior is similar to a call to **MPI_Request_free**, with the difference that upon local completion of the request the user is informed through the invocation of the callback function.

Persistent requests are handled differently, as these are also not freed on completion by wait and test calls, but rather set to an inactive state. To model the ownership transfer, we introduce next to active and inactive persistent requests a new *detached* state for requests. The user retains access to the handle—i.e., it is not reset on detachment—and can refer to it later in the code. To follow the behavior of the wait and test functions, **MPIX_Detach** marks active persistent requests as *detached*. Detaching an inactive request will immediately invoke the callback function. Using a detached request in any MPI function is erroneous. The invocation of the callback signals the completion of the request. At this point, a persistent request is marked inactive and any other request is deallocated.

In its simplest form, a single request is detached and handed to the MPI library along with a function pointer to the callback function that shall be executed on request completion of this specific request.

```
typedef void MPPIX_Detach_function(void *data);

int MPPIX_Detach(
    MPI_Request *request,
    MPPIX_Detach_function *callback,
    void *data);
```

The callback function type **MPPIX_Detach_function** takes as the single argument, which is the pointer to **data** provided to the detach call.

Existing completion routines—the **MPI_{Wait|Test}** family of calls—also provide a possibility to provide a status object, which will contain additional information about the completed message. For existing MPI functions, this means the application provides the storage, and the MPI library writes to the status object unless **MPI_STATUS(ES)_IGNORE** is provided. The status object is caller-owned, and this behavior would be desirable to retain. The use of **MPI_STATUS(ES)_IGNORE** indicates that the user is not interested in status information to be returned and can avoid the cost of collecting and filling the status information, so we also want to support this in the detach interface. For the existing completion routines, the user

³<https://github.com/mpi-forum/mpi-issues/issues/288>

has full control over the execution context of when the request is completed. Thus, it is no problem to provide a memory location on the stack or the heap. With a detached request, the user explicitly gives up the control over the completion context, so status objects allocated on the stack become problematic, as request completion may occur outside of the execution scope where the stack variable was defined.

To retain the ability to allocate a status object either on the heap or on the stack, the detach interface provides an orthogonal set of functions and callbacks to allow for the return of a status object to the user. However, to keep the concept of caller-owned status objects—in this case, the detach runtime is the caller of the completion—the MPI library will provide storage for the status object based on the call used to detach a corresponding request, and pass this to the callback function on invocation. The callback executes any status-dependent code before the runtime frees the status object on return from the callback function. For type safety, we propose a variant of the detach function type to ask for the status object to be provided to the callback function:

```
typedef void MPIX_Detach_status_function(
    void *data,
    MPI_Status status);

int MPIX_Detach_status(
    MPI_Request *request,
    MPIX_Detach_status_function *callback,
    void *data);
```

MPI_Waitsome, **MPI_Waitany**, and their analogous calls of the test family allow us to wait for arbitrary completion of one or more of multiple requests. We will typically use those functions to continue execution based on which requests finished. For the detach interface, similar functionality can be achieved by detaching multiple requests using **MPI_Detach_each**. This function registers a single callback for an array of requests and an associated array of data objects. Similar to the detachment of single requests, the registered callback is invoked for each (hence the name) request completion. Here, the detach interface deviates from the interface for wait and test. The calls to **MPI_Waitsome** and **MPI_Testsome** may return with multiple completion for two reasons: (1) to reduce the overhead of calling into the MPI layer multiple times for each pending request (as needed with **MPI_Waitany** and **MPI_Testany**, respectively), and (2) to reduce the chance of starving a single client in a client-server scenario with one client's requests being repeatedly favored by the selection of a single completed request. The former situation of additional overhead does not apply in this situation, as an implementation may implement the invocation of the callback on a desired low level of the runtime, hence already minimizing the overhead created by calling into the top layer interface of MPI. The latter also does not necessarily apply to the detach interface either, as here the runtime can (and should) ensure that no request completion is held back indefinitely—e.g., by employing the round-robin principle—and then immediately invoke the callback. Therefore, an MPI library can already implement a starvation-free method of completing multiple pending requests. In symmetry with the single request detach interface, separate detach procedures are available for completing with and without status information.

```
int MPIX_Detach_each(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *array_of_data[]);

int MPIX_Detach_each_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_status_function *callback,
    void *array_of_data[]);
```

