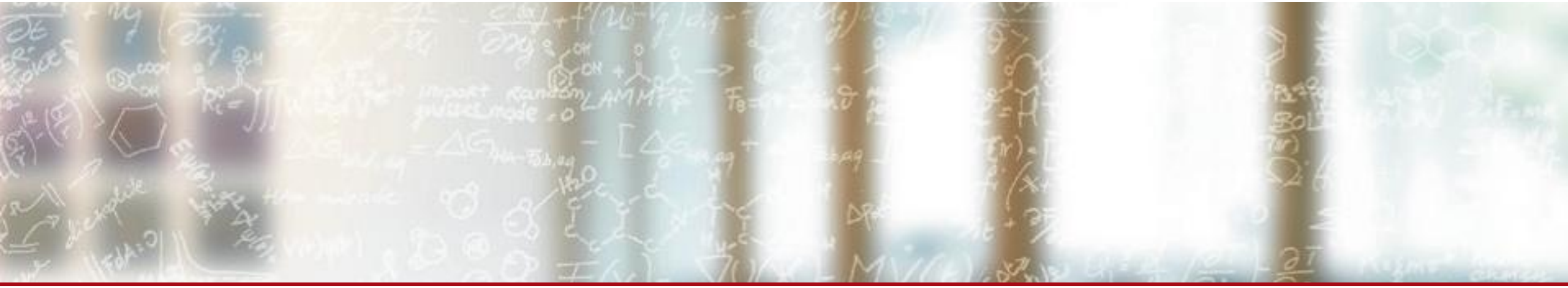
CSCS
Centro Svizzero di Calcolo Scientifico
Swiss National Supercomputing Centre

ETH zürich



MPI and std::execution (the C++ 26 async API)

09/10/2025

John Biddiscombe, Raffaele Solcà, Alberto Invernizzi, Auriane Reverdell, Rocco Meli, Mikael Simberg, Joseph Schuchart

Preview of talk at SC25 in the ExaMPI workshop next month

Will try to add a bit more on MPI library integration …

On the Integration of Lightweight Tasks with MPI using the C++26 std::execution 'Senders' API

John Biddiscombe
Mikael Simberg
Auriane Reverdell
john.biddiscombe@cscs.ch
mikael.simberg@cscs.ch
auriane.reverdell@cscs.ch
ETH Zürich, Swiss National Supercomputing Centre, Switzerland

Raffaele Solcà
Alberto Invernizzi
Rocco Meli
raffaele.solca@cscs.ch
alberto.invernizzi@cscs.ch
rocco.meli@cscs.ch
ETH Zürich, Swiss National Supercomputing Centre, Switzerland

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

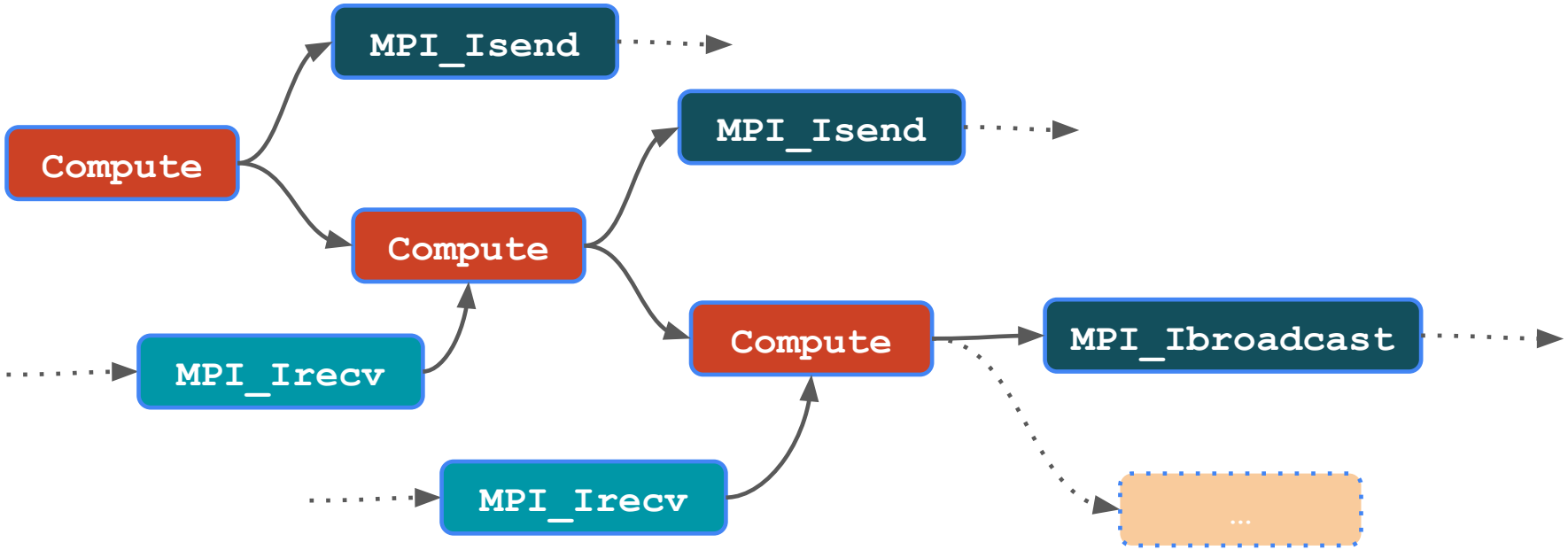
Motivation/Background

DLA-Future: task-based distributed Eigensolver with `HPX` using `futures`

We want MPI as nodes in the graph (like everyone else)

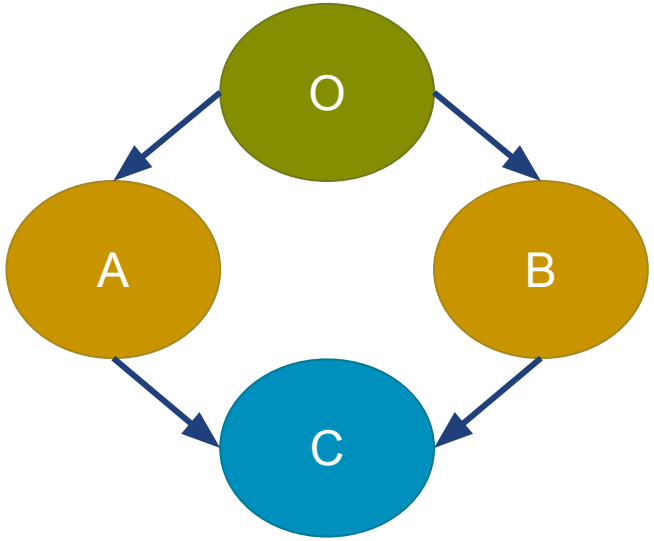
- C++26 introduced `std::execution -> Senders`

