On the integration of lightweight tasks with MPI using the C++26 std::execution ‘Senders’ API

John Biddiscombe
Swiss National Supercomputing Centre
ETH Zürich
Lugano, Switzerland
john.biddiscombe@cscs.ch

Mikael Simberg
Swiss National Supercomputing Centre
ETH Zürich
Zurich, Switzerland
mikael.simberg@cscs.ch

Auriane Reverdell
Swiss National Supercomputing Centre
ETH Zürich
Zurich, Switzerland
auriane.reverdell@cscs.ch

Raffaele Solcà
Swiss National Supercomputing Centre
ETH Zürich
Lugano, Switzerland
raffaele.solca@cscs.ch

Alberto Invernizzi
Swiss National Supercomputing Centre
ETH Zürich
Lugano, Switzerland
alberto.invernizzi@cscs.ch

Rocco Meli
Swiss National Supercomputing Centre
ETH Zürich
Lugano, Switzerland
rocco.meli@cscs.ch

Joseph Schuchart
Institute for Advanced Computational Science
Stony Brook University
Stony Brook, NY, USA
Innovative Computing Laboratory
University of Tennessee
Knoxville, TN, USA
joseph.schuchart@stonybrook.edu

Abstract

Integrating asynchronous MPI messaging with tasking runtimes requires careful handling of request polling and dispatching of associated completions to participating threads. The new C++26 `Senders` (`std::execution`) library offers a flexible collection of interfaces and templates for schedulers, algorithms and adaptors to work with asynchronous functions—and it makes explicit the mechanism to transfer execution from one context to another—essential for high performance. We have implemented the major features of the `Senders` API in the `pika` tasking runtime and used them to wrap asynchronous MPI calls such that messaging operations become nodes in the execution graph with the same calling semantics as other operations. The API allows us to easily experiment with different methods of message scheduling and dispatching completions. We present insights from our implementation on how application performance is affected by design choices surrounding the placement, scheduling and execution of polling and completion tasks using `Senders`.

This work was supported by the Swiss Platform for Advanced Scientific Computing (PASC).

CCS Concepts

• **Software and its engineering** → **Concurrent programming languages**; **Data flow languages**; **Data flow architectures**; **Cooperating communicating processes**; • **Networks** → Network performance analysis.

Keywords

C++, MPI, Senders, Asynchronous, Task, Runtime, Thread Pool

ACM Reference Format:
John Biddiscombe, Mikael Simberg, Auriane Reverdell, Raffaele Solcà, Alberto Invernizzi, Rocco Meli, and Joseph Schuchart. 2025. On the integration of lightweight tasks with MPI using the C++26 std::execution ‘Senders’ API. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3731599.3767394

1 Introduction and Motivation

One of the recurring difficulties of integrating MPI [4] with asynchronous programming frameworks is the efficient handover of execution between threads of computation and communication operations and the scheduling of the task graph that represents the underlying algorithm/s being executed. In this paper we show how this can be achieved using the C++ standard `Senders` proposal

P2300R10 [7] that lays out an API for describing task graphs and their execution on diverse resources.

Some applications consist of a fairly straightforward sequence of iterations that ping pong between computation and communication (usually halo exchange) and these steps can be overlapped reasonably trivially by careful placement and testing of asynchronous MPI requests. When the execution graph becomes more complex and (in particular) more dynamic, the dataflow or Continuation Passing Style (CPS) [1] of programming can help avoid synchronization points that result in wait states and idle resources. Tasks are executed when their input data/dependencies become available, but the triggering of associated continuations must work not only with local data/task dependencies, but also with remote data that is delivered via MPI requests upon which some other task is waiting.

Many tasking libraries and runtimes have addressed this issue, but the solution generally involves application (or framework) specific APIs that users must adapt their code to fit, which in turn leads to incompatibilities between competing libraries—even though the underlying concepts are usually very similar. The `Senders` proposal presents a standardized C++ API that allows asynchronous libraries to be developed *independently*, but still share a common interface to make seamless interoperability possible (as well as supporting C++ coroutines). We have wrapped the parts of the MPI library that use asynchronous `MPI_Request` in a thin adapter such that those MPI calls return a `sender` which can be inserted into the operation pipeline alongside other tasks. The polling for completion of MPI requests is handled internally by the task scheduler which can either execute a continuation immediately, or schedule it for later. Both execution and polling may happen on a dedicated (for networking) thread pool, or a thread pool shared with the main worker threads used for application computation. There are numerous possible combinations of how, where, and when to poll and execute the work associated with MPI requests and the `Senders` API gives us a flexible way to schedule and process work on the polling thread or transfer it to another.

The motivation for this work lies in the development of a task-based distributed linear algebra library, DLA-Future [20, 21] that is written in modern C++ and uses `sender` abstractions throughout, with the goal of being entirely compatible with any implementation of the new C++ asynchronous API, and thus being portable to any runtime that might implement it (e.g. `pika` [19], `HPX` [10], `Qthreads` [23], `PaRSEC` [3], etc.) or even the default `std::execution` implementation provided by compiler `std` library providers when it becomes generally available. The initial integration of MPI within DLA-Future used a simple approach to polling that worked well, but was not implemented in a `Senders` compatible way and since it easy to wrap any asynchronous operation, be it on host, device, accelerator or even network stack, we replaced it with a `Senders` based version and experimented with different ways of triggering and polling for work, to see how they affected performance.

The following sections discuss the implications and trade-offs of the different ways of handling/processing MPI requests in `pika`, and although much of the discussion is specific to the `pika` implementation, the broader concepts and API should be equally valid for other runtimes and the main aim of this paper is to explain and illustrate how to best exploit the `Senders` API in conjunction with a task framework of *any* kind.

1.1 Related work

The `pika` library is derived from `HPX` and shares many of its features, as well as inheriting some of its basic task-scheduling and event polling logic. The MPI polling in this paper was originally implemented using `futures` and contributed to `HPX` by some of the authors, but as the `std::execution` asynchronous API evolved and introduced `Senders` in favour of `futures`, `pika` was created to focus on a stricter adherence to the C++ standards using `Senders` only. Whilst `HPX` supports asynchronous messaging as part of the task execution graph, it makes use of `parcelports` (such as LCI [24] and libfabric [2]) that implement a custom messaging API primarily intended to work with active messages [22], rather than passive messaging via simple MPI calls. Charm++ [11] is similar in this respect in that it provides a custom API for messaging that integrates directly with the tasking framework and is not directly compatible with other runtimes. Other tasking libraries such as `PaRSEC` have also experimented with replacing MPI with custom solutions (see [13]) in an attempt to improve latency and throughput.

In this work we focus on integrating native MPI based messaging into the task system by wrapping it directly in `Senders` so that applications may port code using MPI into an asynchronous/-task model with only trivial modifications to their MPI calls. Note that integrating existing MPI codes with a runtime is discussed at length in [12] using the VT framework—with the emphasis being on ensuring that existing code/libraries using MPI can be cleanly embedded (unchanged) without the need for a rewrite. Careful attention is made to preventing deadlocks due to overlapping or competing uses of collective communications in sub-sections of the code and ensuring that forward progress guarantees are provided. Whilst `pika` does also support calling external libraries that use MPI, we concern ourselves here with the API level changes that affect how the code is structured for modern C++ and not with interoperability/compatibility with legacy or external libraries.

