UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

# Lightweight Communication Interface:

## High-Performance Communication Support for Asynchronous Many-Task Systems

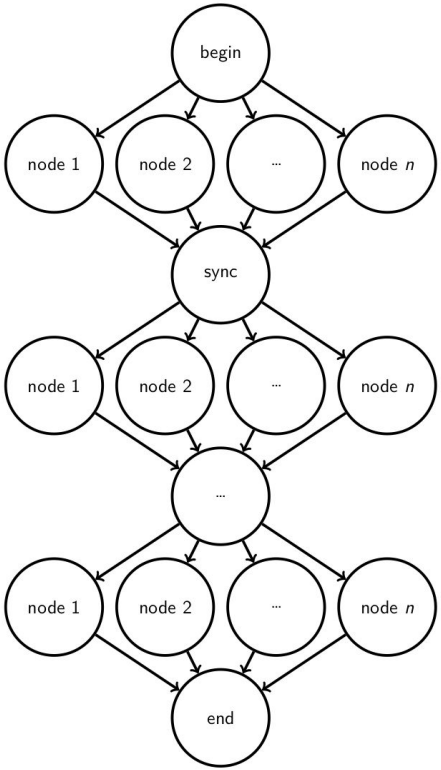
Jiakun Yan, Omri Mor, Marc Snir

Charm++ Workshop 2024

# Traditionally

## Bulk-Synchronous Programming (BSP)

All processes work in lock steps.



# Traditional Comm. Characteristics

Collective communication

Coarse-grained messages

No or a few pending operations

Single thread

# Now

- Architectures are getting heterogeneous with more on-node parallelism
  - More CPU cores per node.
  - Couple of GPUs per node.
  - Uneven cores.
- Programming models are getting asynchronous, over-subscripted, dynamic.
  - **Asynchronous Many-Task Systems**!
- More and more irregular, dynamic applications.
  - Graph analytics.
  - Sparse algebra.
  - Adaptive algorithms.

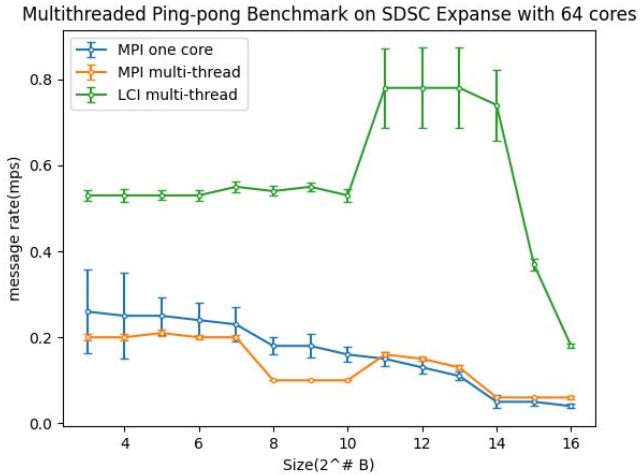
# Shifts in Comm. Characteristics

| | |
|---|---|
| Collective communication | Point-to-point communication |
| Coarse-grained messages | Fine-grained messages |
| No or a few pending operations | Larger number of concurrent messages |
| Single thread | Multiple threads |

The optimization focus of traditional communication libraries is still mainly on the left. This motivates the Lightweight Communication Interface (LCI) project.

# Lightweight Communication Interface (LCI)

- Designed with AMTs as the target clients.
  - Should also apply to other irregular applications such as graph analytics/sparse linear algebra.
- A research tool.
  - Help us understand how to structure communication libraries and design their interfaces to better match AMTs' needs.
- Focuses:
  - Point-to-point communication.
  - Fine-grained messages.
  - Many pending operations.
  - Multithreaded environment.



# LCI: Current Status

- C interface with C/C++ implementation.
- Network backends:
  - libibverbs (for Infiniband)
  - libfabric (for all others, e.g. Slingshot-11)
  - UCX (experimental)
- Existing clients and collaborators:
  - Gluon[1]: graph analytics (Keshav Pingali)
  - PaRSEC[2]: AMT (George Bosilca)
  - HPX[3]: AMT (Hartmut Kaiser)

[1] H. -V. Dang *et al.*, "A Lightweight Communication Runtime for Distributed Graph Analytics," *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC, Canada, 2018, pp. 980-989, doi: 10.1109/IPDPS.2018.00107.
[2] Mor, Omri, George Bosilca, and Marc Snir. "Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine." *Proceedings of the 52nd International Conference on Parallel Processing*. 2023.
[3] Yan, Jiakun, Hartmut Kaiser, and Marc Snir. "Design and Analysis of the Network Software Stack of an Asynchronous Many-task System--The LCI parcelport of HPX." *Proceedings of the SC'23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis*. 2023.