- CSCS forked `pika` from `HPX` : targeting `stdexec` instead of `futures`
  - Single node runtime
  - MPI sender support for traditional distributed applications
    - Not active messages as in distributed HPX runtime



Senders provide an (API level) abstraction for tasks

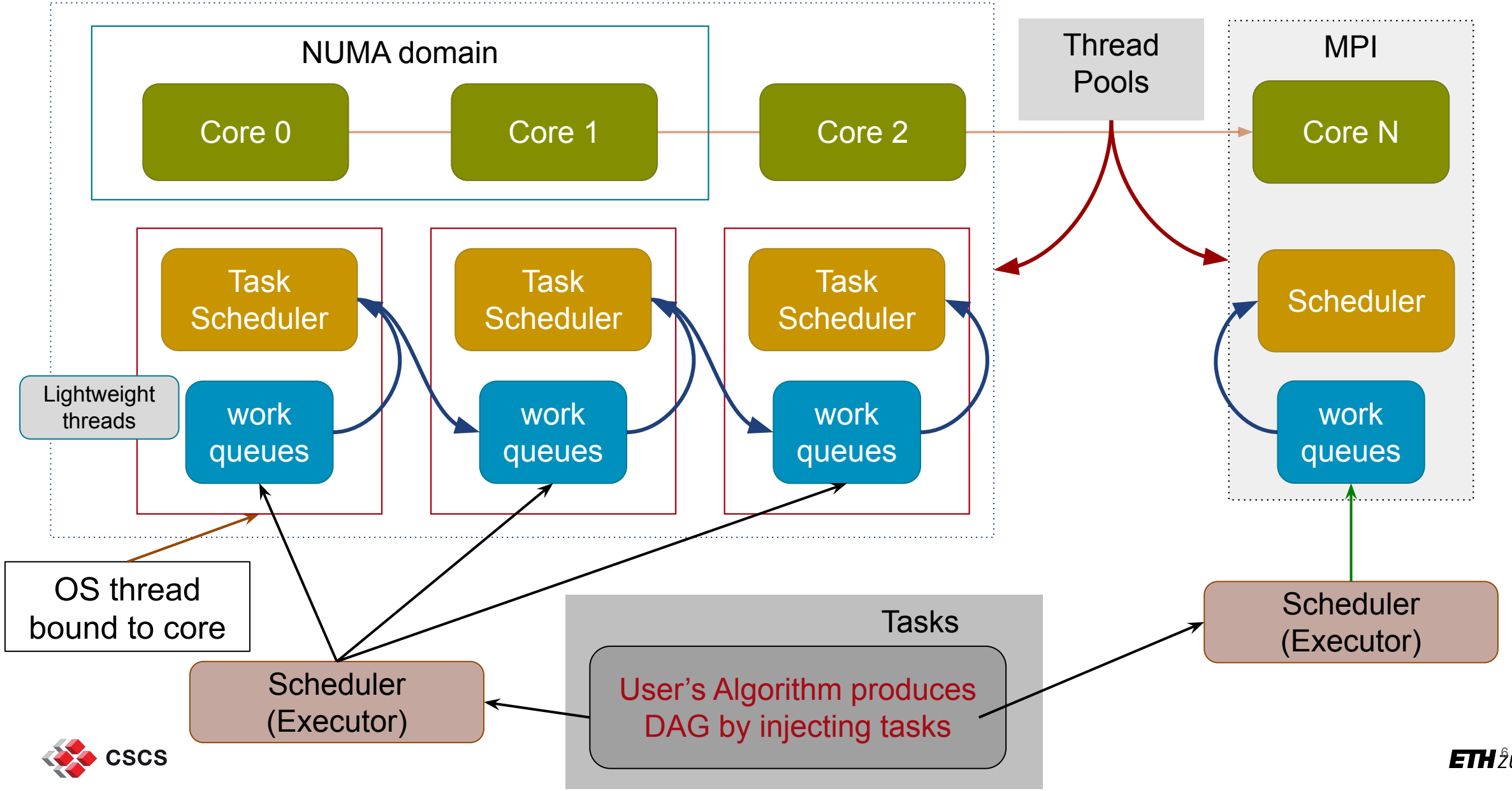
- Continuation Passing Style of computation (CPS or dataflow)
  - Task graph describing work
  - When (A and B) : then C



- Encourages Functional approach to programming
  - Pass parameters
  - Return values (which are passed on)
  - Discourage/avoid global state/vars/other
    - Races/Locks/Scalability

- Similar in concept to cuda graphs
  - Much of std::execution driven by NVidia (to support this) (P2300R10) + friends

pika architecture: Where should the polling be done?



Senders API in summary

- API for schedulers/senders/customizations to adhere to
  - use of templates makes cross library porting easier (or harder …)
- Syntax to specify scheduler to run on (execution context)
  - starts_on(...), schedule(...), just(...)
- Explicit syntax to transfer to a new scheduler (execution context)
  - continues_on(...)
- Syntax to build the DAG
  - then(...), when_all(...), bulk(...)
- Automatic unwrapping of returned params
- Ability to cancel and handle errors in a standard way

- Explicit launch of sender
- No synchronization at creation time
  - futures had an (often useless) overhead (creation / launching / attaching)

Wrapping MPI in senders

```
// is func(Ts..., MPI_Request) invocable
template <typename F, typename... Ts>
inline constexpr bool is_mpi_request_invocable_v =
    std::is_invocable_v<F, std::add_lvalue_reference_t<std::decay_t<Ts>>..., MPI_Request*>;
```

Any function (inc. lambda) that has a last parameter of type `MPI_Request`

Can be given to `transform_mpi` and returns a sender (that can be chained)