The question of whether to use a dedicated thread for polling was asked in [6] and we revisit this question and place it in the context of user level thread pool management rather than at the level of the MPI library or OS kernel responsibility.

Our approach to wrapping MPI calls bears a similarity to the work presented in [5] since we use C++ templates to intercept/abstract asynchronous MPI calls and redirect them through the `Senders` API, but we primarily deal here with polling and synchronization of calls and not with higher-level constructs such as lifetime memory management of buffers (though `pika` does prevent buffers going out of scope during asynchronous calls when possible).

TAMPI [16] provides an abstraction layer that allows the user to integrate both blocking and non-blocking calls to MPI within their task model and the work described in [15] offers a similar capability, by suspending worker threads that are waiting for blocking MPI functions and scheduling other tasks on the freed core. We do not optimize the use of blocking MPI calls in this work since we are principally interested in APIs for asynchronous tasks and discourage the use of blocking operations, the use of which can be handled in a similar manner to the methods discussed in those papers.

Recent work on ExaMPI [17] aims to provide a native C++ implementation of MPI that can abstract away progression of requests without user intervention. Our work complements this by providing the ideal interface to bind the user level code with the backend runtime and threading engine that actually handles transport. Likewise, extensions to support MPI continuations supplied by the user and triggered directly by the MPI library as discussed in [18] complement this work, and in fact, we support the use of those continuations and test the Open MPI implementation described therein, with some results in this paper. Another related work is MPI detach [14] where the MPI API has again been extended to provide a callback interface and give responsibility of triggering of continuations to the MPI implementation. Where our work differs is primarily that we are not attempting (yet) to rewrite MPI itself, but embed it in our tasking runtime and demonstrate the utility of the new C++ API—though it is clear that an improved asynchronous MPI API (for C++ at least) is one of the main objectives of this research as well as those mentioned previously.

2 Implementation overview

2.1 Introduction to Senders

Note that in the following discussion, the `pika` implementation described is based on revision 10 of the P2300 standards proposal and some names may change before C++26 is available. The nomenclature used in this paper should not be considered final. Although some implementation details might change, the basic concepts will remain the same even if the API is updated. The reader is also referred to P3149 [8] for additional information about proposed APIs for structured asynchrony. Note also that some code snippets are simplified for clarity and are not expected to compile verbatim.

At the simplest level, a `sender` represents a kind of abstraction for a function call: in terms of a task graph, when some operation completes, it should trigger one or more other operations, which in turn trigger others and so on—the execution of each dependent task is in reality just a function call operation. However, when tasks are executed (asynchronously in our case) on resources such as a thread, thread pool, accelerator etc. and allowed to run concurrently with other work and move from one device to another, that function call abstraction needs to be extended to support posting/triggering work rather than directly executing it. There also need to be mechanisms to initiate execution, specify where it should happen (offload it if necessary), and eventually retrieve a return value or error, or simply synchronize with it when it completes. To this end, the C++ `std::execution` framework defines mechanisms to start work on an `execution_context` (a resource on which work may take place such as the thread pool just mentioned) via a `scheduler` that is a handle that can be used to start work on the `execution_context`. A `scheduler` in `std::execution` should not be confused with a scheduler that might be found in a runtime such as `pika` (or `HPX`, `Qthreads` etc.)—the schedulers in those runtimes are responsible for managing queues of tasks, stack space, priorities, additional dependency resolutions and other low level details that are implementation-specific. The `std::execution` scheduler acts as an interface that is used to pass work from the users’ declarations down into the runtime’s scheduler.

`Senders` are similar to CUDA graphs [9] in that a complete sequence of operations are defined as a directed acyclic graph (DAG) and then submitted in a single call to the framework’s scheduler. This differs for example from `hpx::future` which can have continuations attached to it before, during or after it has started executing—this makes `Senders` slightly more demanding to work with, but has the advantage of reducing or removing the synchronization overheads associated with attaching continuations.

Construction of execution pipelines is straightforward: it requires the creation of an initial `sender` with output piped into a sender adaptor, and so on (a `sender` takes no input, but an adaptor transforms one sender into another). In the following toy example the `just` factory is used to create the initial `sender`, that ‘just’ passes a list of parameters through to the next operation, which in this case is our custom `pika` MPI transform adaptor that invokes the supplied callable (here `MPI_Isend`) with the received parameters, and when that completes, a `then` adaptor executes cleanup of the message buffer.

```
auto isend_snd =
  just(buf, size, type, dest, tag, comm) |
  transform_mpi(MPI_Isend) |
  then([buf]() { release_msg_buffer(buf);});
start_detached(isend_snd);
```

`just` is the starting point for many pipelines, and can even be used with an empty parameter list to simply start a chain of senders. The `|` operator connects senders together by invoking one of the following completion signals:

- `set_value`: passes a successful result of one sender to the next sender
- `set_error`: propagates errors through the pipeline, each adaptor may handle errors or pass them through to the next sender
- `set_stopped`: like `set_error` but indicates cancellation or intentional termination rather than an error

`sender` adaptors such as `transform_mpi` and `then` take callable objects and return a new `sender`. The final `sender` (held here in variable `isend_snd`) represents all the work to be done as a single object which can be scheduled for execution using a consumer like `start_detached` which is responsible for submitting the `sender` to the runtime and in this case drops (or detaches from) any final returned value.

In this example, we have not specified a scheduler, so work would start on the current execution context when `isend_snd` is submitted: `just` always completes immediately by invoking `set_value` with its parameters. This triggers `transform_mpi` which will invoke its callable (with relevant parameters) and when `transform_mpi` receives completion of the MPI request, it will invoke `set_value` with the result, signaling completion to `then`. `then` in turn calls `set_value` whereupon `start_detached` returns back to the caller. The crucial point to note here is that, whichever thread in the runtime calls the `set_value` from `transform_mpi`, is the thread that will then continue to execute the `then` adaptor and onwards until the pipeline completes (in the absence of any other transfers). This means that when `start_detached` returns, the execution context (thread) the code returns on may not be the same as the one it started on. Although this might seem like an asynchronous operation (and much work might take place on another thread), in fact, the start_detached call does not return control until the entire pipeline has completed and so any code after that call is effectively blocked until the MPI operation is done.

To make this pipeline properly non-blocking, we must include an explicit transfer of work via a scheduler to a different execution context. This can be done by starting the pipeline on a different scheduler, or by inserting a continues_on adaptor somewhere in the pipeline. We can pipe the output of just into a continues_on(pika::thread_pool_scheduler()) adaptor that transfers the subsequent work to a thread pool where it will then be executed. The part of the pipeline before continues_on will be executed on the thread submitting the work and the part after continues_on will be executed on the designated thread pool, but most importantly the call to start_detached will now return immediately after the work is submitted to the scheduler in the continues_on adaptor. The remaining work will take place asynchronously and statements after the start_detached call will not be blocked. In this example we would have transferred the invocation of MPI_Isend to the thread pool rather than invoking it directly on the current thread.

Note in particular the absence of an MPI_Request in our MPI send example: before examining the intricacies of how completions are triggered in our MPI sender pipelines, the transform_mpi adaptor must be explained.