In some cases, the application depends on all pending requests to be finished, as expressed by a call to **MPI_Waitall** or **MPI_Testall**. The detach interface proposed here provides a detachment mechanism with similar semantics in that just a single callback is registered and only invoked once when all corresponding requests completed. Again, in symmetry to the rest of the detach interface, separate detach procedures are available for completing with and without status information. However, as the callback function receiving status information registered will need to process multiple status objects, a separate callback function type ensures type safety.

```
int MPIX_Detach_all(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_function *callback,
    void *data);

typedef void MPIX_Detach_all_statuses_function(
    void *,
    int,
    MPI_Status *);

int MPIX_Detach_all_status(
    int count,
    MPI_Request array_of_requests[],
    MPIX_Detach_all_statuses_function *callback,
    void *data);
```

The integer argument of the callback provides the length of the status array. The value is the same as passed to the corresponding call of **MPIX_Detach_all_status**.

4.2 Driving Progress

As discussed in Section 2.3, MPI provides only weak progress guarantees and does not provide a specific call to advance pending communication. In case the MPI library uses hardware support or a progress thread to drive asynchronous progress, the user does not need to call into MPI itself to ensure progress. In the style of the Deep Computing Messaging Framework (DCMF) [11], therefore, we propose the specific procedure call **MPIX_Progress** which users can employ specifically if they want to communicate opportune moments for the MPI library to provide progress.

```
int MPIX_Progress(void *data);
```

This non-blocking progress function can be employed by polling services of tasking runtimes such as OmpSS-2 to drive progress even without particular knowledge about specific pending requests.

The only requirement from an application perspective is that the completion callback will finally be called after completion if this function is called repeatedly. If an implementation provides other means of progress than just by calling into the MPI library, this procedure may be implemented as a no-op. However, in case an implementation does provide communication progress when calling into the MPI library, a call to this procedure should do exactly that. While the use of **MPI_Test** or **MPI_Iprobe** will provide some form of progress on common MPI implementations, it is not required by MPI, and is rather a matter of quality of implementation. This call therefore enables an easy and clear way of enabling MPI to drive the communication, as needed in the code example 2, where the repeated call to **MPI_Test** allows the MPI library to provide progress. This function could also allow us to receive the callback on a thread with low utilization if the polling service has some knowledge about the utilization of threads. Finally, by providing an explicit call to drive communication progress, it will be easier for performance tools to filter such general progress driving calls and potentially optimize its wrapper implementation.

The signature of **MPHX_Progress** follows the callback type **polling_service_t** defined and used in Nanos6. The ***data** argument is an opaque argument, which is passed while registering the polling service and then provided to the callback. Although we do not use the argument in our current implementation, we might use the argument to restrict polling, for example, to make progress on a specific communicator.

5 IMPLEMENTATION

In this section we discuss three aspects of our implementation of the detach interface: (1) a core library, implementing the functions proposed in the previous section, (2) how we handle persistent requests in this core library, and (3) experiments to integrate this functionality into Boost.MPI.

5.1 MPI detach interface

We provide a proof of concept implementation⁴ for the proposed interface. The code is independent of a specific MPI implementation but needs to be built against the MPI library in use. It can be either compiled into the application or built as a library and linked into the application. The implementation is thread-safe and can be used in two modes to drive progress: (1) with an external polling service, and (2) with an internal progress thread.

As mentioned in Section 4.2, MPI currently provides no explicit interface to drive progress. We therefore proposed and implemented the call to **MPHX_Progress** as a context-free call into the MPI communication layer. This call can then be called by the runtime at regular intervals to ensure progress.

This call can come from an external polling service, as provided by OMPSS-2⁵ or Boost.Asio². In this case, the required threading level for MPI depends on the behavior of the polling service. When considering the MPI threading level, we must include all MPHX functions proposed in this paper to the set of MPI functions.

MPI_THREAD_FUNNELED or even **MPI_THREAD_SERIAL** might be sufficient if the polling service is only active on the master thread and the application also only communicates on the master thread.