`transform_mpi` keeps arguments alive until request completes

end user never sees `MPI_Request`

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

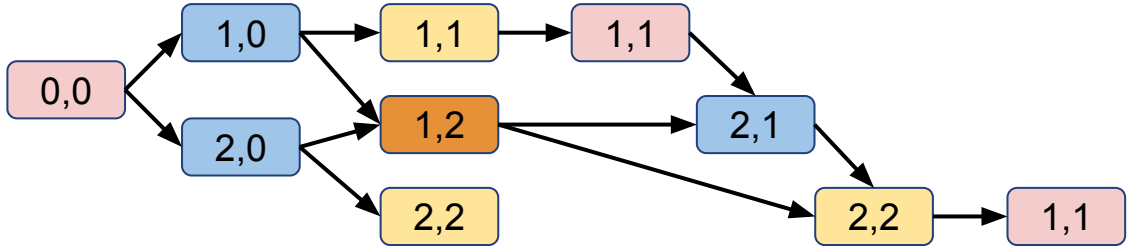
in DLA buf is a sender too

Magic takes place here

Off topic side note: Tiles wrapped via senders too

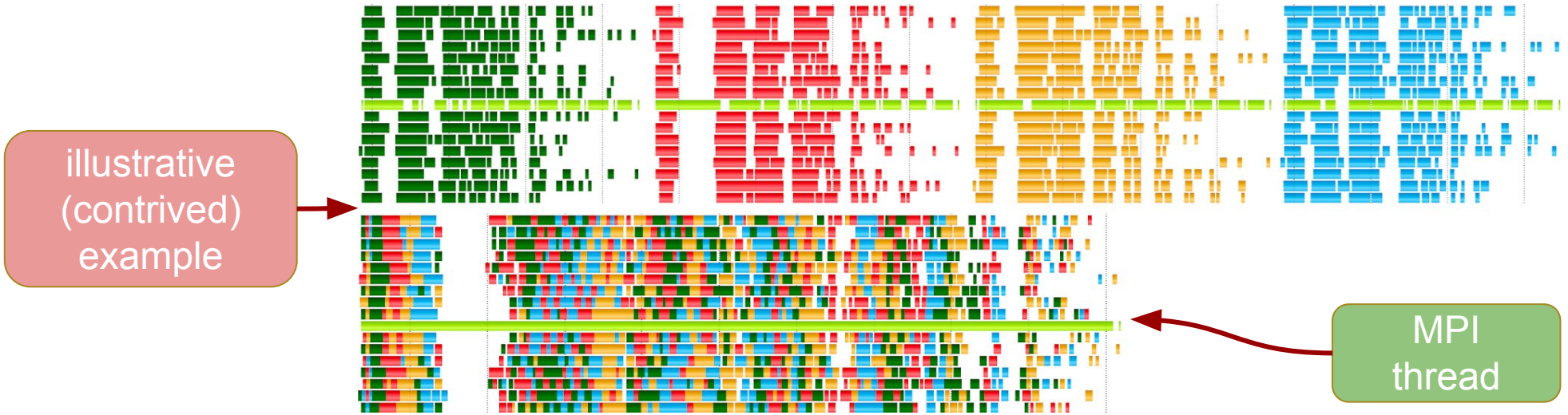
Each tile is a sender - access is controlled via R/W semantics

- Multiple simultaneous reads
- Exclusive writes



Algorithms do not have to wait

- Tiles become available independently from algorithm



side note: Args kept alive

Internally the sender moves all the argos into the operation state:

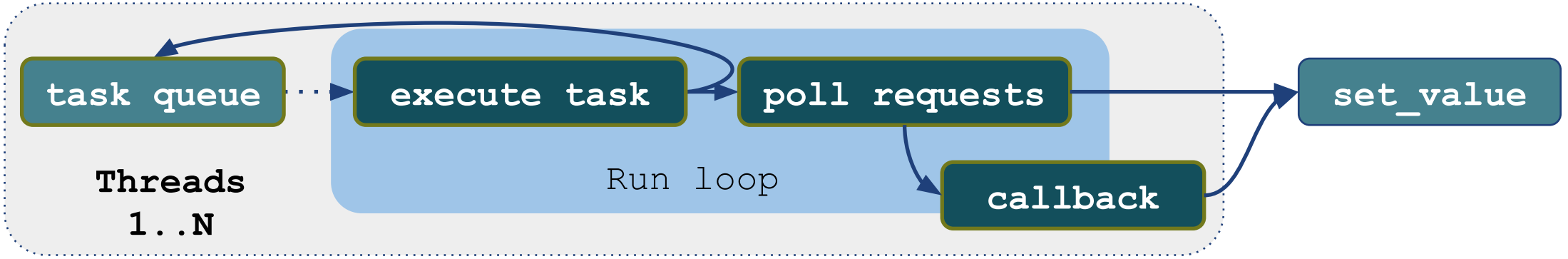
Stored in a tuple using perfect forwarding

(refs are copied by ref, moved argos are kept alive, etc etc)

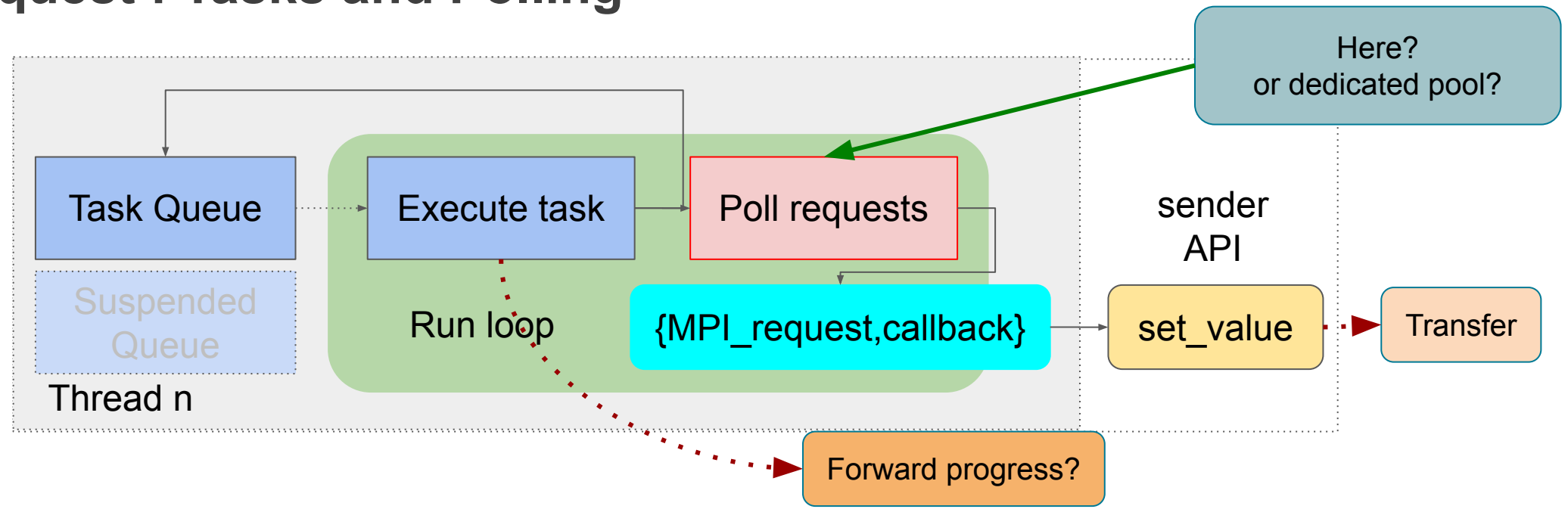
```
// move all the arguments we received into the operation state
using ts_element_type = std::tuple<std::decay_t<Ts>...>;
r.op_state.ts.template emplace<ts_element_type>(std::forward<Ts>(ts)...);
```