2.2 Intercepting and handling MPI calls

Since our main goal is to provide a seamless integration of MPI into the Senders model, we want the runtime to handle all aspects of request management. Our transform_mpi adaptor therefore accepts a list of arguments (passed from a predecessor sender) and a callable (supplied on construction) that fits the asynchronous MPI pattern of accepting the provided arguments. An additional MPI_Request parameter, which transform_mpi handles internally, is passed as the final parameter. The adaptor can also be used as follows:

```
// params passed directly using 'just'
... | transform_mpi(just(buf, size, type, dest,
    comm), func)) | ...
```

Internally, transform_mpi performs a compile-time check to test if the callable func parameter can be invoked with the passed parameters, *plus an additional request parameter* using type checks:

```
template <typename F, typename... Ts>
bool is_mpi_request_invocable_v =
  std::is_invocable_v<F, Ts..., MPI_Request*>;
```

Since all common asynchronous MPI calls (including collectives) follow this pattern, we can accept a broad range of MPI functions as well as user supplied wrappers with MPI_Request as the final parameter in the argument list (a feature exploited in DLA-Future to manage lifetimes of some buffers by wrapping MPI calls in lambdas with additional logic for lifetime management). The MPI_Request is stored internally by transform_mpi and handled (depending on the polling mode used) by the runtime as described later.

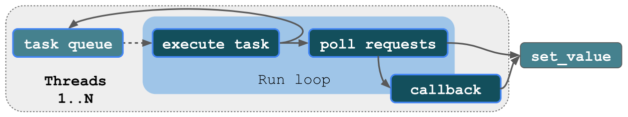

Figure 1: All threads can execute tasks and optionally perform a poll for requests. Polling + callback happens on a native thread, not inside a task object and set_value should be carefully handled.

Fig 1 shows a simple schematic of the runloop *for a single thread running on one core*. The runtime execution loop on each pika worker thread repeatedly executes tasks which are taken from one or more (usually thread local) queues, but between tasks, when one finishes or suspends, the runtime is free to perform bookkeeping work such as polling on requests before de-queuing the next task. Tasks in pika, which may be stackful or stackless, are not in general executed on the native OS thread as a simple function call, a lightweight context switch for stackful tasks is required and we must also store other state information. If the runtime polls for a request and then invokes set_value to execute the continuation associated with that request (i.e., the pipeline after the transform_mpi adaptor), it bypasses the task state management logic (possibly resulting in an unspecified execution context) which can lead to fatal errors. An example being when the continuation invokes a function that attempts to suspend itself in order to wait on another task (lightweight thread suspension relies on task state). Additionally, if the continuation is invoked directly, control leaves the scope of the runtime loop and no further request polling or task de-queuing will occur until control returns. We must therefore be very careful how and when set_value is called. Ideally, set_value should be followed by a task creation step that wraps the continuation function so that it is handled like any other task taken from the queue.

Fortunately, when a task is transferred from one scheduler to another in pika, the runtime takes the callable function and wraps it in a task object that is then submitted to the new scheduler. This means that using the std::execution API, if transform_mpi is followed by a continues_on adaptor, we can be sure that the continuation will be properly managed by the runtime as it is resubmitted to the queues (though this incurs some overhead). The question we seek to answer is how to balance this overhead with the need for responsive polling and task execution: if a thread pool has 16, 32 or even more threads, should all of them poll for requests concurrently, or should polling be limited to an individual thread, and does the cost of transferring work to/from a dedicated pool outweigh the gains made by reducing contention between threads.

To investigate this we have implemented a number of polling and scheduling strategies which we explain in the following section.

2.3 Polling modes and pools in pika

The first implementation of request handling in pika used what we refer to as YieldWhile mode, so named because a task would invoke the MPI callable, then poll on the request directly and yield if the request was not ready. The runloop suspends the yielding task, executes another task from the queue and when the yielded task returns to the front of the queue, it would poll again and retry—this process repeats until the request is ready.

```
invoke(mpi_callable, args..., &request);
while (!poll_request(request)) { yield(); }
set_value(MPI_SUCCESS); // or set_error(...)
```

This works unexpectedly well in many cases because the latency between a successful poll and the continuation being triggered is minimized. As soon as the fulfilled request is detected, the yield loop is exited, `set_value` is called and the continuation begins. In addition, since the polling is embedded inside the task itself (and not managed externally by the runloop), continuations remain in the task wrapper and there are no state-related problems. However, there are two drawbacks. Firstly, if the queue of tasks in the runloop is not empty, the time between polling checks might be long (increasing latency). Secondly, the runloop is being spammed by repeatedly suspending and resuming a task if it is not ready (increasing contention). When many tasks are actively performing communication on a thread pool with many threads, all repeatedly suspending and resuming tasks, this can increase overheads in the backend (particularly if work stealing between threads is allowed and high priority communication tasks are present).

There are a number of possible ways to try to improve things:

- allow the runloop to use `MPI_Testany` or `MPI_Testsome` on all outstanding requests together and trigger callbacks when ready (instead of individual tasks polling their own requests),
- create a custom thread pool only for MPI operations that does not execute regular computation tasks, enable MPI polling on this pool and disable polling on the regular pool,
- transfer all tasks executing `transform_mpi` to the dedicated MPI pool (increases latency on issuance of request, but reduces contention),
- allow the invocation of the initial MPI callable to happen directly by the task on the regular pool (inline, with no latency), but delegate the polling responsibility on the request to the dedicated MPI pool,
- boost the priority of a task once it has invoked an MPI function so that when suspended and subsequently resumed, it spends less time in the queue waiting to be retested, or boost the priority of any transfers/continuations triggered to minimize their time in queues (latencies)
- suspend tasks just once and wait on a condition variable until a polling test is successful, instead of repeatedly yielding and waking for retest,
- execute continuations as separate tasks (using `continues_on`) instead of using suspension,
- allow continuations to either run directly on the polling pool (inline) or transfer them back to the main worker thread pool to execute there.

2.3.1 *Mode flags.*

These options make for a large number of combinations of potential implementations that might or might not prove beneficial alone or in combinations. For example, transferring work from one thread to another generally increases latencies, whilst concentrating polling work on a dedicated thread reduces lock contention. Executing tasks inline on any thread (without transferring them) improves cache reuse and reduces latency, but increases lock contention between polling requests and dispatching MPI calls. To try to test as many options as possible, we have implemented `transform_mpi` with a set of MPI mode flags that can be represented in bitset form that enable us to switch in/out most of the above combinations. Fig 2 illustrates the major aspects of the control flow paths through `transform_mpi` that are enabled via the mode flags, which behave as follows:

- MPI pool: when enabled, a dedicated thread pool of one thread is used for MPI request handling. All polling is done on this thread—this reduces lock contention inside MPI. No regular (user work) tasks are executed on the MPI pool, so polling is not blocked by other work.
- Inline Requests: when enabled, the MPI callable that generates the request is invoked inline, i.e. the MPI function is called by the task that is executing `transform_mpi`, rather than being posted to the MPI pool. If many worker threads are executing communication tasks, this increases thread contention inside MPI, but reduces latency of the initial request issuance.
- Inline Completions: when enabled, a continuation is executed by whichever polling thread finds the request ready. If the dedicated MPI pool is enabled, this polling thread will be the one on the MPI pool and the continuation will be executed there. If no MPI pool is enabled, the continuation will be executed on the regular worker thread that polled the request.
- High Priority: when set, any MPI task created by a transfer to or resumed on a pool is boosted in priority to ensure that it is executed before lower priority tasks in the queue, thus reducing latency.