# Interface

# LCI Interface: Principles

- Flexibility
  - Provide users with versatile communication primitives and synchronization mechanisms.
- Explicit control
  - Give users explicit control of communication resources and behaviors.

Essentially, give users versatile and orthogonal options to best suit their needs.

# Communication Primitive

- For a point-to-point communication to happen, we need
  - source buffer & target buffer
- Two-sided Send/Receive:
  - Sender specifies the source buffer.
  - Receiver specifies the target buffer.
- One-sided Put:
  - Sender specifies both the source and target buffers.
- One-sided Get:
  - Receiver specifies both the source and target buffers.
- One-sided Put “allocate”:
  - Sender specifies the source buffer. LCI allocates the target buffers.

# Communication Protocol

- Depending on how much data to transfer.
  - Four protocols are available.
- Small message (e.g. <= 32B).
  - Inline protocol (eager protocol without memcpy).
- Medium (e.g. <= 8KB).
  - Buffer copy protocol (eager protocol with memcpy).
- Long (arbitrary size).
  - Rendezvous protocol.
  - No memory copy but additional handshakes.
- IO-VEC (one medium + multiple long).
  - Rendezvous protocol with merged handshakes.

All thresholds are visible and configurable. Users can explicitly choose which communication protocol to use.

# Completion Mechanism

- LCI offers four completion notification mechanisms:
  - Synchronizers
  - Completion queues
  - Callback functions
  - No completion
- More details: synchronizers.
  - Completion objects for individual operations similar to MPI requests, but
    - Can have multiple producers.
    - Completely thread-safe.

# Other options

- (For eager messages) direct access to internal pre-registered buffers (packets).
- (For rendezvous messages) direct access to memory registration.
- Explicit control over progress engine invocation.
- Explicit low-level resource representation (LCI device) and the ability to replicate it.

# Interface

- Who provides the source/target buffers
  - Two-sided send/recv
  - One-sided put/get
- How much data to send
  - Short (inline), medium (eager), long (zero-copy)
  - iovec (one medium + multiple long)
- Completion mechanisms
  - Synchronizer
  - Completion queue
  - Active message handler
  - No signaling
- For eager message, whether the source/target buffers are user-provided or LCI-provided
  - Using LCI-provided buffers can potentially save one memory copy
- For long messages, whether the source/target buffers are registered.
- How to match the send and recv
  - tag only/rank+tag
- Explicit progress engine behavior control
- Explicit low-level resource (LCI devices) representation and replication

**All options are orthogonal.**

**Users can choose any valid combination.**

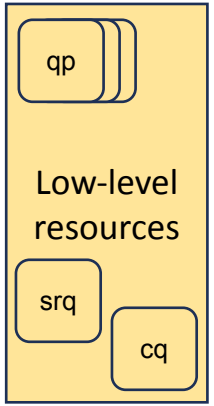
# Design

# Overview

- Focus on libibverbs as the native network interface.
  - To some extent, libfabric/UCX can be viewed as a slightly simpler/higher-level version of it.
- Focus on the the eager protocol (medium messages).
  - The inline protocol (short messages) is very similar.
  - The rendezvous protocol (long messages) used in LCI is similar to those used in other communication libraries.
- Focus on send/recv, put allocate primitives.
  - Regular put/get are simply RDMA operation wrappers.