Where the tuple type matches the args we saw earlier

```
// is func(Ts..., MPI_Request) invocable
template <typename F, typename... Ts>
inline constexpr bool is_mpi_request_invocable_v =
    std::is_invocable_v<F, std::add_lvalue_reference_t<std::decay_t<Ts>>..., MPI_Request*>;
```


task queue
execute task
poll requests
set_value
Run loop
callback
Threads
1..N

MPI_Request : Tasks and Polling



- You can only poll between tasks
  - Tasks “block” progress of MPI (in a sense)
- Every thread can poll
- … or maybe just one thread should do it (c.f. MPI progress ‘auto’)
  - But even if you have low latency polling …
  - you still have to schedule the actual work
  - `set_value` passes control to (executes) next sender directly (danger!)

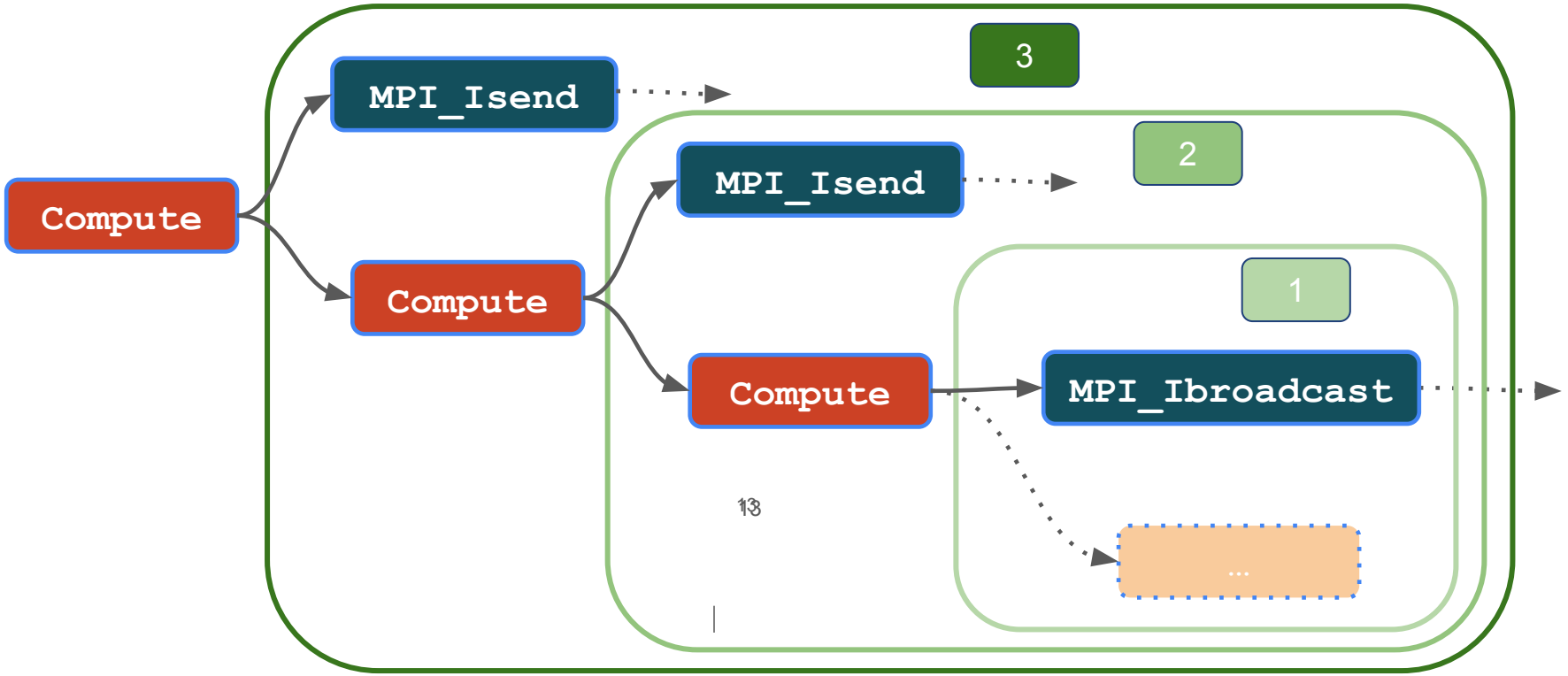
The continuation/callback

Continuation-ception - Each level of callback encapsulates the lower levels

Stack usage needs to be controlled

Template instantiations can become odious (debug symbols 😞)

```
mpi_transform(...) | then([]{...}) | mpi_transform(...) | then([]{...})
```



Triggering continuations - Yield-While

- Original implementation

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

  - Task performs a send/recv/other
  - Tests, yields if not ready (another task is executed)
  - Repeats until ready, then continues on next sender

Pros : no new tasks, low latency (task already active)

Cons: push/pop overhead on queues, multithread polling. MPI pool?

Forward progress if someone locks?

