Examining MPI and its Extensions for Asynchronous Multithreaded Communication

Jiakun Yan[1] Marc Snir[1] Yanfei Guo[2]

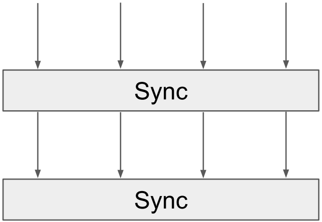
EuroMPI/USA 2025

[1]University of Illinois Urbana–Champaign
[2]Argonne National Laboratory

Introduction

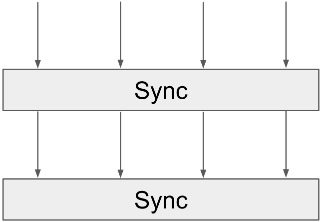
BSP and MPI Legacy

- MPI initially designed in a world where
  - Most processors are single-core.
  - Most applications are BSP.



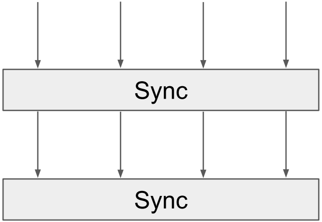
BSP and MPI Legacy

- MPI initially designed in a world where
  - Most processors are single-core.
  - Most applications are BSP.
- Initial MPI implementations:
  - Focus on single-threaded execution.
  - Optimized for coarse-grained communication.



BSP and MPI Legacy

- MPI initially designed in a world where
  - Most processors are single-core.
  - Most applications are BSP.
- Initial MPI implementations:
  - Focus on single-threaded execution.
  - Optimized for coarse-grained communication.
- Modern workloads and architectures are evolving beyond these assumptions.



HPC Trends and AMTs

- Architectures are getting more complex.
    - Modern nodes: 100+ CPU cores, 4–8 GPUs.

HPC Trends and AMTs

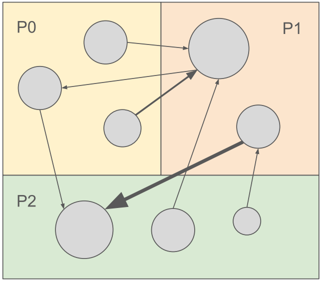
- Architectures are getting more complex.
  - Modern nodes: 100+ CPU cores, 4–8 GPUs.
- Applications are evolving towards irregularity.
  - Adaptive mesh refinement, sparse data structures, etc.

HPC Trends and AMTs

- Architectures are getting more complex.
  - Modern nodes: 100+ CPU cores, 4–8 GPUs.
- Applications are evolving towards irregularity.
  - Adaptive mesh refinement, sparse data structures, etc.
- New programming models emerge to address these trends.
  - We focus on the *Asynchronous Many-Task (AMT)* model.
    - Task oversubscription, dynamic scheduling, communication overlap.
    - Charm++, Legion, HPX, PaRSEC, StarPU, etc.

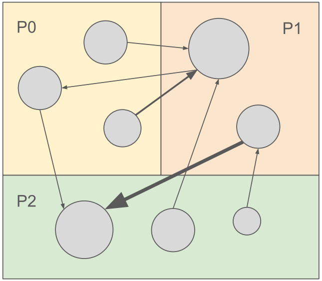
AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:



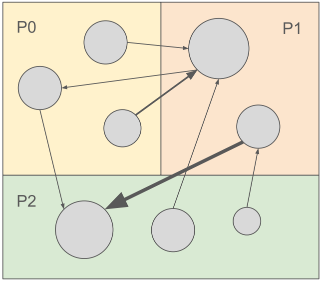
AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:
  - Fine-grained, point-to-point communication.



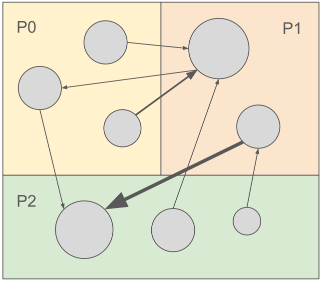
AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:
  - Fine-grained, point-to-point communication.
  - Many concurrent outstanding operations.



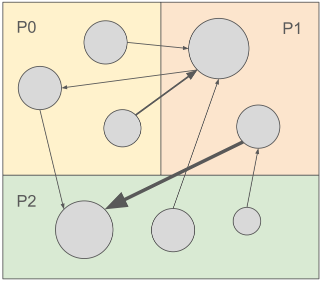
AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:
  - Fine-grained, point-to-point communication.
  - Many concurrent outstanding operations.
  - Frequent unexpected messages.