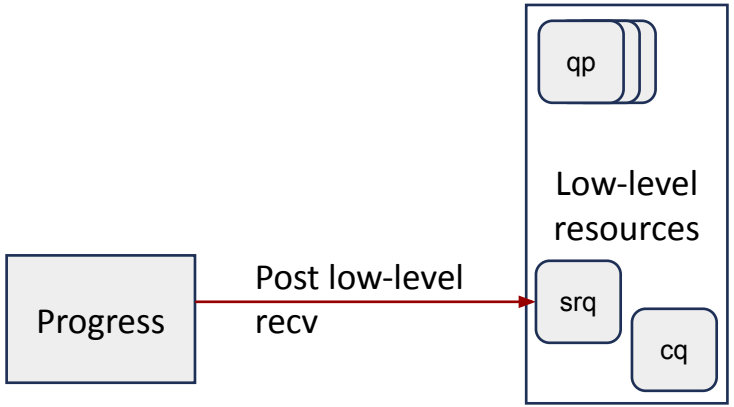
# Two concepts

- Low-level resources
  - Queue pairs (send&recv queues)
  - Shared receive queues
  - completion queues
- Progress engine
  - Responsible for all background works needed to support communications.
  - Implicit in MPI (MPI_Test…).
  - Explicit in LCI (LCI_progress).

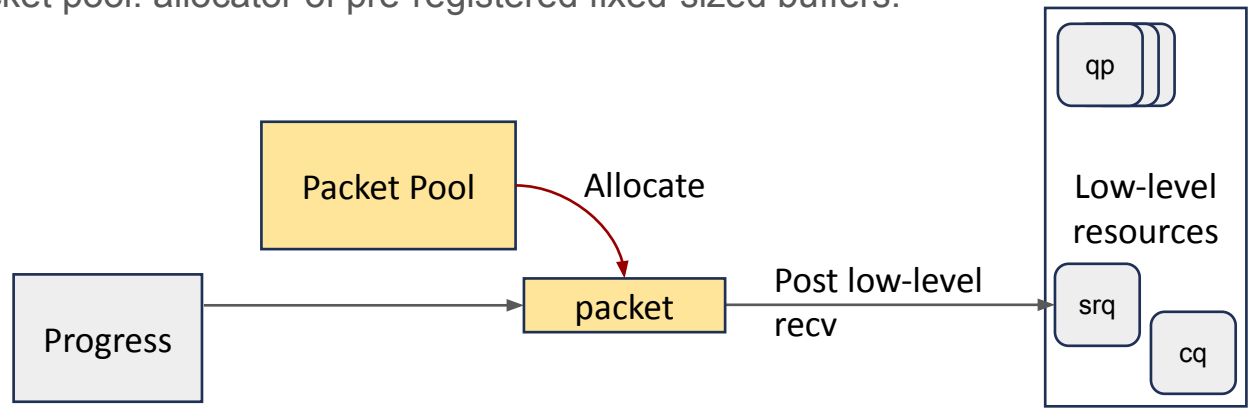
# Prepost receives

- libibverbs does not like unexpected messages.
  - Receive-Not-Ready errors are devastating to application's performance.
- Progress engine always makes sure there are enough preposted receives.
  - So all messages are expected.