Triggering continuations - Suspend/Resume

- Suspend once - get woken once

```
std::unique_lock l{r.op_state.mutex};
mpi::detail::add_suspend_resume_request_callback(r.op_state); // wakes us up
r.op_state.cond_var.wait(l, [&]() {
    return r.op_state.completed;
});
// call set_value/set_error depending on mpi return status
mpi::detail::set_value_error_helper(r.op_state.status, std::move(r.op_state.r));
```

  - Task performs a send/recv/other
  - Task suspends (if quick early poll fails)
  - gets woken when request is ready

Pros : no new tasks, less push/pop/test on this thread

Cons: when woken, must join queue (High/Low Priority). MPI pool?

Forward progress?

Triggering continuations - New Task

- Attach continuation to new task

```
if (status != MPI_SUCCESS) {
    ex::set_error(std::move(op_state.r),
} else {
    execution::thread_priority p = use_priority_boost(op_state.mode_flags) ?
        thread_priority::boost : thread_priority::normal;
    auto snd0 = ex::schedule(default_pool_scheduler(p)) | ex::then([&op_state]() mutable {
        ex::set_value(std::move(op_state.r));
    });
    ex::start_detached(std::move(snd0));
}
```

Pros : Can never block. Honours pool placement

Cons: Obsolete transfers. Creation of new task = $$$ cost.

Triggering continuations - Continuation

- Execute continuation directly

```
detail::add_request_callback([&op_state](int status) mutable {
    if (mpi_status == MPI_SUCCESS) {
        ex::set_value(std::forward<Receiver>(receiver), std::forward<Ts>(ts)...);
    } else {
        ex::set_error(std::forward<Receiver>(receiver),
    }}, op_state.request);
```

Pros : quick and simple, only queuing - safe *when combined with transfer.*

Cons: might block if not transferred. MPI pool execution possible.

Triggering continuations - OpenMPI MPI_EXT_CONTINUE

- OpenMPI has an extension for attaching callbacks

  Joseph Schuchart et al “Callback-based completion notification using MPI Continuations.”

```
// mpix callback calls set_value/error directly
mpi::detail::MPIX_Continue_cb_function* func =
    &mpi::detail::mpix_callback_continuation<operation_state>;
mpi::detail::register_mpix_continuation(
    &r.op_state.request, func, &r.op_state);
```

Pros : Dramatically simplifies polling logic. Safe *when combined with transfer.*

Cons: Clunky “C” API with idiosyncrasies, we want senders please.

Request pipeline

Request inline

Issue the MPI_Ixxx call directly from the parent task

Request transfer

pass the control of the request to MPI pool (if present), issue there

Completions inline

Handle the completion directly when the poll says ready

Completions transfer

Pass the completion to the default pool and handle there

Many combinations are in reality obsolete

Pool transfers (mpi_transform internals)

```
        // get mpi completion mode settings
        bool completions_inline = use_inline_completion(mode);
        bool requests_inline = use_inline_request(mode);
        execution::thread_priority p = use_priority_boost(mode) ?
            execution::thread_priority::boost :
            execution::thread_priority::normal;
        auto f_completion = [&](auto&& sender) mutable -> unique_any_sender<> {
            unique_any_sender<> s{
                transform_mpi_detail::sender<std::decay_t<decltype(sender)>, std::decay_t<F>>{
                    std::forward<decltype(sender)>(sender), std::forward<F>(f), mode}};
3.          if (completions_inline) { return s; }
4.          else { return std::move(s) | continues_on(default_pool_scheduler(p)); }
        };
1.      if (requests_inline) { return f_completion(std::forward<Sender>(sender)); }
        else
        {
2.          return f_completion(
                std::forward<Sender>(sender) | continues_on(mpi_pool_scheduler(p)));
        }
```

Operation state

```
struct operation_state
{
    // these 3 should be passed into constructor
    PIKA_NO_UNIQUE_ADDRESS Receiver r;
    PIKA_NO_UNIQUE_ADDRESS F f;
    // the status of the MPI operation
    int volatile status{MPI_SUCCESS};
    // internal storage for the request object
    MPI_Request request{MPI_REQUEST_NULL};
    // these vars are  only needed by suspend/resume mode
    bool completed{false};
    pika::detail::spinlock mutex;
    pika::condition_variable cond_var;
  ...
```

In a native implementation, the request object and operation state would probably be combined somewhere

Callback triggering from the polling side

```
case handler_method::new_task:
{
    // The callback will call set_value/set_error inside a new task and execution will continue on that thread
    add_new_task_request_callback (r.op_state);
    break;
}
case handler_method::continuation:
{
    // The callback will call set_value/set_error execution will continue on the callback thread
    add_continuation_request_callback (r.op_state);
    break;
}
case handler_method::mpix_continuation:
{
    MPIX_Continue_cb_function * func = &mpix_callback_continuation<operation_state>;
    register_mpix_continuation (&r.op_state.request, func, &r.op_state);
    break;
}
```

We can add other calllback triggering modes easily if we come up with new ones (The YieldWhile and SuspendResume are not shown for space+tedious reasons)