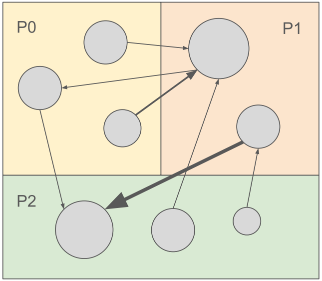
AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:
  - Fine-grained, point-to-point communication.
  - Many concurrent outstanding operations.
  - Frequent unexpected messages.
  - Multiple threads initiate and complete communication.



AMT Communication Characteristics

- AMTs exhibit distinct communication patterns:
  - Fine-grained, point-to-point communication.
  - Many concurrent outstanding operations.
  - Frequent unexpected messages.
  - Multiple threads initiate and complete communication.
- MPI has introduced extensions to better support these patterns.



Goals

- Evaluate how well MPI and recent extensions support AMT communication.
  - Using MPI-level microbenchmarks and AMT-level benchmarks.

Goals

- Evaluate how well MPI and recent extensions support AMT communication.
  - Using MPI-level microbenchmarks and AMT-level benchmarks.
- Focus on two key MPI extensions:
  - Virtual Communication Interfaces (VCIs).
  - Continuations.

Goals

- Evaluate how well MPI and recent extensions support AMT communication.
  - Using MPI-level microbenchmarks and AMT-level benchmarks.
- Focus on two key MPI extensions:
  - Virtual Communication Interfaces (VCIs).
  - Continuations.
- Focus on MPICH and HPX.

Background

MPI Threading Model

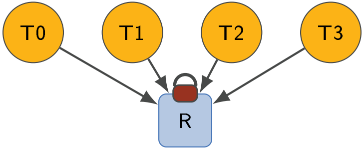
- We focus on MPI_THREAD_MULTIPLE.
  - Multiple threads can call MPI concurrently.

MPI Threading Model

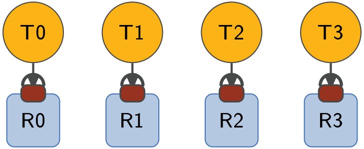
- We focus on `MPI_THREAD_MULTIPLE`.
  - Multiple threads can call MPI concurrently.
- However, thread-safety != thread efficiency.
- Common implementations serialize communication calls with coarse-grained locks.

Virtual Communication Interface (VCI) [0]

- Replicates MPI internal resources per VCI.
  - Typically via separate `MPI_Comm`.



(a) Default Case

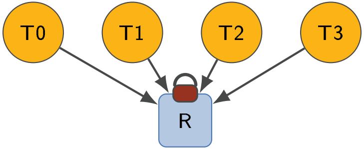


(b) With Multiple VCIs

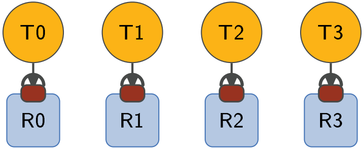
[0] Zambre et al., "How I learned to stop worrying about user-visible endpoints and love MPI." *ICS'20*.

Virtual Communication Interface (VCI) [0]

- Replicates MPI internal resources per VCI.
  - Typically via separate `MPI_Comm`.
- Each VCI is protected by its own spinlock.
  - Enables parallelism across VCIs.



(a) Default Case

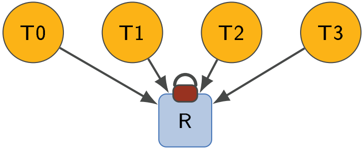


(b) With Multiple VCIs

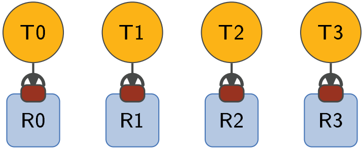
[0] Zambre et al., "How I learned to stop worrying about user-visible endpoints and love MPI." *ICS'20*.