This call can also come from a progress thread spawned by the library itself. If the progress thread is used, the necessary threading level is **MPI_THREAD_MULTIPLE**. If the environmental variable **MPHX_DETACH** is exported and set to **progress**, the library will start a progress thread to poll on **MPHX_Progress** and make progress for the detached requests. This progress thread takes advantage of internal knowledge and suspends itself if no detached request is in the queue. A conditional variable controls the suspension of the polling thread. The detach functions signal to this conditional variable when requests are enqueued. This is done via condition variables that are modified when requests are queued or completed. As long as there are detached requests in the queue, the progress thread will repeatedly call **MPHX_Progress**. However, if the application also uses non-blocking communication outside of the detach runtime, the progress thread might be suspended by the detach runtime while these other requests are still pending and could benefit from a progress thread. In such a scenario, a polling service might be advantageous.

The implementation of the various proposed detach functions directly test for completion of the communication when called. In case the corresponding completion is already returned from the MPI library, the provided callback is called immediately. Otherwise, the provided requests are enqueued. The implementation uses two distinct queues for (1) for single requests and (2) for multiple requests with *wait-all* semantics.

During lazy initialization, i.e., when one of the detach functions is first called, the library decides whether to launch a progress thread. The progress function tests all requests in the queue for completion and calls the corresponding callback in case of completion. If the polling thread is used, **MPI_THREAD_MULTIPLE** is necessary because of the **MPI_Test** calls performed by the polling thread. To avoid the polling thread to become a possible bottle-neck, the application developer needs to make sure, that the callback does not cause excessive execution on this thread, but rather schedules work to be executed by the application threads.

In the current implementation, the *detach-each* functions behave like repeated calls to the *detach* functions. Here, the implementation tests each request individually using **MPI_Test**, for both variants with and without a status object.

The *detach-all* functions need to be handled differently, as the call to the callback depends on the completion of all provided requests. In the case that no status is requested, we can easily compress the vector of requests by removing finished requests from the vector of requests. For this purpose, we use **MPI_Testsome**. In the case that the application requests the status, the use of **MPI_Testsome** would mean rather complex book-keeping of indexes and re-mappings. As proof of concept, the implementation uses **MPI_Testall** in this case. This provides all required statuses when all requests are finished.

5.2 Handling persistent requests

Compared to standard requests, handling persistent requests provides some special requirements. As the detach functions react differently to the two types of requests, the implementation needs

⁴<https://github.com/RWTH-HPC/mpi-detach>

⁵<http://pm.bsc.es/ompss-2>

to be able to distinguish between them. Any non-persistent request is set to `MPI_REQUEST_NULL` before the detach function returns. Persistent requests are not reset but returned as-is to the user. However, the user is not allowed to complete this request while the detach runtime processes it. An MPI implementation can infer from inside knowledge whether a request is persistent or not. However, our prototype implementation only acts as a shim layer on top of MPI using the PMPI interface, and it, therefore, does not have access to such low-level information. Unfortunately, MPI does not provide a function to check whether a request is persistent. Also, there is no possibility to attach an attribute to a request, as it is possible for communicators, windows, and datatypes. Such an attribute system would enable our prototype to intercept the init functions for persistent requests and attach an attribute to the request, allowing state-less tracking of persistent requests. In our implementation, we use a map to store persistent requests at creation time and look up every request passed to the library in this map. We can skip the lookup as long as the map is empty, i.e., the process did not create a persistent request yet. Full integration into the MPI library would provide potential for optimization in this regard.

5.3 Boost interface

In addition to the C interface, we provide several extensions to Boost.MPI². The detach functions are wrapped by replacing the function pointer argument with a `std::function` which enables lambda captures. The additional state is stored as part of the data argument of the C interface (i.e., the `void*`). We furthermore enforce type safety on the user data through template parameters, leading to the following signature for the singular detach:

Source Code 6

```
template <typename type>
void detach(
    boost::mpi::request& request ,
    const std::function<void(const type*)>& callback,
    const type& data);
```

In case the uniform call syntax proposal [20] gets accepted in a future standard, it would be possible to call the functions as members of `boost::mpi::request`, omitting the first argument. Until then, we overload the `operator/` of the request to imitate extension functions. For convenience purposes we also overload the `boost::mpi::communicator::isend()/irecv()` to support detach natively, adding a callback argument to each specialization.

The futures implementation is based mainly on applying the concept presented in Section 3.2 to each function; the promise state is kept internally, and its coupled future is handed back to the caller. Although continuous futures have already been accepted into the standard, the majority of STL implementations do not yet provide support for it. Hence we have utilized Boost.Thread's² version, which we intend to replace with the standard later on.

