Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations

Joseph Schuchart, Christoph Niethammer, José Gracia
HLRS, Stuttgart, Germany

EuroMPI'20
MPI and the Emergence of Asynchronous Programming Models

Asynchronous Programming Models: C++ `std::async`, OpenMP tasks, TBB...

- Dispatching work to a `scheduler` for eventual execution
- Constraints on order of execution (dependencies, data-flow, ...)

```c
#pragma omp task depend (in: sendbuf)
{
    MPI_Send (sendbuf, myrank, ...);
}
#pragma omp task depend (out: recvbuf)
{
    MPI_Recv (recvbuf, myrank, ...);
}
```
MPI and the Emergence of Asynchronous Programming Models

Asynchronous Programming Models: C++ `std::async`, OpenMP tasks, TBB . . .

▶ Dispatching work to a `scheduler` for eventual execution
▶ `Constraints` on order of execution (dependencies, data-flow, . . .)

▶ MPI ≈ dependencies not exposed to the scheduler
▶ Coordinating interaction with MPI is tedious
▶ Test-yield cycles are inefficient, at best

```c
#pragma omp task depend(in: sendbuf)
{
    MPI_Send(sendbuf, myrank, ...);
}
```

```c
#pragma omp task depend(out: recvbuf)
{
    MPI_Recv(recvbuf, myrank, ...);
}
```
MPI and the Emergence of Asynchronous Programming Models

Asynchronous Programming Models: C++ std::async, OpenMP tasks, TBB . . .

- Dispatching work to a scheduler for eventual execution
- Constraints on order of execution (dependencies, data-flow, . . .)

- MPI \approx dependencies not exposed to the scheduler
- Coordinating interaction with MPI is tedious
- Test-yield cycles are inefficient, at best
- Previously proposed: integration of higher-level concurrency abstractions with the MPI layer

```cpp
#pragma omp task depend(in: sendbuf)
{
    MPI_Send(sendbuf, myrank, ...);
}
#pragma omp task depend(out: recvbuf)
{
    MPI_Recv(recvbuf, myrank, ...);
}
```
MPI and the Emergence of Asynchronous Programming Models

Asynchronous Programming Models: C++ std::async, OpenMP tasks, TBB ...

- Dispatching work to a scheduler for eventual execution
- Constraints on order of execution (dependencies, data-flow, ...)

- MPI \(\approx\) dependencies not exposed to the scheduler
- Coordinating interaction with MPI is tedious
- Test-yield cycles are inefficient, at best
- Previously proposed: integration of higher-level concurrency abstractions with the MPI layer

```
#pragma omp task depend(in: sendbuf)
{
    MPI_Send(sendbuf, myrank, ...);
}
```
```
#pragma omp task depend(out: recvbuf)
{
    MPI_Recv(recvbuf, myrank, ...);
}
```
MPI and the Emergence of Asynchronous Programming Models

Asynchronous Programming Models: C++ std::async, OpenMP tasks, TBB ...

- Dispatching work to a scheduler for eventual execution
- Constraints on order of execution (dependencies, data-flow, ...)

- MPI ≈ dependencies not exposed to the scheduler
- Coordinating interaction with MPI is tedious
- Test-yield cycles are inefficient, at best
- Previously proposed: integration of higher-level concurrency abstractions with the MPI layer

```cpp
#pragma omp task depend(in: sendbuf)
{
    MPI_Send(sendbuf, myrank, ...);
}
#pragma omp task depend(out: recvbuf)
{
    MPI_Recv(recvbuf, myrank, ...);
}
```
MPI and the Emergence of Asynchronous Programming Models

**Asynchronous Programming Models:** C++ `std::async`, OpenMP tasks, TBB...

- Dispatching work to a *scheduler* for eventual execution
- *Constraints* on order of execution (dependencies, data-flow, ...)