Virtual Communication Interface (VCI) [0]

- Replicates MPI internal resources per VCI.
  - Typically via separate `MPI_Comm`.
- Each VCI is protected by its own spinlock.
  - Enables parallelism across VCIs.
- Common advice: one VCI-enabled communicator per thread.
  - *We will revisit this advice later.*



(a) Default Case

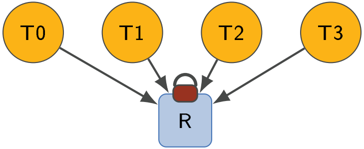


(b) With Multiple VCIs

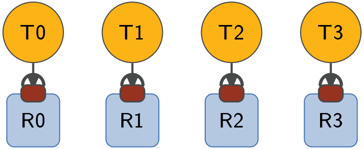
[0] Zambre et al., "How I learned to stop worrying about user-visible endpoints and love MPI." *ICS'20*.

Virtual Communication Interface (VCI) [0]

- Replicates MPI internal resources per VCI.
  - Typically via separate `MPI_Comm`.
- Each VCI is protected by its own spinlock.
  - Enables parallelism across VCIs.
- Common advice: one VCI-enabled communicator per thread.
  - *We will revisit this advice later.*
- Hybrid progress strategy:
  - By default, one global progress for every 255 local progress.



(a) Default Case



(b) With Multiple VCIs

[0] Zambre et al., "How I learned to stop worrying about user-visible endpoints and love MPI." *ICS'20*.

MPI Continuation [0]

- Callback-based completion mechanism.
- Reduces need for polling many requests.

```
int complete_cb(int rc, void *user_data);

MPI_Request cont_req;
MPIX_Continue_init(..., &cont_req);
MPI_Start(&cont_req);
// ...
MPI_Request op_req;
MPI_Irecv(..., &op_req);
MPIX_Continue(&op_req, &complete_cb,
      user_data, 0, MPI_STATUS_IGNORE,
      cont_req);
// ...
int is_done = 0;
MPI_Test(&cont_req, &is_done,
      MPI_STATUS_IGNORE);
// ...
MPI_Request_free(&cont_req);
```

[0] Schuchart et al. "Callback-based completion notification using MPI Continuations." *Parallel Computing (2021)*.

MPI Continuation [0]

- Callback-based completion mechanism.
- Reduces need for polling many requests.
- Introduces a *continuation request* object:
  - Groups requests with callbacks.
  - Tested to drive progress and manage callback execution.

```
int complete_cb(int rc, void *user_data);

MPI_Request cont_req;
MPIX_Continue_init(..., &cont_req);
MPI_Start(&cont_req);
// ...
MPI_Request op_req;
MPI_Irecv(..., &op_req);
MPIX_Continue(&op_req, &complete_cb,
      user_data, 0, MPI_STATUS_IGNORE,
      cont_req);
// ...
int is_done = 0;
MPI_Test(&cont_req, &is_done,
      MPI_STATUS_IGNORE);
// ...
MPI_Request_free(&cont_req);
```

[0] Schuchart et al. "Callback-based completion notification using MPI Continuations." *Parallel Computing (2021)*.

MPI Continuation [0]

- Callback-based completion mechanism.
- Reduces need for polling many requests.
- Introduces a *continuation request* object:
  - Groups requests with callbacks.
  - Tested to drive progress and manage callback execution.
- MPICH extension allows bypassing the continuation request (`MPI_REQUEST_NULL`) to reduce overhead.

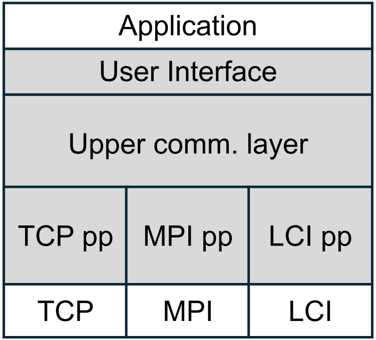
```
int complete_cb(int rc, void *user_data);

MPI_Request cont_req;
MPIX_Continue_init(..., &cont_req);
MPI_Start(&cont_req);
// ...
MPI_Request op_req;
MPI_Irecv(..., &op_req);
MPIX_Continue(&op_req, &complete_cb,
        user_data, 0, MPI_STATUS_IGNORE,
        cont_req);
// ...
int is_done = 0;
MPI_Test(&cont_req, &is_done,
        MPI_STATUS_IGNORE);
// ...
MPI_Request_free(&cont_req);
```

[0] Schuchart et al. "Callback-based completion notification using MPI Continuations." *Parallel Computing (2021)*.