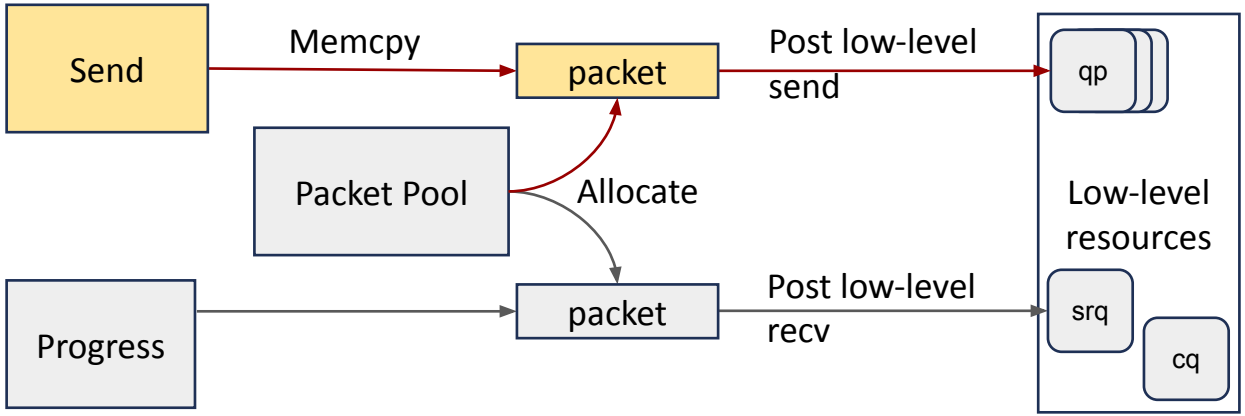


# Packet pool

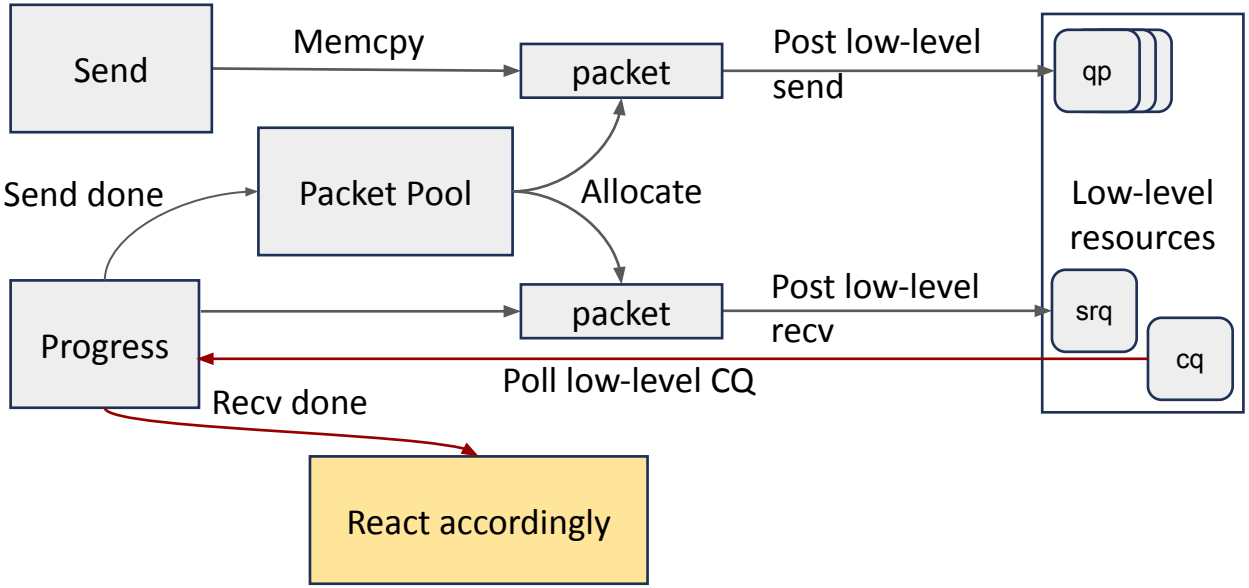
Libibverbs requires all communication buffers to be registered.
Packet pool: allocator of pre-registered fixed-sized buffers.