- MPI $\approx$ dependencies not exposed to the scheduler
- Coordinating interaction with MPI is tedious
- Test-yield cycles are inefficient, at best
- **Previously proposed:** integration of higher-level concurrency abstractions with the MPI layer

```
#pragma omp task depend(in: sendbuf)
{
    MPI_Send(sendbuf, myrank, ...);
}
#pragma omp task depend(out: recvbuf)
{
    MPI_Recv(recvbuf, myrank, ...);
}
```

---

Not portable!
Threads in the MPI Standard

The two main requirements for a thread-compliant implementation:

1. All MPI calls are **thread-safe**.
2. Blocking MPI calls will **block the calling thread only**, allowing another thread to execute, if available.

MPI 3.1, §12.4.1
Threads in the MPI Standard

The two main requirements for a thread-compliant implementation:

1. All MPI calls are thread-safe.
2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available.

MPI 3.1, §12.4.1

Correct MPI Implementations:

- Prevent internal data structure corruption
- Release mutexes/locks before blocking
- Pthreads integration always thread-compliant
Threads in the MPI Standard

The two main requirements for a thread-compliant implementation:

1. All MPI calls are **thread-safe**.
2. Blocking MPI calls will **block the calling thread only**, allowing another thread to execute, if available.

*MPI 3.1, §12.4.1*

**Correct MPI Implementations:**
- Prevent internal data structure corruption
- Release mutexes/locks before blocking
- *P*threads integration always thread-compliant

**Portable Applications:**
- Do not rely on implementation details
- Ensure all communication started eventually
- Coordinate MPI ↔ scheduler interaction
Threads in the MPI Standard

The two main requirements for a thread-compliant implementation:

1. All MPI calls are thread-safe.
2. Blocking MPI calls will block the calling thread only, allowing another thread to execute, if available.

MPI 3.1, §12.4.1

Correct MPI Implementations:
- Prevent internal data structure corruption
- Release mutexes/locks before blocking
- Pthreads integration always thread-compliant

Portable Applications:
- Do not rely on implementation details
- Ensure all communication started eventually
- Coordinate MPI ↔ scheduler interaction

What constitutes a thread?
A decade-old problem...

We're actually in a thread/process terminology crisis in Linux. Various people have various ideas about what we should mean by “thread,” “process,” “task,” and “thread group.”

https://lwn.net/Articles/81790/, 2004
A decade-old problem...

We’re actually in a thread/process terminology crisis in Linux. Various people have various ideas about what we should mean by “thread,” “process,” “task,” and “thread group.”

https://lwn.net/Articles/81790/, 2004

16 years later, the HPC community is in a similar situation...
Taxonomy used in this work (excerpt)

**Kernel Thread**  Thread in kernel space  
(I/O, signal handling, Light-Weight Process (LWP))

**User Thread**  Lowest system-level concurrency abstraction in user space, mapped 1:1 or N:M to LWP, scheduled preemptively (aka. *a thread*)

**Fiber**  User-space execution context (stack/registers), scheduled cooperatively onto user threads

**Task**  Package of work, execution state in fiber or thread
Taxonomy used in this work (excerpt)

**Kernel Thread**  Thread in kernel space
(I/O, signal handling, Light-Weight Process (LWP))

**User Thread**  Lowest system-level concurrency abstraction in user space, mapped 1:1 or N:M to LWP, scheduled preemptively (aka. a thread)

**Fiber**  User-space execution context (stack/registers), scheduled cooperatively onto user threads

**Task**  Package of work, execution state in fiber or thread

Observation:
- User-Level Threads (ULT) are fibers (like boost.fiber, MS Fibers, ...)
- Underlying Shepherds, Execution Streams, ... are threads

Recommended read:
Gor Nishanov: *Fibers under the magnifying glass*, 2018.
Fiber Integration (Should Be) Considered Harmful\textsuperscript{1}

\textsuperscript{1}Dijkstra is said to have penned his famous letter after a talk on a continuation-like concept in Algol60.
Let’s separate concerns

- MPI should manage **communication concerns** (requests)
- Application layer should manage **task concerns**

**How to loosely couple different concerns?**
Let’s separate concerns

- MPI should manage **communication concerns** (requests)
- Application layer should manage **task concerns**

**How to loosely couple different concerns?**

**Continuations**

Example: `std::future::then`
Let’s separate concerns

- MPI should manage **communication concerns** (requests)
- Application layer should manage **task concerns**

How to loosely couple different concerns?