Aside from the concurrency features, the recent networking technical specification [21] introduces the `std::io_context`, a dispatcher which may be utilized to call `MPIX_Progress()` in regular intervals. We provide an example implementation on top of Boost.ASIO² which the proposal is based upon. The user is expected to construct the `boost::mpi::detach_service` once for explicit

management of progress, and may then call the detach functions freely.

6 EXPERIMENTS

We test the implementation of our proposed implementation with two different applications. First, we show improved readability as well as improved performance when using MPI detach with a distributed Blocked Cholesky Factorization using OpenMP tasks with fine-grained dependencies. We perform detailed performance studies using this code. During the experiment, we also fixed a race condition in the detached task implementation in the LLVM/OpenMP runtime.

As a second experiment, we modify a distributed memory particle tracing application to use MPI non-blocking communication. The use of the new MPI detached functionality allows for cleaner C++ code.

6.1 Environment

We run the experiments on dual-socket nodes of Claix-18, which are equipped with two 24 core Intel Skylake Platinum 8160 CPUs and 192GB memory. The nodes have hyper-threading disabled, and sub-NUMA clustering enabled. The operating system is CentOS 7.7.

For the experiments, we use various custom-built compilers. Clang is built from git commit d23131a on master after applying a patch from LLVM phabricator⁶. Support for the detach clause was added after the LLVM/10.0 release; therefore, we choose to build from git. Overall, the version of clang includes fixes for two race conditions in the OpenMP runtime, that we identified during our experiments.

OmpSs is built from the 19.06 release tarball. OmpSs-2 is built from git commit 59000cc on master.

All versions of the code use the Intel MKL in version 19.0.1.144 for the math kernels.

6.2 Blocked Cholesky Factorization

Code. Schuchart et al. [18] provide two MPI-distributed and taskified versions of Blocked Cholesky Factorization. One version funnels all MPI communication through the master thread (*singlecom*); the other version has fine-grained dependencies and performs non-blocking communication in tasks (*taskyield*). The latter version uses OpenMP taskyield and MPI test similar to Source Code 2.

We extend the taskyield code and integrate our MPI detach implementation into the code. The necessary changes to the code are conceptually already presented by the transition from Source Code 2 to Source Code 3, where we use `MPI_Detach` instead of `MPI_Foo`. OmpSs-2 provides a detach interface similar to OpenMP detached tasks, which we use interchangeably in the *detach* setup. The used codes are available in the `mpi-detach` branch of our fork⁷.

Build configurations. The baseline is the *singlecom* code, which performs all MPI communication on a single thread. We built and executed this code with Clang, OmpSs, and ICC (since the results for clang and ICC are almost the same, we only report the data for

⁶<https://reviews.llvm.org/D80480>

⁷https://github.com/RWTH-HPC/cholesky_omptasks

Figure 1: Node scaling measurements of distributed Blocked Cholesky Factorization

Clang). From time to time, the execution of the Clang/ICC builds deadlocked, because of a race condition in the OpenMP runtime. We submitted a patch for review⁶.

The *taskyield* code represents a non-blocking implementation based on existing OpenMP functionality. The code only reliably executes when built with OmpSs, and deadlocks with Clang/ICC because of the way the runtime implements taskyield.

The *detach* code uses non-blocking communication and our proposed MPI detach functions to complete tasks automatically when the communication completes. This functionality is available with Clang and OmpSs-2. In this setup, we use the polling thread from our library for progress.

The OmpSs-2 runtime furthermore allows to register a callback with a polling service. We use this service instead of our polling thread in the *polling* configuration.

Run configurations. We perform two scaling measurements. Each measurement point represents ten repetitions, and we plot the average and the standard deviation of the measured time. The nature of the decomposition code only allows powers of two for the number of MPI processes.

The first measurement series fills a single node with 48 threads, which are equally distributed to a varying number of 1 to 16 MPI processes. We run this series with an $n \times n$ matrix of size $n = 32\,768$ and $b \times b$ blocks of size $b = 256$. This results in 16 384 blocks.

The second measurement series uses a fixed number of 12 threads per MPI process and scales from 1 to 32 nodes. We run this series with a matrix of size $n = 65\,536$ and blocks of size $b = 256$. This results in 65 536 blocks. As the complexity of the problem is $O(n^3)$, the expected execution time for the second series is eight times the execution time of the first series.