HPX Integration

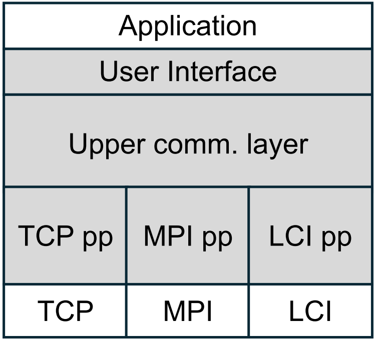
HPX Communication Stack Overview

- HPX Application Interface (Simplified).
  - An actor model.
  - Any process can invoke arbitrary actions on any other process.



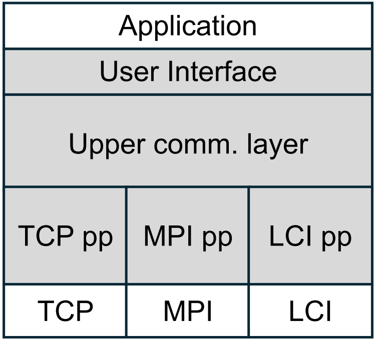
HPX Communication Stack Overview

- HPX Application Interface (Simplified).
  - An actor model.
  - Any process can invoke arbitrary actions on any other process.
- Upper Communication layer.
  - Handles serialization, aggregation, etc.



HPX Communication Stack Overview

- HPX Application Interface (Simplified).
  - An actor model.
  - Any process can invoke arbitrary actions on any other process.
- Upper Communication layer.
  - Handles serialization, aggregation, etc.
- Parcelport layer (main focus).
  - Actual send the parcel (serialized action metadata and arguments) to the target rank.
  - A parcel = one or multiple buffers.



Threading and VCIs

- Multiple worker threads (C++ Thread) per process.
- All worker threads run an infinite scheduling loop.
  - Pick and execute tasks from the task queues.
  - Periodically poll the network layer to drive communication.

Threading and VCIs

- Multiple worker threads (C++ Thread) per process.
- All worker threads run an infinite scheduling loop.
  - Pick and execute tasks from the task queues.
  - Periodically poll the network layer to drive communication.
- All threads can initiate communication and poll the network.
  - `MPI_Isend`, `MPI_Irecv`, `MPI_Test` will be called simultaneously by multiple threads.

Threading and VCIs

- Multiple worker threads (C++ Thread) per process.
- All worker threads run an infinite scheduling loop.
  - Pick and execute tasks from the task queues.
  - Periodically poll the network layer to drive communication.
- All threads can initiate communication and poll the network.
  - `MPI_Isend`, `MPI_Irecv`, `MPI_Test` will be called simultaneously by multiple threads.
- Old MPI parcelport uses a single shared communicator.

Threading and VCIs

- Multiple worker threads (C++ Thread) per process.
- All worker threads run an infinite scheduling loop.
    - Pick and execute tasks from the task queues.
    - Periodically poll the network layer to drive communication.
- All threads can initiate communication and poll the network.
    - `MPI_Isend`, `MPI_Irecv`, `MPI_Test` will be called simultaneously by multiple threads.
- Old MPI parcelport uses a single shared communicator.
- New MPIx parcelport adds an option to split the traffic across multiple communicators (VCIs).
    - Roughly, every thread has a statically assigned communicator.

Request Polling and Continuations

- The parcelport layer will have many pending communication operations –> A lot of MPI requests.

Request Polling and Continuations

- The parcelport layer will have many pending communication operations –> A lot of MPI requests.
- Old MPI Parcelport:
  - Maintain a C++ `std::deque` (Double-ended Queue) of pending requests.
  - Periodically poll (`MPI_Test`) one request in the queue in a round-robin fashion.
  - The deque is protected by a spinlock.

Request Polling and Continuations

- The parcelport layer will have many pending communication operations –> A lot of MPI requests.
- Old MPI Parcelport:
  - Maintain a C++ `std::deque` (Double-ended Queue) of pending requests.
  - Periodically poll (`MPI_Test`) one request in the queue in a round-robin fashion.
  - The deque is protected by a spinlock.
- Two approaches in MPIx Parcelport:

Request Polling and Continuations

- The parcelport layer will have many pending communication operations –> A lot of MPI requests.
- Old MPI Parcelport:
    - Maintain a C++ `std::deque` (Double-ended Queue) of pending requests.
    - Periodically poll (`MPI_Test`) one request in the queue in a round-robin fashion.
    - The deque is protected by a spinlock.
- Two approaches in MPIx Parcelport:
    - Replicate the request pool with the communicator.