**Continuations**

Example: `std::future::then`
MPI Continuations Interface

- Introduce 3 new functions:
  - MPIX_Continue_init: initialize a continuation request
  - MPIX_Continue: attach a continuation to single operation
  - MPIX_Continueall: attach a continuation to a set of operations (executed once all are complete)
MPI Continuations Interface

- Introduce 3 new functions:
  - `MPIX_Continue_init`: initialize a continuation request
  - `MPIX_Continue`: attach a continuation to single operation
  - `MPIX_Continueall`: attach a continuation to a set of operations (executed once all are complete)

```c
/* Initialize a continuation request */
int MPIX_Continue_init (MPI_Request *cont_req);

/* Callback function signature */
void (MPIX_Continue_cb_function)(
  void * user_data,
  MPI_Status *statuses);

/* Attach a continuation to a single operation */
int MPIX_Continue(
  MPI_Request *request,
  int *flag,
  MPIX_Continue_cb_function *cont_cb,
  void *cb_data,
  MPI_Status *status,
  MPI_Request cont_req);

/* Set up continuation to be executed once all operation have completed */
int MPIX_Continueall(
  int count,
  MPI_Request requests[],
  int *flag,
  MPIX_Continue_cb_function *cont_cb,
  void *cb_data,
  MPI_Status statuses[],
  MPI_Request cont_req);
```
MPI Continuations Interface

Introduce 3 new functions:

- **MPIX_CONTINUE_init**: initialize a continuation request
- **MPIX_CONTINUE**: attach a continuation to single operation
- **MPIX_CONTINUEall**: attach a continuation to a set of operations (executed once all are complete)

```c
/** Initialize a continuation request */
int MPIX_CONTINUE_init(MPI_Request *cont_req);

/* Callback function signature */
void (MPIX_CONTINUE_cb_function)(
void *user_data,
MPI_Status *statuses);

/** Attach a continuation to a single operation */
int MPIX_CONTINUE(
MPI_Request request,
int *flag,
MPIX_CONTINUE_cb_function *cont_cb,
void *cb_data,
MPI_Status *status,
MPI_Request cont_req);

/** Set up continuation to be executed once all * operation have completed */
int MPIX_CONTINUEall(
int count,
MPI_Request requests[],
int *flag,
MPIX_CONTINUE_cb_function *cont_cb,
void *cb_data[],
MPI_Status statuses[],
MPI_Request cont_req);
```
MPI Continuations Interface

- Introduce 3 new functions:
  - MPIX_Continue_init: initialize a continuation request
  - MPIX_Continue: attach a continuation to single operation
  - MPIX_Continueall: attach a continuation to a set of operations
    (executed once all are complete)

- Continuation requests accumulate and track active continuations
  - Progress and check for completion
  - May itself have a continuation attached

```c
/** Initialize a continuation request */
int MPIX_Continue_init (MPI_Request * cont_req);

/* Callback function signature */
void (MPIX_Continue_cb_funtion) (
    void * user_data,
    MPI_Status * statuses);

/** Attach a continuation to a single operation */
int MPIX_Continue(
    MPI_Request * request,
    int * flag,
    MPIX_Continue_cb_funtion * cont_cb,
    void * cb_data,
    MPI_Status * status,
    MPI_Request cont_req);

/** Set up continuation to be executed once all operation have completed */
int MPIX_Continueall(
    int count,
    MPI_Request requests[],
    int * flag,
    MPIX_Continue_cb_funtion * cont_cb,
    void * cb_data,
    MPI_Status statuses[],
    MPI_Request cont_req);
```
MPI Continuations Interface