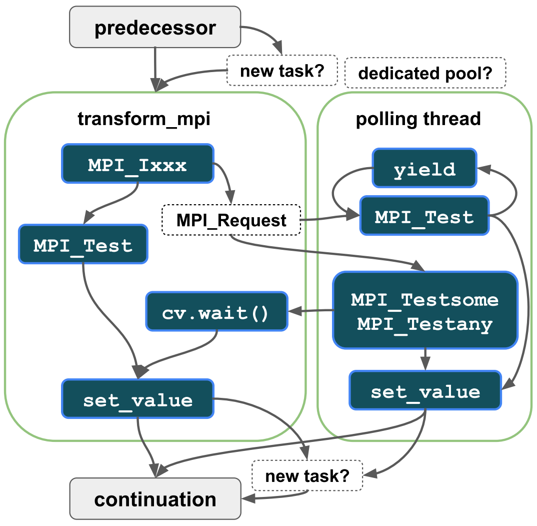

Figure 2: The control flow through `transform_mpi` can take one of several paths depending on whether it is triggered via an early termination test, or from inside a polling context and whether it wakes a thread, creates a new one, or executes a continuation directly.

Whilst the description of the possible execution paths through `transform_mpi` appears complex, the `std::execution` API makes it remarkably easy to construct a sender that manages the different combinations of request and completion handling. The following code snippet illustrates how we can create a sender that embeds the necessary logic for both request dispatching and completion handling by combining other senders with the internal `detail` namespace `transform_mpi` sender that does the internal work of managing MPI requests and completions.

```
// completion sender for users continuation
auto completion = [&](auto&& input)
  -> any_sender<> {
    any_sender<> s{detail::transform_mpi{input,
    continuation}};
    if (completions_inline) return s;
    else return std::move(s)
      | continues_on(default_scheduler(priority));
};
// transfer request to another pool if required
if (requests_inline)
  return completion(predecessor);
else
  return completion(predecessor | continues_on(
    mpi_scheduler(priority)));
```

Although much template magic is hidden in this greatly simplified snippet, the essence is that we can optionally transfer the invocation of the MPI function generating a request to another pool, and then optionally transfer the completion back again—whilst also including a priority flag. These 3 options give 8 combinations, but this doubles to 16 when we either enable or disable the MPI pool prior to this (`pika` supports the creation of thread pools during runtime initialization and the user may request at startup an MPI pool with polling enabled). In the code example, the *continuation* is the callable function that will execute when the MPI request completes and the *predecessor* is the function that passes input to the `transform_mpi` `sender`. Note that a request that is *not* transferred prior to invocation is referred to as *inline* and a continuation that is executed without transfer is also *inline*—it is assumed that inline invocations will generally be more efficient than a transferred one.

It is an unfortunate side-effect of the C++ type system that the two `Senders` constructed inside the lambda `sender` and `sender | continues_on` are considered by the compiler to be different types, even if they ultimately return the same value type when they complete (they represent different function objects when template instantiations have been unrolled). We must therefore wrap the return in an `any_sender` which is a special type-erased `sender` utility that can hold any other `sender` type. Likewise the final returned completion `sender` must also be an `any_sender`, the user does not need to be concerned by these implementation details.

In the event that an error occurs during an MPI operation, either during the invocation of the MPI function which may not return `MPI_SUCCESS`, or during the polling of the request when a bad status might be returned, the error is reported back to the user via the `set_error` mechanism. In our implementation, any error will be passed via `set_error` through the sender chain and ultimately result in an exception which the user can handle as required. Alternatively users may add their own `upon_error` handlers to the sender pipeline to catch such problems and deal with them appropriately. We have not implemented cancellation of requests at this time, but it may be added in the future.

2.3.2 *Handler methods.* In addition to the mode flags that control the behavior of the `transform_mpi` `sender`, we have also implemented different handler methods that determine how the polling and completion triggering mechanism is managed internally by the runtime since we wanted to improve on our original `YieldWhile` version. There are in fact five different modes that can be selected:

- **YieldWhile** (`YW`): As previously mentioned, `transform_mpi` polls and yields possibly repeatedly until ready. This mode has the drawback that it puts 'strain' on the task queues but has low latency between a request polling ready and the continuation being executed. Note that if another thread is polling and MPI internally marks our request as ready, we can do nothing about it until our next wake event when we poll again since this mode only polls on an individual request.
- **SuspendResume** (`SR`): To try to improving slightly on `YW` mode, `transform_mpi` invokes the MPI call, then suspends by waiting on a condition variable after passing a (callback, request) pair to the runtime polling loop which notifies the condition variable via the callback when the request completes. The suspended thread is woken and returns to the task queue where it will be picked up by the runloop. `SR` is a cleaner version of `YW` but it does not necessarily have the same low latency since waking the thread does not mean that it will immediately execute (though using a high priority flag helps push it to the front of the queue), it must still wait in the queue until a thread is free to serve it (which `YW` mode does not as it performs the polling directly). Ideally, the thread that has performed the polling operation and woken it, could execute it as its next task, but multiple requests may be ready at the same time and so it is not guaranteed that the same thread will execute the continuation.
- **Continuation** mode simply hands a callback to the polling loop, containing the `set_value/set_error` function and when the request is ready, the runloop calls the function, thus executing the continuation. This is potentially dangerous as we have now broken out of the runloop task wrapper (see Fig 1 and our earlier discussion) since a native thread (a thread not part of a task) is now executing user code. In most cases it turns out this is harmless as most continuations quickly trigger new tasks (and return control back to the run loop) or they are short operations that do not prevent the runloop from polling other requests for very long. There can however be cases where a continuation attempts to suspend whilst waiting for other work or perform some other operation that triggers an exception as the task state is not set up correctly.
- For this reason **NewTask** mode was added which *always* puts the continuation into a new task that is created inside the callback. The runloop can now never execute user code on an unwrapped thread (which does incur a cost) and the use of high priority mode is encouraged.
- The final **MPIx** mode is only supported by Open MPI with continuations extensions (as described in [18]) and it allows us to pass our request and a callback into MPI itself and poll on a master request handle that will call our callback during the `MPI_Test`

operation. We now have a `pika` polling thread calling into MPI, which in turn calls our user continuation and if we were to suspend that thread, things could go very wrong. We therefore expect that it is safest to not use inline continuations in this case and always use a `transfer` since it is an operation that has to create a new task object and put it into a queue on an execution context (even if we transfer back to the same thread pool, a new task will be created). It is worth noting that the `MPIx` mode dramatically simplifies our polling code since all management of requests is now handled by the MPI library itself. We only need to register each request and poll on a single object, our runloop polling code does not need to track any lists/queues or perform any locking operations since everything is handled internally by MPI.

It should be added that all modes have an early termination condition handled by `transform_mpi`: When an `MPI_Request` is returned from an MPI invocable function, an `MPI_Test` is performed on the request immediately since (for example) very small messages are internally buffered by MPI (typically when sending) and the poll will return success immediately. There is then no need to pass the request or callbacks into the runtime polling handler and the continuation is executed immediately without any further polling or waiting. This code path flows down the left side of Fig 2. Note also that in Fig 2 both `MPI_Testsome` and `MPI_Testany` are included—by default, `Testsome` is used for polling, but `Testany` may be used by setting a flag during startup. The effects are not significant and not discussed further here.