The reported number of threads is the number of dedicated cores associated with the process by the batch system, e.g., by using **taskset**. For OpenMP applications, we export this value as **OMP_NUM_THREADS**. For OmpSs-2, there is no documented way to control the number of threads.

For easier evaluation and reproducibility, we used the Jülich Benchmarking Environment (JUBE) [12] in version 2.2.2 to configure and run the measurements in the jube directory⁷.

Results. Figures 1 and 2 show the collected results. We distinguish the time spent in the compute kernels in MKL (**calc(mkl)**) from the remaining time (**noncalc**). We expect the same execution time for the compute kernels independent of the compiler and build configuration, as we always use the same MKL routines. There is no redundant computation caused by the distribution. This is expectation is held within the scope of measurement errors. The plotted bars show time averaged over the executing threads and 10 repetitions. The white error bar shows the standard deviation in the calculation time across all threads; this is a result of an imbalance in the distribution between processes and an imbalance in the scheduling on the available threads. The black error bar shows the standard deviation of the overall execution time.

We do not measure the time of the polling thread explicitly, but the polling thread will always share a core with a calculating thread due to the process placement. So, the polling thread is included in the execution time of the measured threads.

For the node scaling experiment in Figure 1, we can see the influence of memory accesses across NUMA domains when executing with just one MPI process. With two MPI processes, we still fill one socket with threads and can see a slight impact of sub-NUMA clustering. For this reason, we chose the configuration with four MPI processes per node for the strong scaling experiment. We can see that the overhead of the *detach* code with Clang has the lowest overhead across all configurations. From our understanding, the high overhead of the OmpSs versions for a single MPI process is a result of creating the dependency graph for a large number of tasks. The number of tasks halves with doubling the number of processes, therefore this overhead decreases. For the other process configurations, we can see a performance improvement from *singlecom* to *taskyield*. For the clang versions we don't see such overhead, because the runtime chooses to execute a task immediately when a threshold is reached.

For the strong scaling experiment in Figure 2, we show two representations of the data. In Figure 2a, we plot the average execution time of each single thread or, respectively, the wallclock time of the code, where we expect an inversely proportional decrease of the calculation time (**calc**). In Figure 2b, we plot the aggregated overall

Figure 2: Strong scaling measurements of distributed Blocked Cholesky Factorization

execution time of all threads, where we expect constant calculation time independent of the number of processes or threads. We see a slight increase in the calculation time for 128 processes.

In the first plot, we observe a constant and low per thread overhead for the *detach* code with Clang among 4 to 32 processes, and a slight increase for higher process counts. In the second plot, we can observe how this constant per thread overhead rapidly amounts to significant overhead for the overall execution. The *detach* code with Clang again outperforms most other codes. Only the *taskyield* code with OmpSs provides less overhead for high process count. Unfortunately, this code version is not deadlock-free, based on a strict interpretation of the OpenMP specification, and therefore not portable to other OpenMP implementations.

The experiment also shows that the use of a polling service seems to provide slight performance benefits for this code.

Discussion. Overall the scaling experiment shows that an implementation of the algorithm based on our proposed extension can outperform any portable implementation based on currently available interfaces. We cannot consider the taskyield approach portable, as it relies on a specific implementation of the OpenMP taskyield construct. At the same time, this taskyield implementation shows us that we do not reach possible performance limits with our implementation. We think, that an integration of the interface into an MPI implementation and an integration of a polling service in the OpenMP implementation can solve this issue and provide the possible performance. This is subject to further investigation.

6.3 Particle tracer

Code. We have modified an MPI-distributed particle tracer [4] to support non-blocking communication through detach. The particle tracer utilizes Boost.MPI² for distributed-memory and Intel TBB [15] for shared-memory operations. It is intended to run with one process per compute node, with the processes saturating the cores and memory of their nodes. Hybrid parallelism is achieved through partitioning the domain (vector field) to the processes statically, and then balancing the work (particles) across the partitions dynamically using a variant of diffusive load balancing [2].