- Introduce 3 new functions:
  - **MPIX_Continue_init**: initialize a continuation request
  - **MPIX_Continue**: attach a continuation to single operation
  - **MPIX_Continueall**: attach a continuation to a set of operations (executed once all are complete)

- Continuation requests **accumulate and track** active continuations
  - Progress and check for completion
  - May itself have a continuation attached

- One continuation per non-persistent operation, multiple for persistent operations

```c
/* Initialize a continuation request */
int MPIX_Continue_init ( MPI_Request * cont_req );

/* Callback function signature */
void ( MPIX_Continue_cb_funtion ) ( void * user_data, MPI_Status * statuses );

/* Attach a continuation to a single operation */
int MPIX_Continue ( MPI_Request * request, int *flag, MPIX_Continue_cb_funtion * cont_cb, void * cb_data, MPI_Status *status, MPI_Request cont_req );

/* Set up continuation to be executed once all operation have completed */
int MPIX_Continueall ( int count, MPI_Request requests[], int *flag, MPIX_Continue_cb_funtion * cont_cb, void * cb_data, MPI_Status statuses[], MPI_Request cont_req );
```
MPI Continuations Interface

Introduce 3 new functions:
- MPIX_Continue_init: initialize a continuation request
- MPIX_Continue: attach a continuation to single operation
- MPIX_Continueall: attach a continuation to a set of operations (executed once all are complete)

Continuation requests accumulate and track active continuations
- Progress and check for completion
- May itself have a continuation attached

One continuation per non-persistent operation, multiple for persistent operations

See paper for details on continuation requests, restrictions, status handling, rationales, . . .
MPI Continuations Interface

- **Introduce 3 new functions:**
  - `MPIX_Continue_init`: initialize a *continuation request*
  - `MPIX_Continue`: attach a continuation to single operation
  - `MPIX_Continueall`: attach a continuation to a set of operations (executed once all are complete)

- **Continuation requests accumulate and track active continuations**
  - Progress and check for completion
  - May itself have a continuation attached

- **One continuation per non-persistent operation, multiple for persistent operations**

- **See paper for details on continuation requests, restrictions, status handling, rationales, ...**

- **Benefit of having MPI interface**: invocation as soon as implementation sees completion

```c
/** Initialize a continuation request */
int MPIX_Continue_init ( MPI_Request * cont_req );

/* Callback function signature */
void (MPIX_Continue_cb_funtion)(
    void * user_data,
    MPI_Status * statuses);

/** Attach a continuation to a single operation */
int MPIX_Continue(
    MPI_Request * request,
    int * flag,
    MPIX_Continue_cb_funtion * cont_cb,
    void * cb_data,
    MPI_Status * status,
    MPI_Request cont_req);

/** Set up continuation to be executed once all operation have completed */
int MPIX_Continueall(
    int count,
    MPI_Request * requests[],
    int * flag,
    MPIX_Continue_cb_funtion * cont_cb,
    void * cb_data,
    MPI_Status * statuses[],
    MPI_Request cont_req);
```
MPI Continuations Interface: Example

```c
omp_event_handle_t event;

/* set up continuation request */
MPI_Request contreq;
MPIX_Continue_init(&contreq);

/* task to receive data */
#pragma omp task depend (out : recvbuf) detach (event)
{
    int flag;
    MPI_Request opreq;
    MPI_Irecv(recvbuf, ..., &opreq);
    MPIX_Continue(*opreq, &flag, /* flag set to 1 if complete */
                    &complete_event, /* callback to invoke */
                    (intptr_t)event, /* argument to pass */
                    MPI_STATUS_IGNORE, contreq);
    if (flag) complete_event(event);
}

/* task to process received data */
#pragma omp task depend (in : recvbuf)
{
    process_received_data(recvbuf);
}

/* wait for all tasks to complete */
#pragma omp taskwait

MPI_Request_free (&contreq);
```