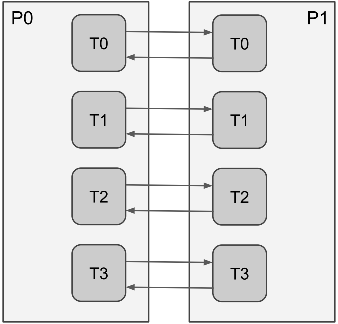
Request Polling and Continuations

- The parcelport layer will have many pending communication operations –> A lot of MPI requests.
- Old MPI Parcelport:
  - Maintain a C++ `std::deque` (Double-ended Queue) of pending requests.
  - Periodically poll (`MPI_Test`) one request in the queue in a round-robin fashion.
  - The deque is protected by a spinlock.
- Two approaches in MPIx Parcelport:
  - Replicate the request pool with the communicator.
  - Replace polling with Continuations.

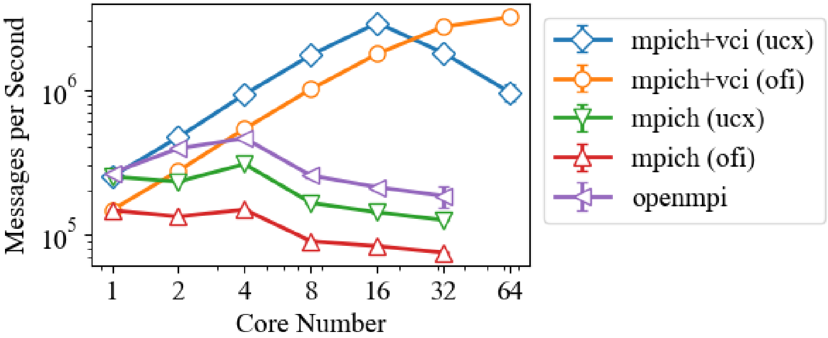
Experiments

MPI-level Multithreaded Ping-Pong

- 2 nodes; 1 MPI rank/node; N threads; thread-to-thread ping-pong.
- 8-byte messages.
- Turn hybrid progress and continuation request off.
- Platform: SDSC Expanse (IB) and NCSA Delta (SS-11).

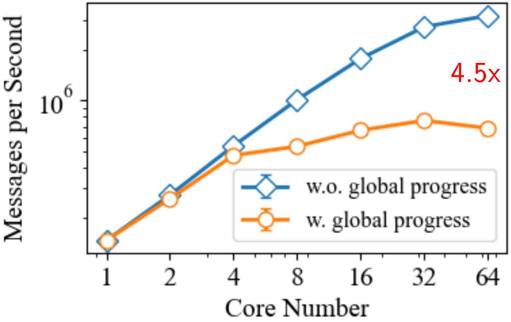


VCI Impact



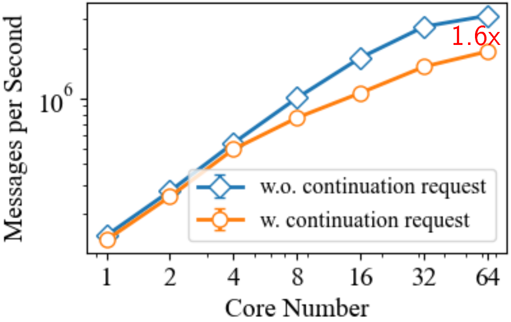
Multi-VCI improves multithreaded message rate; Trade-off between UCX/OFI.

Occasional Global Progress Cost



Occasional global progress increases cross-VCI contention.

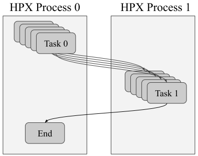
Continuation Request Overhead



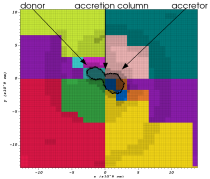
Current continuation request adds overhead at high thread counts.

HPX Benchmarks

- Flood microbenchmark:
  - 2 nodes, 1 process/node, 63 threads/process.
  - Process 0 sends a flood of messages to process 1.
  - 8 bytes or 16 KiB payloads.