Ring Send benchmark

MPI_Ring_Send test/benchmark

- I Iterations
  - On each iteration
    - Every rank starts a round

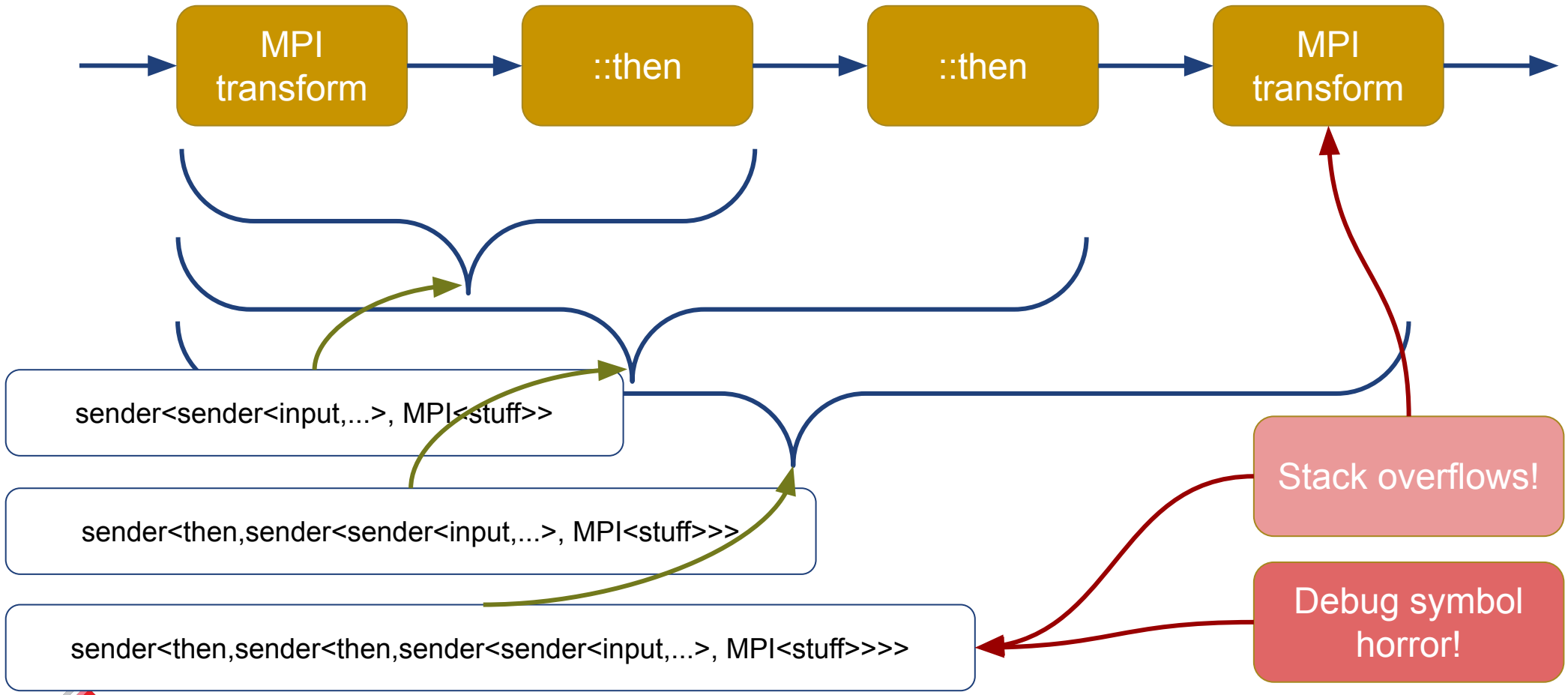
- R Rounds
  - Each round
  - Message travels through N-1 ranks
    - and then back to itself

Total messages Sent = N * (N*R) * I

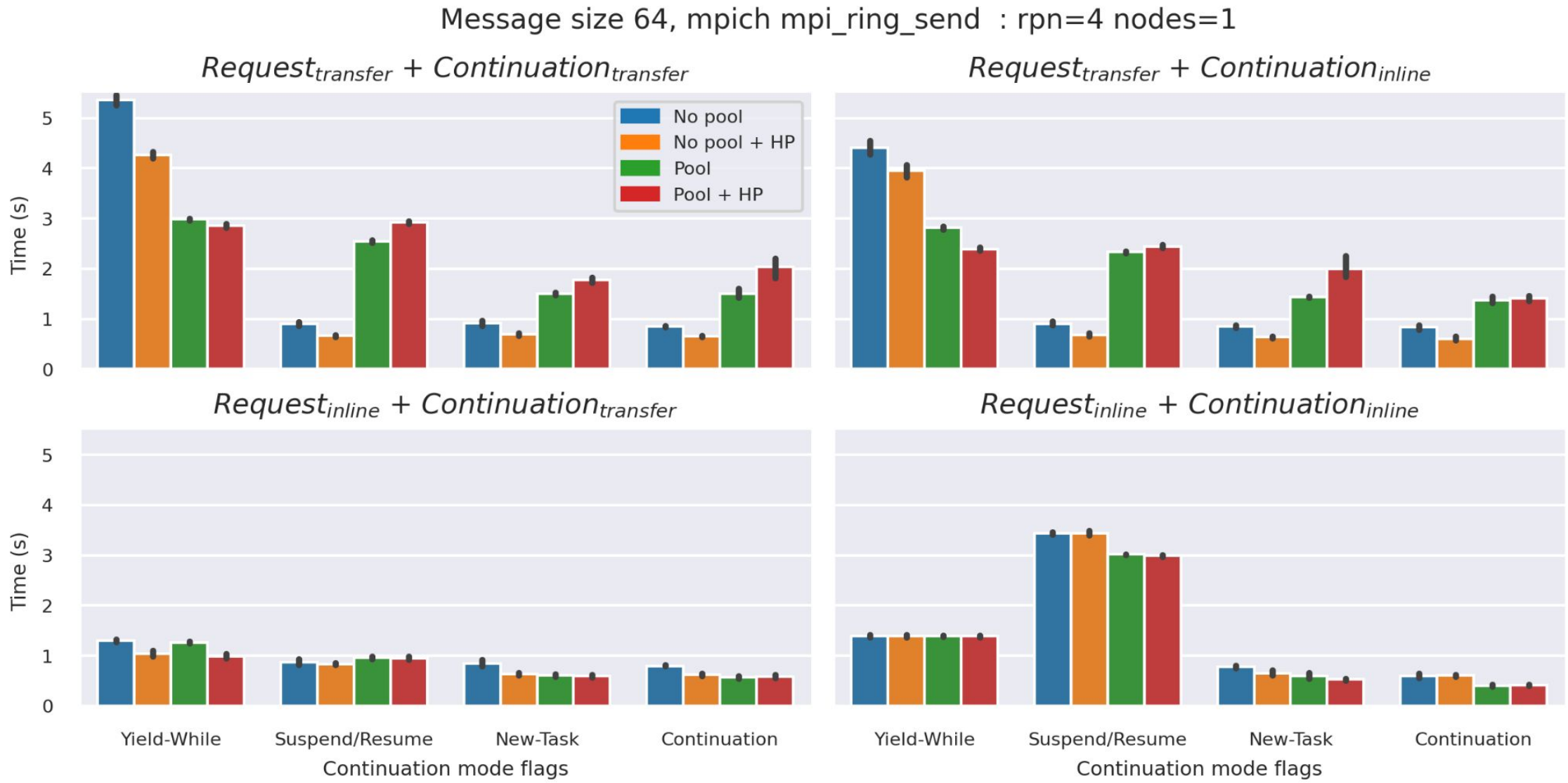
Must limit number of iterations in flight to prevent overloading queues