Continuation Callback

```c
void complete_event (void * cb_data, MPI_Status * status)
{
    omp_event_handle_t event = (omp_event_handle_t) cb_data;
    /* release dependencies waiting for event */
    omp_fulfill_event (event);
}
```

Progress Function

```c
void mpi_progress()
{
    int flag;
    // ignored
    MPI_Test (&contreq, &flag, MPI_STATUS_IGNORE);
}
```

⇝ Progress thread, recurring task, or service
MPI Continuations Interface: Example

```c
omp_event_handle_t event;

MPI_Request contreq;
MPIX_Continue_init(&contreq);

/* task to receive data */
#pragma omp task depend(out : recvbuf) detach(event)
{
    int flag;
    MPI_Request opreq;
    MPI_Irecv(recvbuf, ..., &opreq);
    MPIX_Continue(*opreq, &flag, /* flag set to 1 if complete */
                   &complete_event, /* callback to invoke */
                   (intptr_t)event, /* argument to pass */
                   MPI_STATUS_IGNORE, contreq);
    if (flag) complete_event(event);
}

/* task to process received data */
#pragma omp task depend(in : recvbuf)
{
    process_received_data(recvbuf);
}

/* wait for all tasks to complete */
#pragma omp taskwait
MPI_Request_free(&contreq);
```

**Continuation Callback**

```c
void complete_event(
    void *cb_data,
    MPI_Status *status)
{
    omp_event_handle_t event = (omp_event_handle_t) cb_data;
    /* release dependencies waiting for event */
    omp_fulfill_event(event);
}
```

**Progress Function**

```c
void mpi_progress()
{
    int flag;
    // ignored
    MPI_Test(&contreq, &flag, MPI_STATUS_IGNORE);
}
```

Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations :: EuroMPI'20
MPI Continuations Interface: Example

```c
omp_event_handle_t event;

/* set up continuation request */
MPI_Request contreq;
MPIX_Continue_init(&contreq);

/* task to receive data */
#pragma omp task depend(out: recvbuf) detach(event)
{
    int flag;
    MPI_Request opreq;
    MPI_Irecv(recvbuf, ... , &opreq);
    MPIX_Continue(*opreq, &flag, /* flag set to 1 if complete */
                  &complete_event, /* callback to invoke */
                  (intptr_t)event, /* argument to pass */
                  MPI_STATUS_IGNORE, contreq);
    if (flag) complete_event(event);
}

/* task to process received data */
#pragma omp task depend(in: recvbuf)
{
    process_received_data(recvbuf);
}

/* wait for all tasks to complete */
#pragma omp taskwait
MPI_Request_free(&contreq);
```

Continuation Callback

```c
void complete_event(
    void * cb_data,
    MPI_Status *status)
{
    omp_event_handle_t event = (omp_event_handle_t) cb_data;
    /* release dependencies waiting for event */
    omp_fulfill_event(event);
}
```

Progress Function

```c
void mpi_progress()
{
    int flag; // ignored
    MPI_Test(&contreq, &flag, MPI_STATUS_IGNORE);
}
```

⇝ Progress thread, recurring task, or service
MPI Continuation Interface: Implementation

Proof-of-Concept implementation in Open MPI

▶ Request without continuation: +12 instructions (≈ 2%)
▶ Request with continuation: +300 instructions, incl. registration and invocation

Test system: Dual-socket 12C Intel Haswell, ConnectX-3
MPI Continuation Interface: Implementation

Proof-of-Concept implementation in Open MPI

- Request without continuation: +12 instructions ($\approx 2\%$)
- Request with continuation: +300 instructions, incl. registration and invocation

