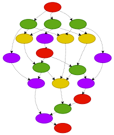
Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations

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

EuroMPI'20

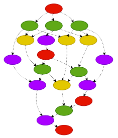
MPI and the Emergence of Asynchronous Programming Models



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, . . .)

MPI and the Emergence of Asynchronous Programming Models



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

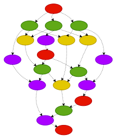
- Dispatching work to a *scheduler* for eventual execution
- *Constraints* on order of execution (dependencies, data-flow, . . .)

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

```
#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

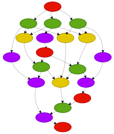
```
#pragma omp task depend(in: sendbuf)
{
  MPI_Send(sendbuf, myrank, ...);
}

#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 . . .

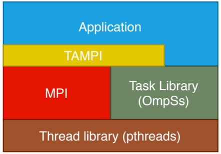
- Dispatching work to a *scheduler* for eventual execution
- *Constraints* on order of execution (dependencies, data-flow, . . .)



- 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

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

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



Application

TAMPI

MPI

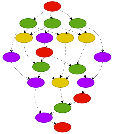
Task Library (OmpSs)

Thread library (pthreads)

MPI and the Emergence of Asynchronous Programming Models

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

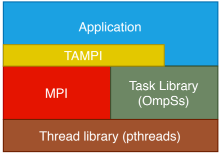
- Dispatching work to a *scheduler* for eventual execution
- *Constraints* on order of execution (dependencies, data-flow, . . .)

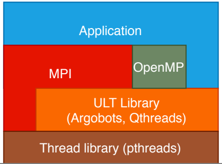


- MPI ≈ dependencies not exposed to the scheduler
- Coordinating interaction with MPI is tedious
- Test-yield cycles are inefficient, at best
- **Previously proposed**: integration of higher-level concurrency abstractions with the MPI layer

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

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

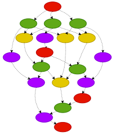




MPI and the Emergence of Asynchronous Programming Models

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

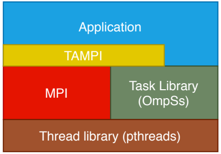
- Dispatching work to a *scheduler* for eventual execution
- *Constraints* on order of execution (dependencies, data-flow, . . .)



- 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

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

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



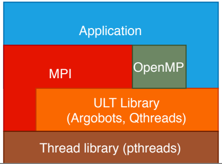
Application

TAMPI

MPI

Task Library (OmpSs)

Thread library (pthreads)



Application

MPI

OpenMP

ULT Library (Argobots, Qthreads)

Thread library (pthreads)

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
- *Pthreads 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*?

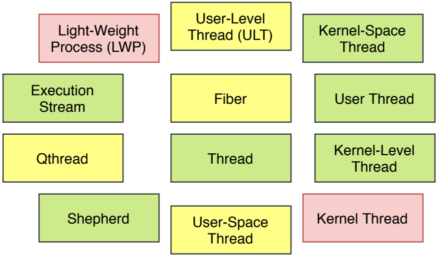
A decade-old problem...

> *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. . .

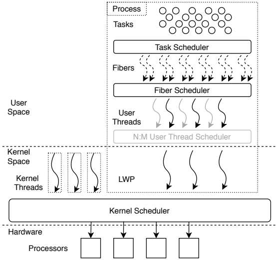
> *We're actually in a thread/process terminology crisis in Linux. Various people have various ideas about what we should mean by "thread," "process," "task," and "thread group."*
> *https://lwn.net/Articles/81790/, 2004*

16 years later, the **HPC community** is in a similar situation. . .



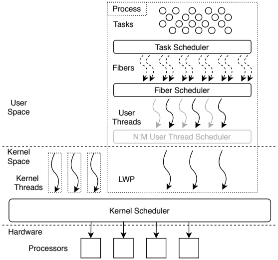
Taxonomy used in this work (excerpt)

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

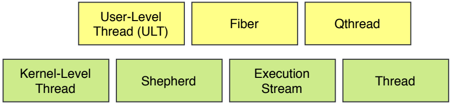


Observation:

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

Recommended read:

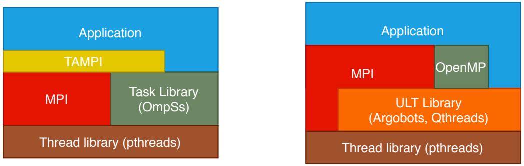
Gor Nishanov: *Fibers under the magnifying glass*, 2018.



User-Level Thread (ULT)
Fiber
Qthread

Kernel-Level Thread
Shepherd
Execution Stream
Thread

Fiber Integration (Should Be) Considered Harmful[1]



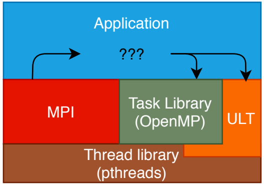
---

[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**

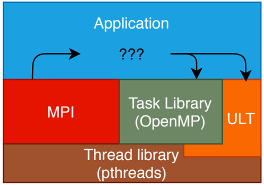
How to loosely couple different concerns?