Source Code 7: The core loop of the particle tracer repeatedly performs load balancing, allocation of curves, integration of curves in the vector field and redistribution of particles

```
auto dpa::particle_tracer::trace(
    const dpa::vector_field& local_vector_field,
    std::vector<dpa::particle>& particles) {
    dpa::integral_curves<dpa::particle> curves;
    while(!check_completion(particles)) {
        load_balance(particles);
        allocate_curves(particles, curves);
        integrate(local_vector_field, particles, curves);
        distribute_out_of_bounds(particles);
    }
    return curves;
}
```

Source Code 7 outlines the function ran by each process during integral curve generation. Regarding inter-process communication, the `check_completion()` function consists of two global collective operations; a gather for retrieving the active particle counts of each process, followed by a broadcast containing a Boolean evaluating to true when no process has active particles. Additionally, both the `load_balance()` and `distribute_out_of_bounds()` functions contain neighborhood all-to-all collectives where each process sends/receives particles to/from its first neighbors. Thus, detach is utilized with the `MPI_Igather()`, `MPI_Ibcast()`, `MPI_Ineighbor_alltoall()` and `MPI_Ineighbor_alltoallv()` non-blocking collectives. The three communication points are factored to be sequential, and converted to continuous futures using the interface presented in Section 5.3.

Configuration. The execution times of the original and modified codes are measured in a fixed setting to assess the performance overhead of detach. The number of processes/nodes is fixed to 64 for the experiment. The vector field has a size of 1024^3 . The seed generation is performed uniformly with a stride of $[4, 4, 4]$, leading to a total of 16.78 million particles. The integrator is set to Runge-Kutta 4, with a fixed step size of 0.001. The maximum iteration count is set to 1000. We apply each variant to three test datasets, and compute the average total runtime over ten measurements.

Results. The mean execution time for the original, blocking particle tracer is measured as 74.05 ± 1.64 , 45.65 ± 1.05 and 32.85 ± 1.47 seconds for the three datasets, whereas for the detach version, the times are 77.29 ± 1.98 , 48.71 ± 1.13 and 35.58 ± 1.95 seconds respectively. The results indicate that the conversion to futures introduces a slight overhead, although negligible with respect to the rest of the experiment. Yet, the non-blocking implementation enables optimizations such as passive termination detection, which may, in turn, compensate for the increased execution time.

7 CONCLUSIONS

Initially driven from a C++ application use case, we proposed a new interface for MPI in this paper to support asynchronous control flow in MPI applications. Our proposed interface provides a clear separation of distributed memory communication provided by MPI and the various asynchronous execution concepts such as OpenMP tasks or C++ futures. We presented a proof of concept implementation, which works as a PMPI wrapper library. By integrating the interface into OpenMP and C++ codes, we demonstrated the applicability of the approach. We can see in the performance experiments that our proof of concept implementation introduces relatively low overhead when we compare a blocking code with a non-blocking code, which immediately waits for the future. This overhead is partially a result of the implementation as a shim layer rather than integrating the functionality directly into an MPI implementation. Integration into MPI implementation has potential for further optimization and therefore reduced overhead. The use of detach in the distributed Blocked Cholesky Factorization shows the real potential of this approach. Other than a taskyield based approach of non-blocking communication in OpenMP tasking programs, the

use of non-blocking communication in combination with the proposed detach functions is completely conforming with the OpenMP specification.

ACKNOWLEDGMENTS

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement 824080.