**Test system**: Dual-socket 12C Intel Haswell, ConnectX-3

OSU P2P using `Isend/Irecv` and **MPI Continuations** to handle reply

$\rightarrow$ Small latency increase for small messages
MPI Continuations vs Argobots Integration: Message Scaling

OSU multi-threaded latency benchmark using Argobots

1 Execution Stream (thread), 1 fiber

\[ \text{Yield used in MPI implementations provides lower latencies (23\%)} \]
MPI Continuations vs Argobots Integration: Message Scaling

OSU multi-threaded latency benchmark using Argobots

1 Execution Stream (thread), 1 fiber

12 Execution Streams (threads), 12 fibers

\[
\begin{align*}
\text{Message Size [B]} & \quad \text{Latency [us]} \\
0 & \quad 10^0 \\
1 & \quad 10^1 \\
2 & \quad 10^2 \\
4 & \quad 10^3 \\
8 & \quad 10^4 \\
16 & \quad 10^5 \\
32 & \quad 10^6 \\
64 & \quad 10^7 \\
128 & \quad 10^8 \\
256 & \quad 10^9 \\
512 & \quad 10^{10} \\
1K & \quad 10^{11} \\
2K & \quad 10^{12} \\
4K & \quad 10^{13} \\
8K & \quad 10^{14} \\
16K & \quad 10^{15} \\
32K & \quad 10^{16} \\
64K & \quad 10^{17} \\
128K & \quad 10^{18} \\
\end{align*}
\]

\[\text{Continue ABT ES1 F1} \quad \text{OMPI ABT ES1 F1} \quad \text{MPICH ABT ES1 F1} \]

\[\text{Continue ABT ES12 F12} \quad \text{OMPI ABT ES12 F12} \quad \text{MPICH ABT ES12 F12} \]

\(\Rightarrow\) Yield used in MPI implementations provides lower latencies (23%)

\(\Rightarrow\) Conditional variables used in continuations provide significantly lower latencies (2 – 3×)
MPI Continuations vs Argobots Integration: Fiber Scaling

OSU multi-threaded latency benchmark using Argobots (1 B messages)

1 Execution Stream (thread), 1 – 128 fibers

12 Execution Streams, 12 – 128 fibers
MPI Continuation: NPB BT-MZ

NPB BT-MZ C++ port using Clang OpenMP detached tasks and OmpSs-2

![Graph showing speedup over C++ port]

- +1.2% over TAMPI
- +6.8% over OpenMP tasks with bulk communication
Conclusion

Reconsider convoluted use of term thread in the MPI/HPC community
Conclusion

Reconsider convoluted use of term *thread* in the MPI/HPC community

Integration of high-level concurrency abstractions in MPI potentially harmful
Conclusion

Reconsider convoluted use of term *thread* in the MPI/HPC community

Integration of high-level concurrency abstractions in MPI potentially harmful

**Proposal**: Loose coupling through *MPI Continuations*
Conclusion

Reconsider convoluted use of term *thread* in the MPI/HPC community

Integration of high-level concurrency abstractions in MPI potentially harmful

**Proposal:** Loose coupling through *MPI Continuations*

Progress still an issue, but continuation requests provide means to trigger progress
Conclusion

Reconsider convoluted use of term *thread* in the MPI/HPC community

Integration of high-level concurrency abstractions in MPI potentially harmful

**Proposal:** Loose coupling through *MPI Continuations*

Progress still an issue, but continuation requests provide means to trigger progress

Results demonstrate **efficient implementation** in Open MPI
Questions?

Reference implementation: https://github.com/devreal/ompi/tree/mpi-continue-master

(Any) Feedback welcome at: schuchart(at)hlrs.de
Google announced “ULT kernel patches”

- Adds futex switch to primitive
- Threads still managed by kernel, user space has some control
- No idea where the N:M part is here...
- Again: ULT is misleading...
- The Register: mentions fiber
- https://www.theregister.com/2020/08/10/google_scheduling_code_reaches_linux/