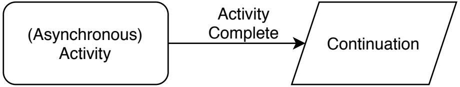
Let’s separate concerns

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

How to loosely couple different concerns?



Continuations

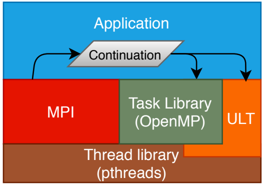


Example: `std::future::then`

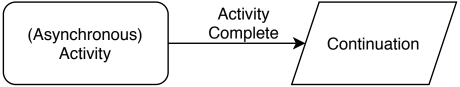
Let's separate concerns

- 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)

```
/** Initialize a continuation request */
int MPIX_Continue_init(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)

```
/** 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);
```

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)

```
/** 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

```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

```
/** 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, . . .

```
/** 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, . . .
- **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

```
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);
```

MPI Continuations Interface: Example

```
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

```
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);
}
```

MPI Continuations Interface: Example

```
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

```
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

```
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

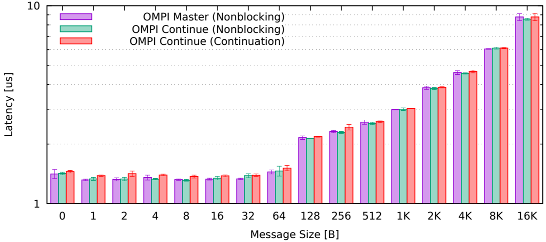
MPI Continuation Interface: Implementation

Proof-of-Concept implementation in Open MPI

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

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

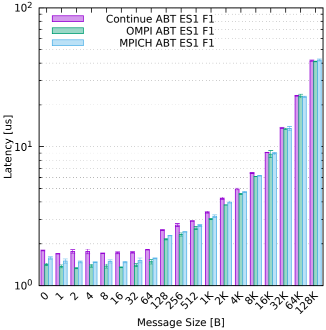
OSU P2P using `Isend`/`Irecv` and MPI Continuations to handle reply
⇝ Small latency increase for small messages



MPI Continuations vs Argobots Integration: Message Scaling

OSU multi-threaded latency benchmark using Argobots

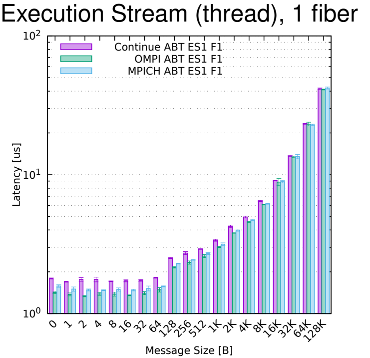
1 Execution Stream (thread), 1 fiber

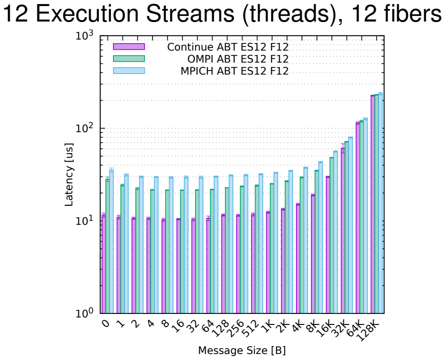


⇝ *Yield* used in MPI implementations provides lower latencies (23%)

MPI Continuations vs Argobots Integration: Message Scaling

OSU multi-threaded latency benchmark using Argobots





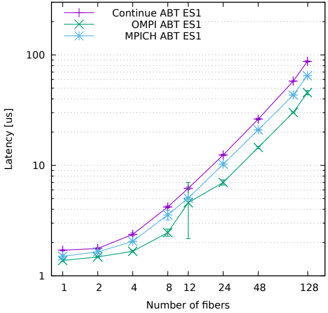
⇝ *Yield* used in MPI implementations provides lower latencies (23%)

⇝ *Conditional variables* used in continuations provide significantly lower latencies ($2 - 3\times$)

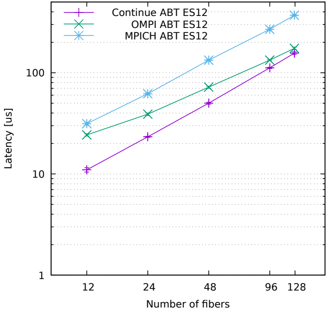
MPI Continuations vs Argobots Integration: Fiber Scaling

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

1 Execution Stream (thread), 1 – 128 fibers

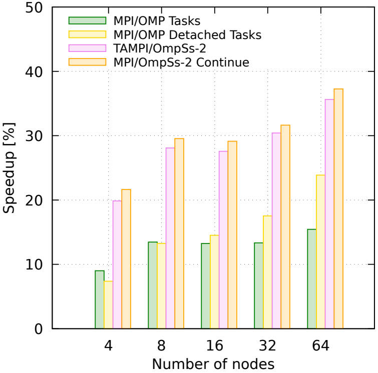


12 Execution Streams, 12 – 128 fibers



MPI Continuation: NPB BT-MZ

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



Class D, 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 switchto 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/`