Senders : cons

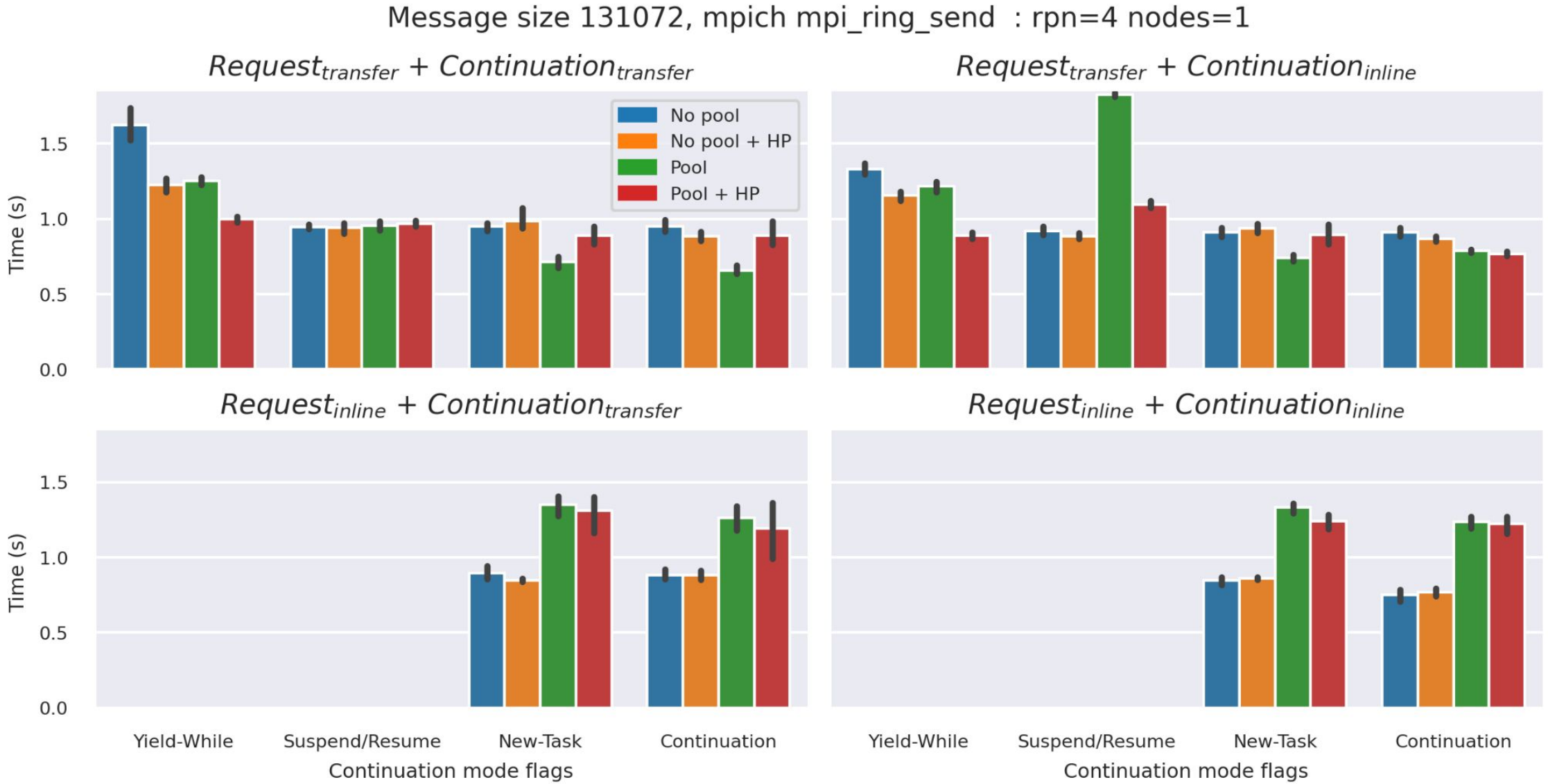
The problem/explosion of types, (ring send with 25 rounds …)



mpi_ring_send 64 byte messages



mpi_ring_send 128k messages



Why do we lose data on inline requests?

If the MPI operation blocks/suspends for any reason, the loop gets stuck

Not safe

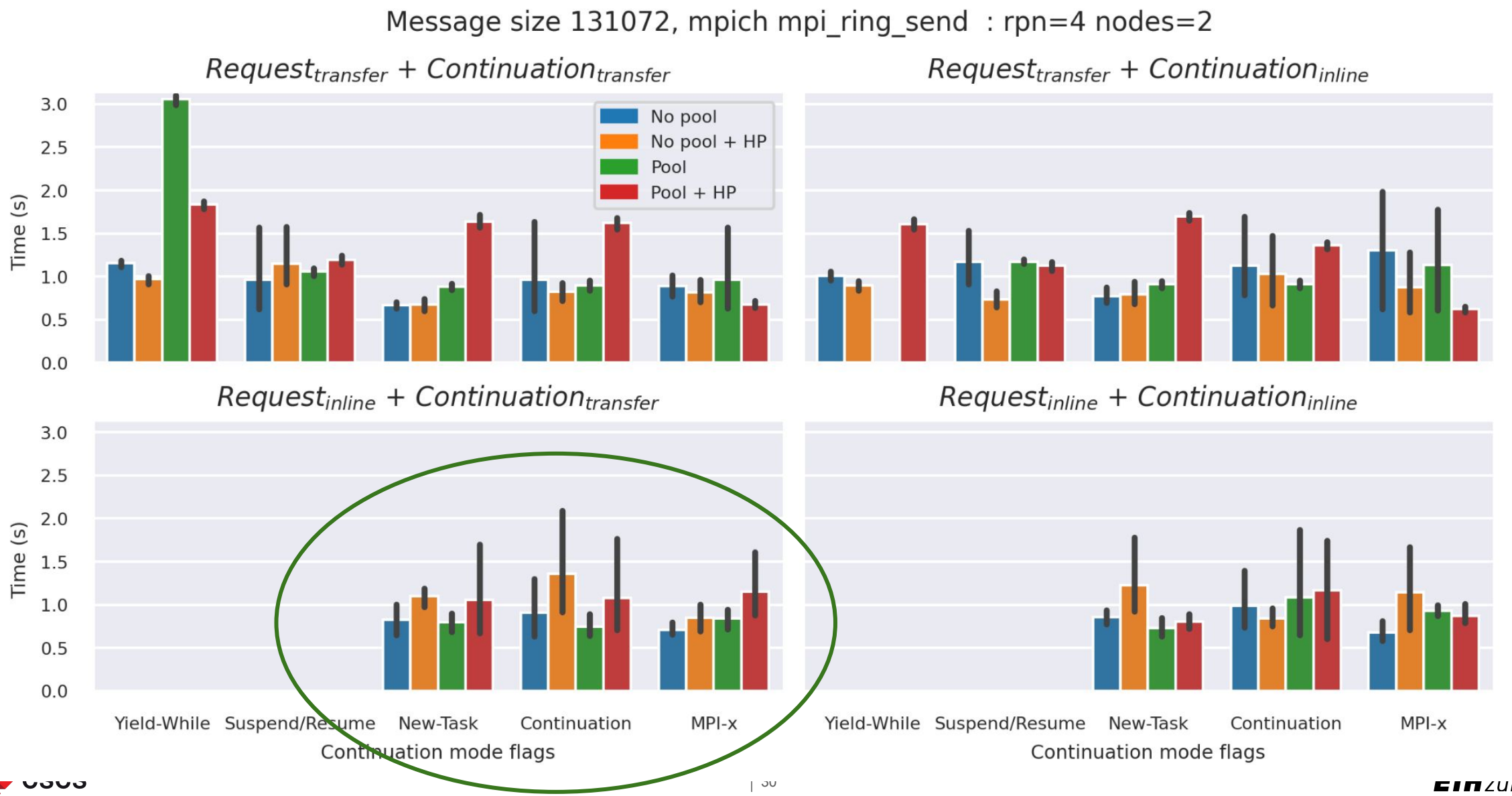
```
Loop
  Perform a send/recv.other
EndLoop
```