There are in total 5 methods and 16 possible combinations of mode flags. which gives a large number of possible ways of handling task execution and completion. Several modes are redundant and could potentially be simplified or combined as we shall see in the results section.

3 Benchmarking polling modes

In order to assess the performance of different modes with our `sender` implementation, we have run tests using the DLA-Future Eigensolver which uses 8 algorithms internally to give us an idea of overall application performance under a variety of communication patterns. We also created a simple benchmark called `mpi_ring_send` which sends messages in a ring around all ranks sequentially to stress test the continuations concept. In `mpi_ring_send` each rank (of N) starts a ring by sending a message to its neighbour which has posted a receive with a continuation attached that forwards the message on to the next rank and so on until it returns to its origin (one complete round of N messages). We set the number of rounds before termination of the ring to 5 and each rank starts a new ring running at the start of each iteration of a larger loop. Thus, on each iteration N new messages are sent and after $N \times 5$ iterations, messages start to be retired as they have completed 5 rounds. For our tests we allow 32 rings (per rank) to be in flight at any time, giving $32 \times N \times 5$ messages active per rank at any moment after the 32nd iteration. The total number of iterations is 5000 for small messages (64B) and 1000 for larger messages (128KB). These messages sizes may be changed, but are useful to test the eager/rendezvous modes that MPI uses. The total messages sent will be $5000 \times N^2 \times 5$ and $1000 \times N^2 \times 5$ respectively.

3.1 `mpi_ring_send` results

Before we look at actual results, we have certain *a priori* ideas about performance. For example, if we request a custom MPI pool consisting of a single thread, and we transfer all dispatches of MPI calls to this pool (adding latency, as task transfers to another pool are not free), then all MPI function calls and all MPI polling operations will happen on that same thread, and so we can initialize MPI in single threaded mode and this may perform better (lock free) than if we use inline requests on a default pool of (say) 16 threads with polling enabled on it—where 16 threads will be issuing MPI calls, and those same 16 threads will also be testing for completions as part of their polling loop. It is not possible unfortunately to have our cake and eat it, but we can at least experiment with transferring between pools or not, polling on all threads or not and seeing which works best.

The default implementation in our motivating application DLA-Future, used pool enabled, requests transferred, inline continuations and the yield while handler which meant that all messaging tasks were transferred to the MPI pool for dispatch, then polling on that pool triggered continuations inline. Executing continuations on the MPI pool ought to be a bad thing since it will block polling operations whilst the continuations are running. However, this worked reasonably well in practice because MPI did not have any contention from threads and the continuations were mostly short lived operations (such as submitting work to the GPU scheduler). Handling continuations on the MPI pool did cause some jitter, as polling would be interrupted when continuations ran. However, the fact that the MPI pool was executing work tasks as well as polling meant that it performed well on lower thread counts per rank. Our aim in this development was to establish whether this was the best strategy.

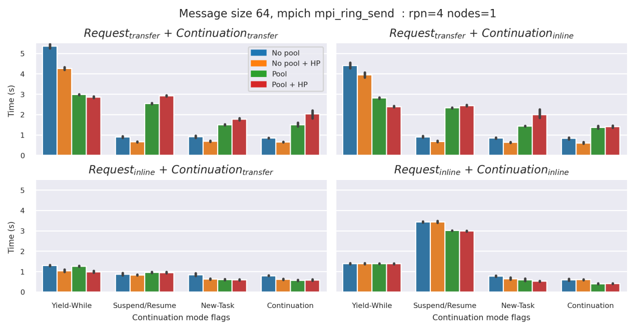

Figure 3: Grace CPU, ring-send, 64B, Cray MPICH, 4 ranks, 32 threads/rank, 1 node

Fig 3 shows timings for small messages of 64 bytes sent around a ring of just 4 ranks on a single node of the Grace-Hopper GH200 machine at CSCS (see *machine configuration* at the end of this section for node information). The graphs are broken into categories that show which flags were set. An inline request means that a thread initiates a message in place (no transfer) and an inline completion means that whichever thread polls the request will trigger the sender that fires the continuation (no transfer to another pool). Certain combinations are meaningless or redundant: if the MPI pool is disabled, then non inline requests are transferred back to the same pool they started on—which increases latency due to task creation, but also frees up the invoking thread, allowing it to immediately continue. Non inline completions likewise transfer back to themselves which also creates a new task and thus avoids a naked thread running code in certain handler modes. Some other modes are essentially equivalent, creating a new thread without a transfer, costs the same as an inline continuation invocation followed by a transfer (both amount to a thread creation and insertion into a queue). From Fig 3 we can see that the lower left quadrant (inline requests, transferred continuations) appears to be the most stable—which is a good sign. We prefer to issue requests directly on the thread that is generating them because it means low latency initiation of messages, transferring completions (when a pool is enabled) frees the polling thread and keeps work where we want it (though MPI must be multithread enabled when requests are inline).

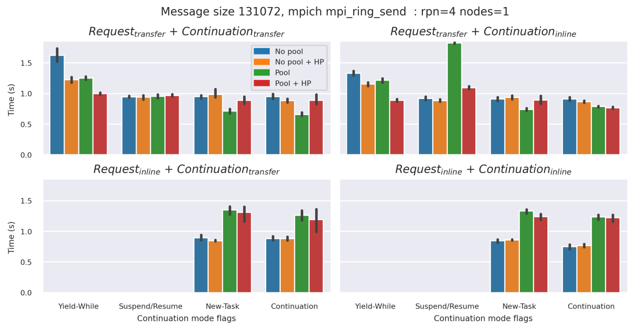

Figure 4: Grace CPU, ring-send, 128KB, Cray MPICH, 4 ranks, 32 threads/rank, 1 node

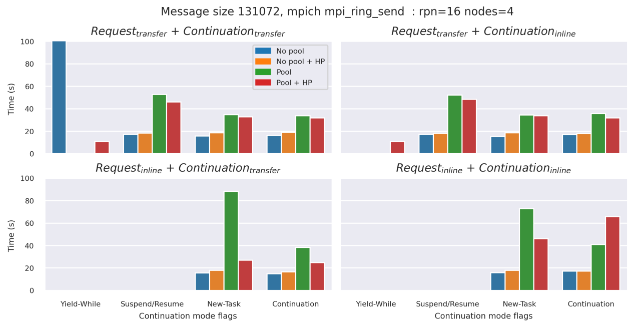

Figure 5: Grace CPU, ring-send, 128KB, Cray MPICH, 64 ranks, 16 threads/rank, 4 nodes (missing bars indicate tests that timed out)

Fig 4 shows the same test, except with larger messages. The lower left quadrant is now half empty, the `YW` and `SR` modes did not complete at all. The explanation for this is that `YW` and `SR` effectively block the issuing thread (task) until the request completes, 64B messages use the eager protocol and a send will complete immediately (data copied to buffer), but 128kB messages use the rendezvous protocol and MPI does not immediately complete. The `mpi_ring_send` is deliberately written to be pathological and issues sends/receives in a loop which becomes stuck if the initiating thread blocks. When inline requests are made, the iteration loop is halted and a deadlock occurs. These (inline request) modes must be considered "unsafe" for any application that might call any blocking function or other operation that might lead to deadlocks (via complex circular dependencies between threads or ranks for example).