HPX flooding

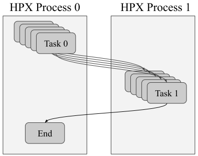


OctoTiger

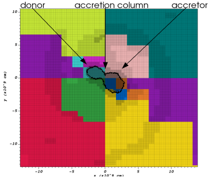
[0] Yan et al. "Design and Analysis of the Network Software Stack of an Asynchronous Many-task System–The LCI parcelport of HPX." *PAW-ATM (2023)*.

HPX Benchmarks

- Flood microbenchmark:
  - 2 nodes, 1 process/node, 63 threads/process.
  - Process 0 sends a flood of messages to process 1.
  - 8 bytes or 16 KiB payloads.
- OctoTiger application:
  - Astrophysics simulation of binary star systems.
  - Fast multipole method + adaptive mesh refinement.
  - 32 nodes, 2 processes/node, 63 threads/process.



HPX flooding

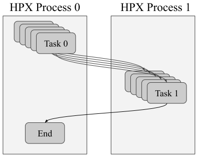


OctoTiger

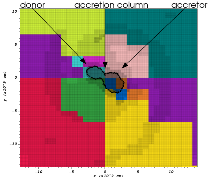
[0] Yan et al. "Design and Analysis of the Network Software Stack of an Asynchronous Many-task System–The LCI parcelport of HPX." *PAW-ATM (2023)*.

HPX Benchmarks

- Flood microbenchmark:
  - 2 nodes, 1 process/node, 63 threads/process.
  - Process 0 sends a flood of messages to process 1.
  - 8 bytes or 16 KiB payloads.
- OctoTiger application:
  - Astrophysics simulation of binary star systems.
  - Fast multipole method + adaptive mesh refinement.
  - 32 nodes, 2 processes/node, 63 threads/process.
- Comparing: standard MPI, LCI [0], MPIx, MPIx (w/o continuation).



HPX flooding

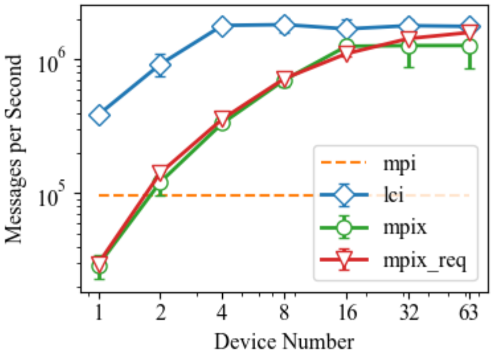


OctoTiger

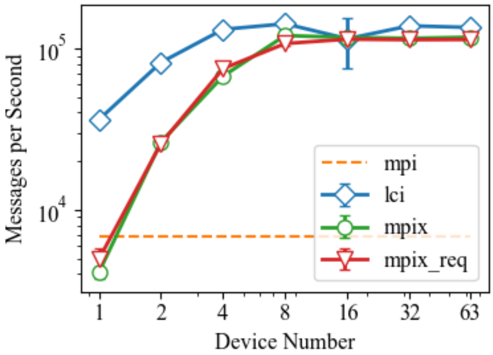
[0] Yan et al. "Design and Analysis of the Network Software Stack of an Asynchronous Many-task System–The LCI parcelport of HPX." *PAW-ATM (2023)*.

Flood Microbenchmark

(a) 8-byte payloads.

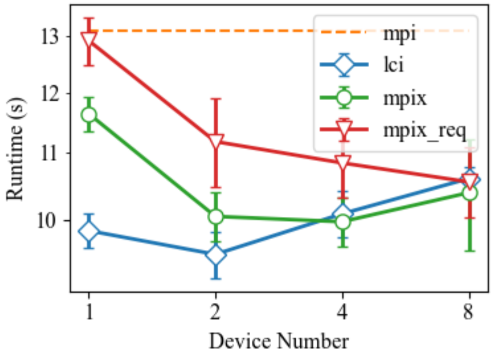


(b) 16KiB payloads.



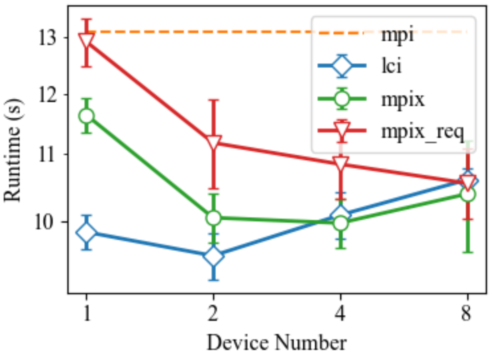
MPIx (especially the usage of VCIs) closes gap vs LCI.