REFERENCES

- [1] M. B. Baker et al. 2017. OpenSHMEM Specification 1.4. (12 2017). <https://doi.org/10.2172/1460190>
- [2] George C. 1989. Dynamic load balancing for distributed memory multiprocessors. *J. Parallel and Distrib. Comput.* 7, 2 (1989), 279 – 301. [https://doi.org/10.1016/0743-7315\(89\)90021-X](https://doi.org/10.1016/0743-7315(89)90021-X)
- [3] James D., R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. 2014. Enabling communication concurrency through flexible MPI endpoints. *Int. J. High Perform. Comput. Appl.* 28, 4 (2014), 390–405. <https://doi.org/10.1177/1094342014548772>
- [4] A. C. Demiralp, D. N. Helmrich, and J. Protze. 2019. DPA - the distributed particle advector. <https://devhub.vr.rwth-aachen.de/VR-Group/dpa>.
- [5] Message Passing Interface Forum. 2015. *MPI: A Message-passing Interface Standard, Version 3.1*.
- [6] R. E. Grant, M. G. F. Dosanjh, M. J. Levenhagen, R. Brightwell, and A. Skjellum. 2019. Finepoints: Partitioned Multithreaded MPI Communication. In *High Performance Computing - 34th Intl. Conf., ISC High Performance 2019, Frankfurt/Main, Germany, June 16-20, 2019, Proc.* 330–350. https://doi.org/10.1007/978-3-030-20656-7_17
- [7] N. Gustafsson, A. Laksberg, H. Sutter, and S. Mithani. 2013. *Improvements to std::future<T> and Related APIs*. Technical Report. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3784.pdf>
- [8] M.-A. Hermanns, M. Geimer, B. Mohr, and F. Wolf. 2017. Trace-Based Detection of Lock Contention in MPI One-Sided Communication. In *Tools for High Performance Computing 2016*. Springer Intl. Publishing, Cham, 97–114. https://doi.org/10.1007/978-3-319-56702-0_6
- [9] ISO. 2016. *Programming Languages — Technical Specification for C++ Extensions for Concurrency*. Technical Report. <https://www.iso.org/standard/65242.html>
- [10] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller. 2020. CHAMELEON: Reactive Load Balancing for Hybrid MPI+OpenMP Task-Parallel Applications. *J. Parallel Distributed Comput.* 138 (2020), 55–64. <https://doi.org/10.1016/j.jpdc.2019.12.005>
- [11] S. Kumar et al. 2008. The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer. In *Proc. of the 22nd Annual Intl. Conf. on Supercomputing, ICS 2008, 2008*. 94–103. <https://doi.org/10.1145/1375527.1375544>
- [12] S. Lührs, D. Rohe, A. Schnurpfeil, K. Thust, and W. Frings. 2016. Flexible and Generic Workflow Management. In *Parallel Computing: On the Road to Exascale (2015-09-01) (Advances in parallel computing)*, Vol. 27. Intl. Conf. on Parallel Computing 2015, Edinburgh (United Kingdom), 1 Sep 2015 - 4 Sep 2015, IOS Press, Amsterdam, 431 – 438. <https://doi.org/10.3233/978-1-61499-621-7-431>
- [13] OpenMP Architecture Review Board. 2018. *OpenMP Application Program Interface Version 5.0*. Specification.
- [14] S. Pellegrini, R. Prodan, and T. Fahringer. 2012. A Lightweight C++ Interface to MPI. In *2012 20th Euromicro Intl. Conf. on Parallel, Distributed and Network-based Processing*. 3–10.
- [15] J. Reinders. 2007. *Intel Threading Building Blocks* (first ed.). O'Reilly & Associates, Inc., USA.
- [16] K. Sala, J. Bellón, P. Farré, X. Teruel, J. M. Pérez, A. J. Peña, D. J. Holmes, V. Beltran, and J. Labarta. 2018. Improving the Interoperability between MPI and Task-Based Programming Models. In *Proc. of the 25th European MPI Users' Group Meeting, 2018*. 6:1–6:11. <https://doi.org/10.1145/3236367.3236382>
- [17] K. Sala, X. Teruel, J. M. Pérez, A. J. Peña, V. Beltran, and J. Labarta. 2019. Integrating Blocking and Non-Blocking MPI Primitives with Task-Based Programming Models. *CoRR abs/1901.03271* (2019). [arXiv:1901.03271](https://arxiv.org/abs/1901.03271) <http://arxiv.org/abs/1901.03271>
- [18] J. Schuchart, K. Tsugane, J. Gracia, and M. Sato. 2018. The Impact of Taskyield on the Design of Tasks Communicating Through MPI. In *Evolving OpenMP for Evolving Architectures - Proc. of the 14th Intl. Workshop on OpenMP, IWOMP 2018*. 3–17. https://doi.org/10.1007/978-3-319-98521-3_1
- [19] A. Stepanov and M. Lee. 1995. *The standard template library*. Vol. 1501. Hewlett Packard Laboratories 1501 Page Mill Road, Palo Alto, CA 94304.
- [20] B. Stroustrup and H. Sutter. 2015. *Unified Call Syntax: x.f(y) and f(x,y)*. Technical Report. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4474.pdf>
- [21] J. Wakely. 2018. *Working Draft, C++ Extensions for Networking*. Technical Report. <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4771.pdf>