Interestingly in Fig 4, we see that issuing request inline and triggering continuations inline in `continuation` mode and not bothering with a pool gives very good results, however this is also somewhat "unsafe" because now the bare polling thread is running user code—this mode should only ever be used if the user is aware of the implications and has made sure that all continuations are short lived and do not rely on task state. Future work will explore if this can be made "safe", by transferring to a custom *bypass* scheduler that takes the continuation, wraps it into a task and executes it inline without going via the task queues (an immediate context switch that models exactly how the runloop executes tasks when they are taken off queues).

Note also the similarities and difference between Fig 3, and Fig 4 and also Fig 5 which has even more ranks sending more data and stressing the network further: `YW` mode performs particularly poorly when using transferred requests as it causes the MPI pool to continuously yield tasks back to itself and cause queue contention. We can also see that the top row (transferred requests) are mostly slower when a pool is used for small messages which complete immediately—there is no advantage to transferring the messages to a single thread and having it do all the work, better to simply issue the MPI call and carry-on. Conversely we can see that transferred requests of large messages slightly benefit from using the MPI pool, since any waiting when issuing the call will delay subsequent messages, but when transferred, the calling task can inject messages more quickly and keep the network saturated. However, the bottom row (inline requests) are slower with a pool for large messages—this can be explained by observing that our aim is to saturate the network with messages, but with inline large message requests, each thread is slowed slightly as they wait for MPI to process them and there is only one thread doing the polling and releasing the injecting threads, so we do not reach the same level of performance. The same pattern appears in Fig 5 where we have even more ranks participating. When requests are inline, but there is no pool, all worker threads are processing requests and the test performs better. This tells us that when network performance dominates the code, it may be generally better to forgo the use of a custom pool and allow more threads to help out. Note that in Fig 5 we see performance worsen as continuations are done inline on the MPI pool and progress is even further impeded by placing extra work on the polling thread.

We note also that bumping the priority of tasks does not always help—since the benchmark is synthetic and task ordering is not particularly crucial (every message is triggering new/more messages) it is not surprising that priority does not universally improve performance. We do surprisingly, see one `YW` on Fig 5 with a pool and transferred requests performing well which deserves further investigation (other missing bars were caused by timeouts when testing).

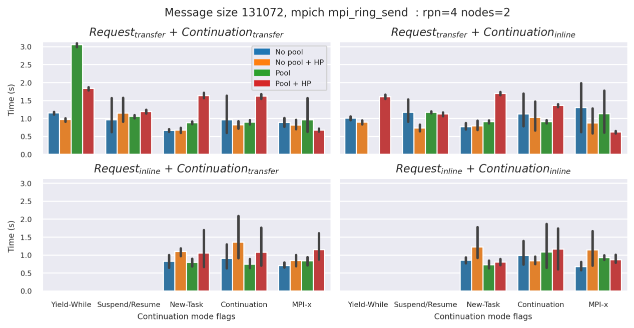

Figure 6: Grace CPU, ring-send, 128kB, Open MPI, 4 ranks, 32 threads/rank, 1 node

Fig 6 shows results using the Open MPI library (which we note is not yet well supported on Cray Slingshot machines). The results are much noisier than those from MPICH, but what is interesting, is that `MPIx` mode using extensions for continuation callbacks gives results comparable to our own polling methods. This tells us that the request callback management present in the Open MPI library works well enough that if it were generally available in the MPI distributions we could make use of it and simplify our own implementation significantly. A complicated polling loop managing queues and locks could be reduced to a small fraction of its existing size; the reduction in code complexity and maintenance burden would be substantial though the onus would be on the MPI maintainers to ensure the implementation remains efficient.

Our synthetic benchmark is however, a somewhat contrived application example, since no real computational work is being done on the cores between polling checks—blocking a worker thread by polling has no real cost, we wish to know how the modes perform in a real application and whether the trends here hold.

3.2 DLA-Future results

The distributed generalized eigensolver in DLA-Future has been implemented with backends for both GPU and CPU, though some parts of the GPU implementation are still reliant on CPU-based computation. It represents a much better real-world test of our MPI modes as it contains a mixture of code that will not all be affected identically by changes in the communication layer. There are 8 algorithms included in the eigensolver which can be categorized into 4 principal groups according to the communication and computation patterns:

- Point to point communication with next neighbors, computation performed on CPU only, communication cannot overlap with computation and therefore low latency is the key to good performance. This category includes Band to Tridiagonal (band2trid).
- Column and row-wise collective communication (broadcasts and reduce), CPU computation is non negligible. This category includes Reduction to Band (red2band).
- Row-wise collective communication (broadcasts), column-wise point to point communication, CPU computation is non negligible. Communication can overlap with computation. This group includes Back-transformation Band to Tridiagonal (bt_band2trid).
- Column and row-wise collective communication (broadcasts and reduce), CPU computation is limited or absent. In this category fall all the algorithms not included in the other groups (trsm, bt_red2band, trid_evp, hegst, cholesky)

The categories are ordered according to the expected severity of the impact of changing the communication methods. For the CPU eigensolver, the categories become effectively just 2, since no computation is offloaded to the GPU anymore. Only the distinction between the latency bound algorithm (band2trid) and the other algorithms remains.

Fig 7 shows the timing of the algorithms in the GPU eigensolver as a function of the completion mode used. The differences across all modes are less pronounced than in the synthetic benchmark but there are some clear trends. Using a thread pool, gives an advantage since the CPUs are busy with real work and isolating polling on a dedicated thread helps. `YW` and `SR` modes are slightly worse than the task-creating modes and unusable as expected when requests are inline. We can see that different algorithms are affected more by the modes than others, and this is a reflection of the categories listed above. Band to Tridiagonal being the most affected, with significant differences between the best and worst modes—implying that choosing the right mode can make a real difference to time to solution. Reduction to Band suffers when no MPI pool is present and the other modes have less obvious minor changes across modes. For the CPU variant of the algorithms the differences smooth out as time to solution for the same problem increases due to the absence of the GPUs. However the impact of a dedicated pool and/or high priority execution is clearer, as CPU threads are busy with high load computation and less resources are available to schedule communication tasks.

Fig 9 shows a detailed view of the effects of the MPI modes on the Band to Tridiagonal algorithm which is confined to run on the CPU and this along with Fig 8 showing the timings of the CPU-only version of the eigensolver confirms a similar trend. The CPU results are slightly worse for the `NewTask` completion mode, but `Continuation` mode with a transfer works well and is safe to use. Note that the timings shown in Figs 7,8 show the algorithms run independently; the eigensolver itself completes in a shorter time than the sum total of the algorithms as task and matrix/tile dependencies in DLA-Future allow the algorithms to overlap.

These results indicate that for the eigensolver, transferring continuations to the worker pool guarantees that polling is not blocked, and distributes the work more evenly—In summary, our recommended default is to: Use an MPI pool for polling, issue requests inline and transfer continuations back with high priority.

It is interesting to note that our preferred modes for the eigensolver are not always the best for the `mpi_ring_send` benchmark, which tells us that the way we handle messages is sensitive to the application’s traffic and workload and that it is worth trying several modes during development. We have added support to `transform_mpi` to allow different MPI modes to be used in different places on a per function basis, this may improve future performance if we find that particular algorithms benefit more from one mode whereas others require a different mode and assessing this is part of our future work.