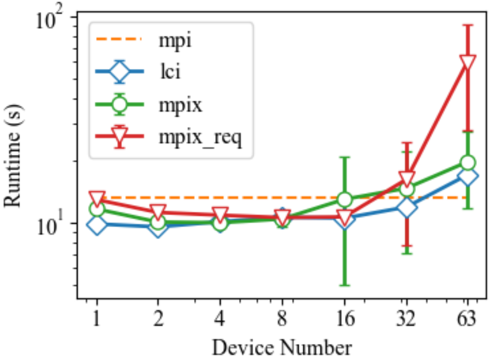
OctoTiger



VCI keeps help, but too many VCIs hurt. Continuation benefits are modest.

OctoTiger





VCI keeps help, but too many VCIs hurt. Continuation benefits are modest.

Why Too Many VCIs Hurt?

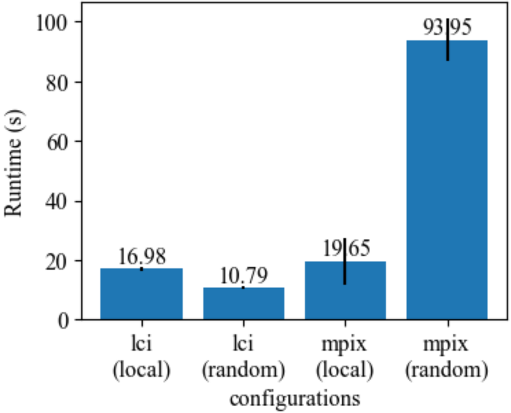
- Both LCI and MPIx show similar trends – some generic issue.

Why Too Many VCIs Hurt?

- Both LCI and MPIx show similar trends – some generic issue.
- Assumption: Lack of attentiveness due to heavy computation.
  - Each VCI is polled by only one thread.
  - If that thread is busy computing, the VCI is not polled often enough.
  - Even if other threads are idle, they cannot help.

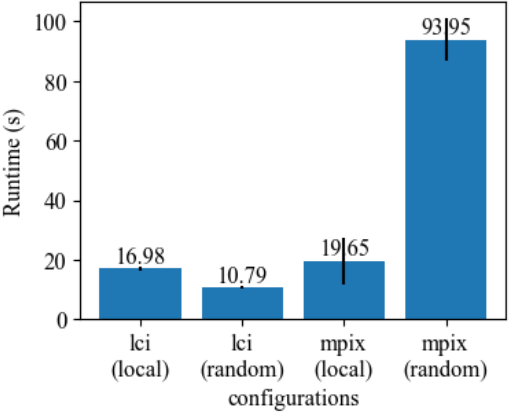
Random Progress Strategy

- To verify attentiveness assumption:
  - Randomly select a VCI/device to poll.



Random Progress Strategy

- To verify attentiveness assumption:
  - Randomly select a VCI/device to poll.
- Random polling helps LCI but hurts MPIx
  - VCI has coarser-grained locking than LCI device.
  - Tradeoff between attentiveness and contention.



Conclusion

Conclusion

- VCI greatly boosts multithreaded performance.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.
- UCX and OFI shows different multithreaded efficiency, but both not ideal.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.
- UCX and OFI shows different multithreaded efficiency, but both not ideal.
- Global progress hurts when multithreading.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.
- UCX and OFI shows different multithreaded efficiency, but both not ideal.
- Global progress hurts when multithreading.
- Continuation request brings noticeable overhead under heavily multithreading.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.
- UCX and OFI shows different multithreaded efficiency, but both not ideal.
- Global progress hurts when multithreading.
- Continuation request brings noticeable overhead under heavily multithreading.
- One VCI per thread may not be perfect: attentiveness problem.

Conclusion

- VCI greatly boosts multithreaded performance.
- Continuation shows modest performance benefits.
- UCX and OFI shows different multithreaded efficiency, but both not ideal.
- Global progress hurts when multithreading.
- Continuation request brings noticeable overhead under heavily multithreading.
- One VCI per thread may not be perfect: attentiveness problem.
- Some level of sharing is needed. We need better VCI implementation than coarse-grained locks.

Q&A

Questions?

Examining MPI and its Extensions for Asynchronous Multithreaded Communication

Jiakun Yan, Marc Snir and Yanfei Guo

jiakuny3@illinois.edu