Safe

```
Loop
  transfer(Perform a send/recv.other)
EndLoop
```

mpi_ring_send 128k 64-ranks

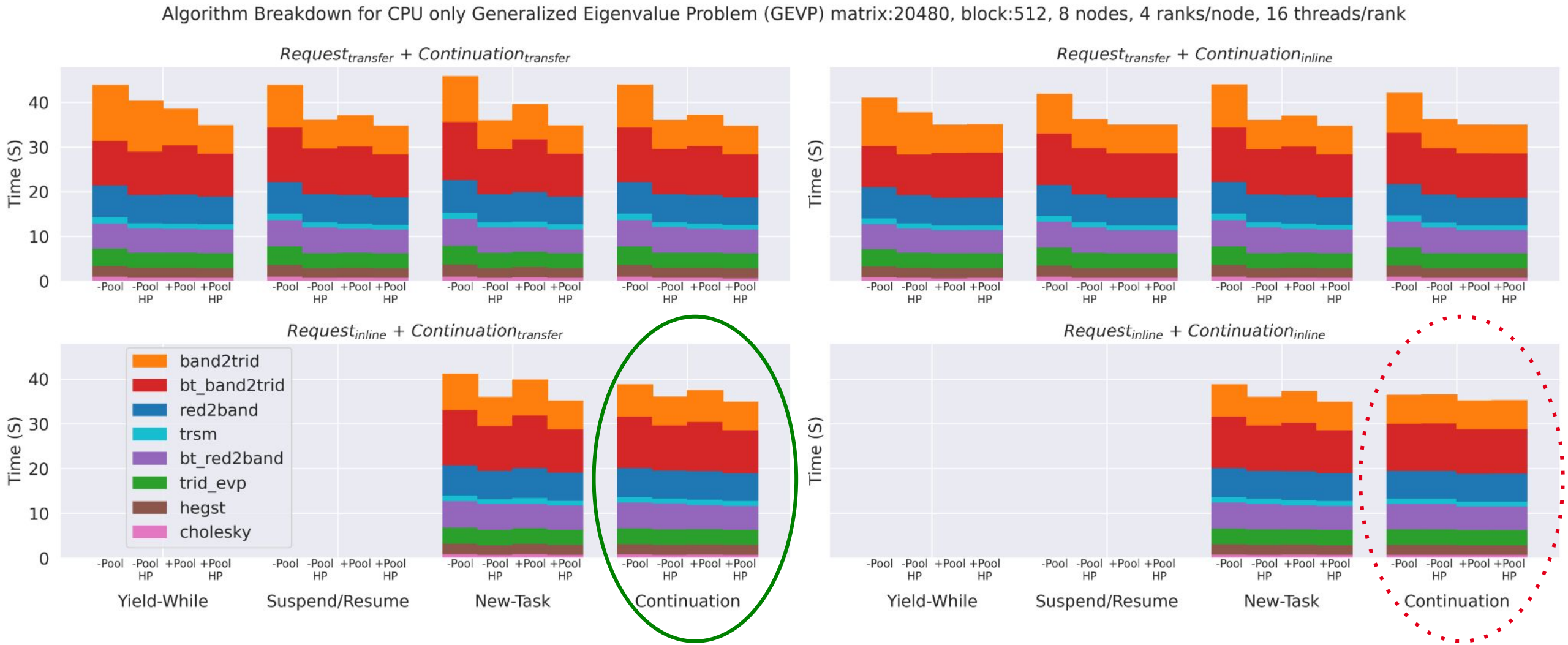
OpenMPI MPI_EXT_CONTINUE



Eigensolver

GPU + CPU Eigensolver

CPU Eigensolver



Best strategy/mode

Depends …

DLA mostly runs GPU calculations (on gh200) after completion handled

- no real penalty for burning MPI thread (manycores anyway)
- Continuations inline ok(ish) - often short

Pool use generally better

- guarantees (at least one) thread transfer (inline continuations can block)
- Optimization : (`MPI_Init_thread`(single), when transferred)

Ring_Send test benefits from not using a pool.

Heavy use of long sender chains : short latency needed - don’t block polling thread

Native MPI

Final remarks: Native MPI

No need for user to ever touch an MPI_Request

Put all the polling into MPI_PROGRESS_AUTO

operation state stack allocated

Tell the user to always use

continues_on

(or require a pool handle up front - since everything is templated, this is doable)

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

Greatly simplifies the process of integrating MPI into task graphs

API is implementation independent,

pika/HPX/Kokkos/QThreads/ParSEC would all share the same interface