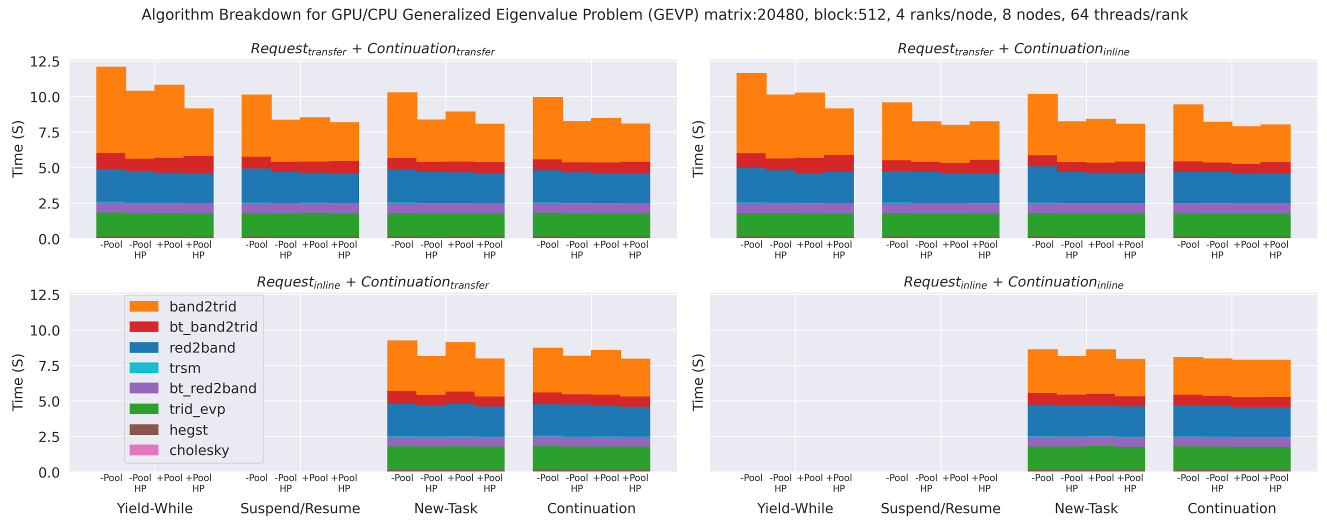

Figure 7: Plot showing the breakdown in timing for different completion modes of each of the 8 sub-algorithms implemented within the DLA-Future Generalized Eigensolver running on 'Daint' GH200 GPU nodes. Gaps in the plot represent modes that did not complete and should be avoided. The timing variance shows how the workload of each algorithm is affected by the completion mode.

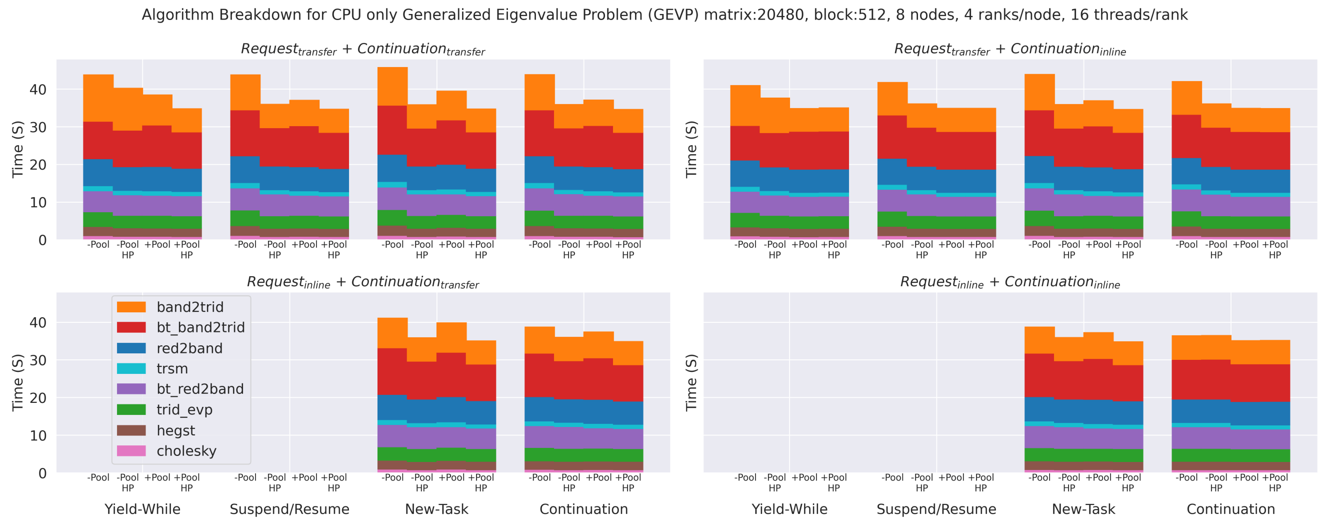

Figure 8: Plot showing the breakdown in timing for different completion modes of each of the 8 sub-algorithms implemented within the DLA-Future Generalized Eigensolver running on 'Eiger' CPU only nodes. The same completion modes fail to complete as in the GPU version, but the variation in timing between algorithms is more pronounced with the CPU version

Machine configuration. GPU Eigensolver results and MPI Ring benchmarks were collected on CSCS's ALPS 'Daint' nodes, each containing 4x Grace-Hopper GH200 superchips (72-core, Arm V9.0 aarch64 (Neoverse V2), 120GB LPDDR5 RAM; H100 GPU with 94GB HBM3) and Cray Slingshot-11 interconnect using cray-mpich 8.1.32.

CPU Eigensolver results were collected on CSCS's ALPS 'Eiger' nodes, each containing 2x AMD Epyc 7742 (64-core), 256GB DDR4 RAM and Cray Slingshot-11 interconnect using cray-mpich 8.1.32.

4 Conclusions

We have improved the handling of MPI messaging in DLA-Future by introducing a more flexible and efficient polling strategy based upon the `std::execution` Senders API. The most striking feature of our implementation, is how easy it is for us to compose new polling strategies by combining `Senders` and switch between them. If we find that the polling method used in one application, or on one particular machine or architecture, does not work as well as another, we can easily benchmark and change the default mode, and we can

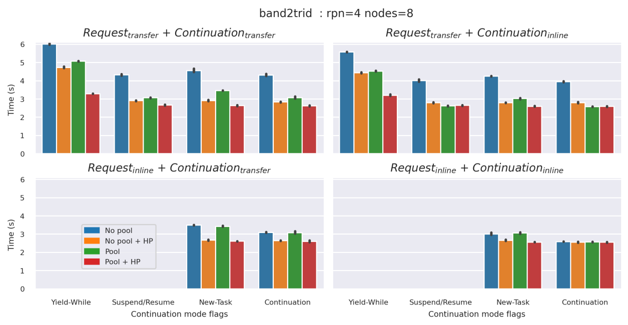

Figure 9: Timing of the Band to Tridiagonal portion of the GPU Eigensolver using different MPI modes

add new modes should we come up with other better strategies. The simplicity and composability of the API and the ability to pass work between thread pools (as well as to/from accelerator devices) with a simple syntax gives programmers a powerful and expressive tool that will have a significant impact on software design as the API becomes more widely adopted.

The APIs used in this implementation are consistent with the C++26 `std::execution` proposal (at the time of writing), and with the exception of runtime specific hooks to insert the polling code (which would need to be implemented afresh in every runtime) the work is portable to any other tasking environment that implements the new standard.

MPI itself partially owes its success to a stable API over the years, and now that C++ has one of its own designed specifically for asynchronous programming, there should be a concerted effort to develop a `sender` based API for MPI so that all C++ HPC applications can work with continuation based messaging without needing to reinvent the wheel. Our experience with the Open MPI callback extensions shows us that not only would each runtime no longer need to reimplement their own polling and completion handling, but a layer of abstraction would be removed from applications (or libraries) and by exposing senders instead of requests, the MPI implementation would probably become simpler and more efficient too.

References

[1] A. W. Appel and T. Jim. 1989. Continuation-passing, closure-passing style. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Austin, Texas, United States). ACM, New York, NY, USA, 293–302.

[2] John Biddiscombe, Thomas Heller, Anton Bikineev, and Hartmut Kaiser. 2017. Zero Copy Serialization using RMA in the Distributed Task-Based HPX runtime. In *14th International Conference on Applied Computing*. IADIS, International Association for Development of the Information Society.

[3] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemariner, and Jack Dongarra. 2011. DAGuE: A Generic Distributed DAG Engine for High Performance Computing, In Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops). *Proceedings of the Workshops of the 25th IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2011 Workshops)*, 1151–1158.

[4] Message Passing Interface Forum. 2021. MPI: A Message-Passing Interface Standard, Version 4.0. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf. , 1139 pages.

[5] Demian Hespe, Lukas Hübner, Florian Kurpicz, Peter Sanders, Matthias Schimek, Daniel Seemaier, Christoph Stelz, and Tim Niklas Uhl. 2024. KaMPIng: Flexible and (Near) Zero-overhead C++ Bindings for MPI. arXiv:2404.05610 [cs.DC] https://arxiv.org/abs/2404.05610

[6] Torsten Hoefler and Andrew Lumsdaine. 2008. Message progression in parallel computing - to thread or not to thread?. In *2008 IEEE International Conference on Cluster Computing*. 213–222. https://doi.org/10.1109/CLUSTR.2008.4663774

[7] ISO/IEC JTC1/SC22/WG21 ISO/IEC 14882 Programming Languages C++. 2024. P2300R10, std::execution. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2300r10.html.

[8] ISO/IEC JTC1/SC22/WG21 ISO/IEC 14882 Programming Languages C++. 2024. P3149R11, async-scope. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3149r11.html.

[9] Stephen Jones. 2018. Cuda: new features and beyond. 2018. *URL: http://ondemand.gputechconf.com/gtc/2018/presentation/s8278-cuda-newfeatures-and-beyond.pdf* (2018).

[10] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, et al. 2020. HPX - The C++ Standard Library for Parallelism and Concurrency. *Journal of Open Source Software* 5, 53 (2020), 2352. https://doi.org/10.21105/joss.02352

[11] Laxmikant V. Kale. [n. d.]. The Charm++ Parallel Programming System. https://doi.org/10.5281/zenodo.3370873

[12] Jonathan Lifflander, Phil Miller, Nicole Lemaster Slattengren, Nicolas Morales, Paul Stickney, and Philippe P. Pébaÿ. 2020. Design and Implementation Techniques for an MPI-Oriented AMT Runtime. In *2020 Workshop on Exascale MPI (ExaMPI)*. 31–40. https://doi.org/10.1109/ExaMPI52011.2020.00009

[13] Omri Mor, George Bosilca, and Marc Snir. 2023. Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine. In *Proceedings of the 52nd International Conference on Parallel Processing* (Salt Lake City, UT, USA) *(ICPP '23)*. Association for Computing Machinery, New York, NY, USA, 153–162. https://doi.org/10.1145/3605573.3605642

[14] Joachim Protze, Marc-André Hermanns, Matthias Müller, Van Nguyen, Julien Jaeger, Emmanuelle Saillard, Patrick Carribault, and Denis Barthou. 2021. MPI detach—Towards automatic asynchronous local completion. *Parallel Comput.* 109 (10 2021), 102859. https://doi.org/10.1016/j.parco.2021.102859

[15] Kevin Sala, Jorge Bellón, Pau Farré, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Daniel Holmes, Vicenç Beltran, and Jesus Labarta. 2018. Improving the Interoperability between MPI and Task-Based Programming Models. In *Proceedings of the 25th European MPI Users' Group Meeting* (Barcelona, Spain) *(EuroMPI '18)*. Association for Computing Machinery, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/3236367.3236382

[16] Kevin Sala, Xavier Teruel, Josep M. Perez, Antonio J. Peña, Vicenç Beltran, and Jesus Labarta. 2019. Integrating blocking and non-blocking MPI primitives with task-based programming models. *Parallel Comput.* 85 (2019), 153–166. https://doi.org/10.1016/j.parco.2018.12.008

[17] Derek Schafer, Thomas Hines, Evan Drake Suggs, Martin Rüfenacht, and Anthony Skjellum. 2021. Overlapping Communication and Computation with ExaMPI's Strong Progress and Modern C++ Design. In *2021 Workshop on Exascale MPI (ExaMPI)*. 18–26. https://doi.org/10.1109/ExaMPI54564.2021.00008

[18] Joseph Schuchart, Philipp Samfass, Christoph Niethammer, José Gracia, and George Bosilca. 2021. Callback-based completion notification using MPI Continuations. *Parallel Comput.* 106 (2021), 102793. https://doi.org/10.1016/j.parco.2021.102793

[19] Mikael Simberg, Auriane R., John Biddiscombe, Hartmut Kaiser, Akhil Nair, Bhumit Attarde, Severin Strobl, Hannes Vogt, Dimitra Karatza, Rocco Meli, yuri@FreeBSD, Ângelo Andrade Cirino, Sergey Fedorov, and Srinivas Yadav. 2024. pika-org/pika: pika 0.26.1. https://doi.org/10.5281/zenodo.13141745

[20] Raffaele Solcà, Mikael Simberg, Rocco Meli, Alberto Invernizzi, Auriane Reverdell, and John Biddiscombe. 2024. DLA-Future: A Task-Based Linear Algebra Library Which Provides a GPU-Enabled Distributed Eigensolver. In *Asynchronous Many-Task Systems and Applications*, Patrick Diehl, Joseph Schuchart, Pedro Valero-Lara, and George Bosilca (Eds.). Springer Nature Switzerland, Cham, 135–141.

[21] Raffaele Solcà, Alberto Invernizzi, Auriane Reverdell, John Biddiscombe, Mikael Simberg, Rocco Meli, Guglielmo Gagliardi, Andreas Fink, Teodor Nikolov, Harmen Stoppels, Lara Querciagrossa, Alo Roosing, and Alessandro Colombo. 2025. DLA-Future 0.10.0. https://doi.org/10.5281/zenodo.15426276

[22] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. 1992. Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News* 20, 2 (apr 1992), 256–266. https://doi.org/10.1145/146628.140382

[23] Kyle B. Wheeler, Richard C. Murphy, and Douglas Thain. 2008. Qthreads: An API for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. https://doi.org/10.1109/IPDPS.2008.4536359

[24] Jiakun Yan, Hartmut Kaiser, and Marc Snir. 2023. Design and Analysis of the Network Software Stack of an Asynchronous Many-task System – The LCI parcelport of HPX. In *Proceedings of the SC '23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis* (Denver, CO, USA) *(SC-W '23)*. Association for Computing Machinery, New York, NY, USA, 1151–1161. https://doi.org/10.1145/3624062.3624598