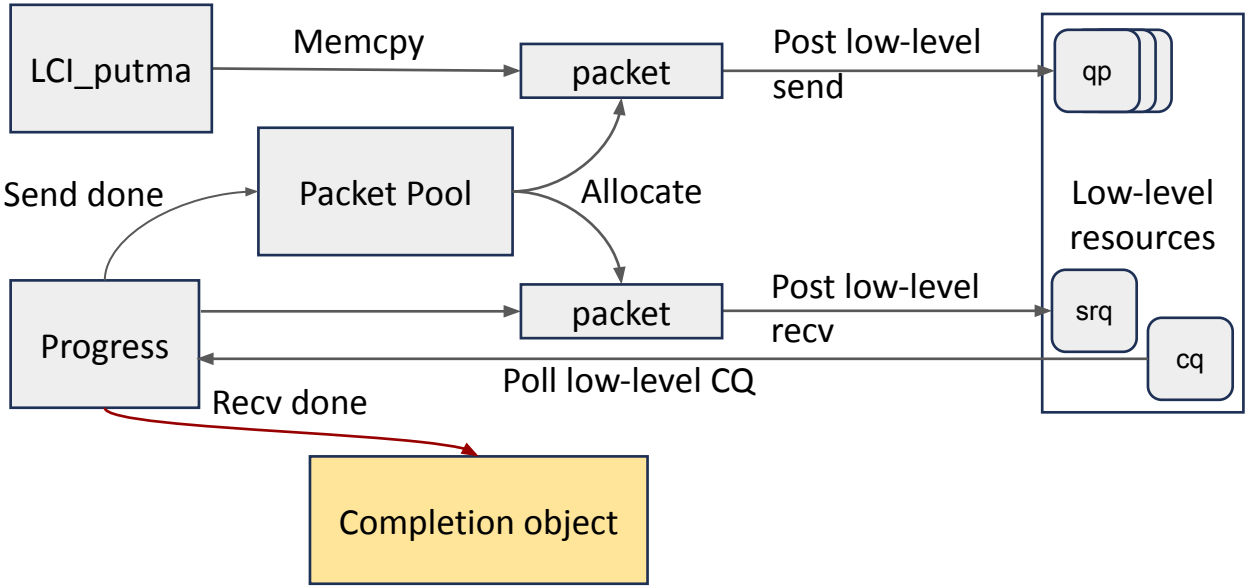
# Send messages



# Polling for incoming messages

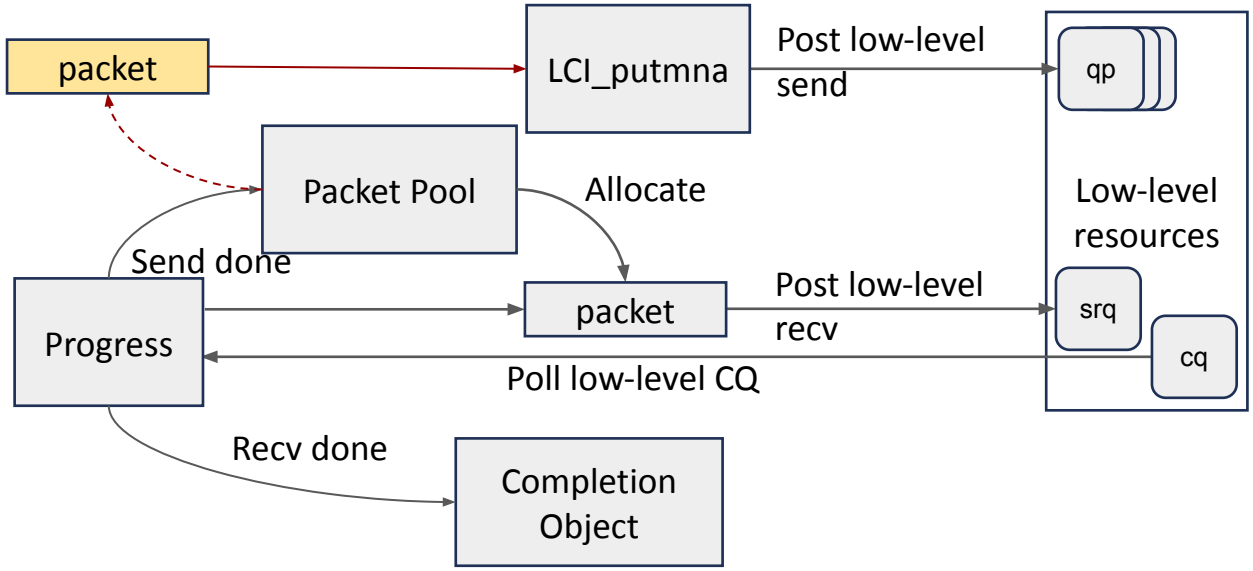


# LCI_putma: put medium allocate

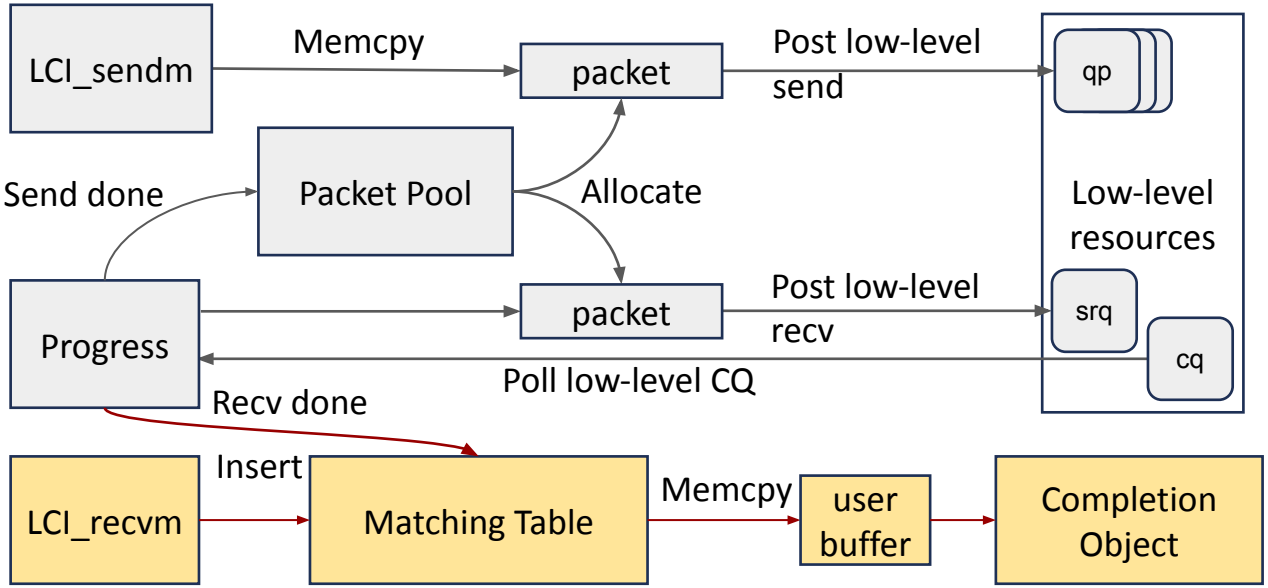


# LCI_putmna: LCI put medium “no-copy” allocate

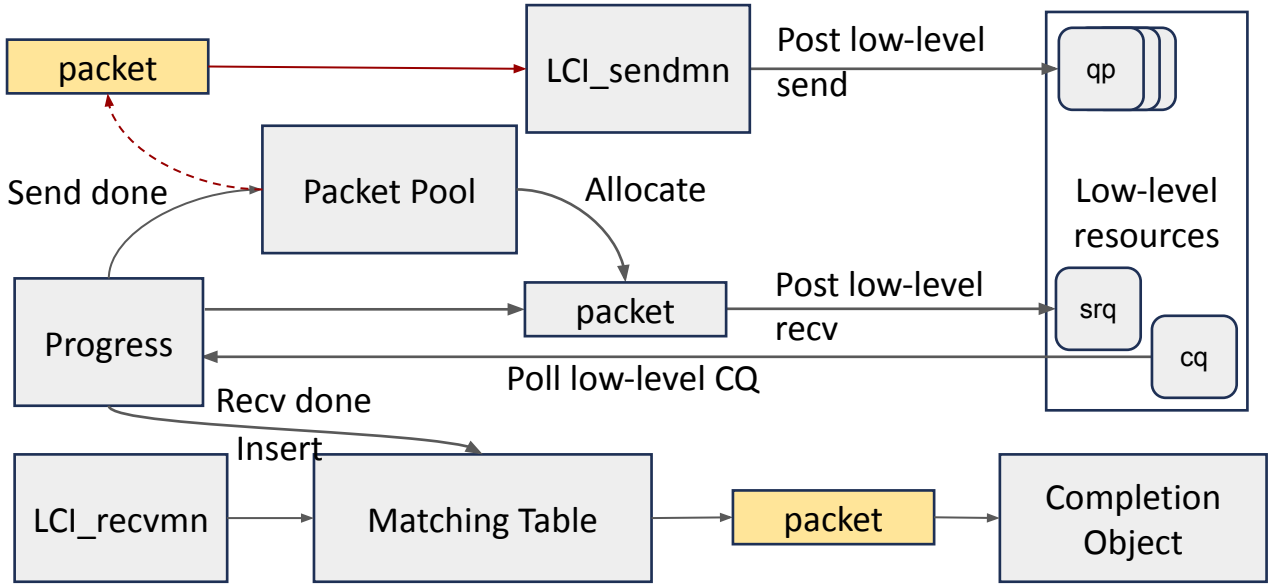
- Users can directly assemble their message in a pre-allocated/reused packet.



# LCI_sendm/recvm: send/recv medium



# LCI_sendmn/recvmn: send/recv medium “no-copy”



# Tag matching: more details

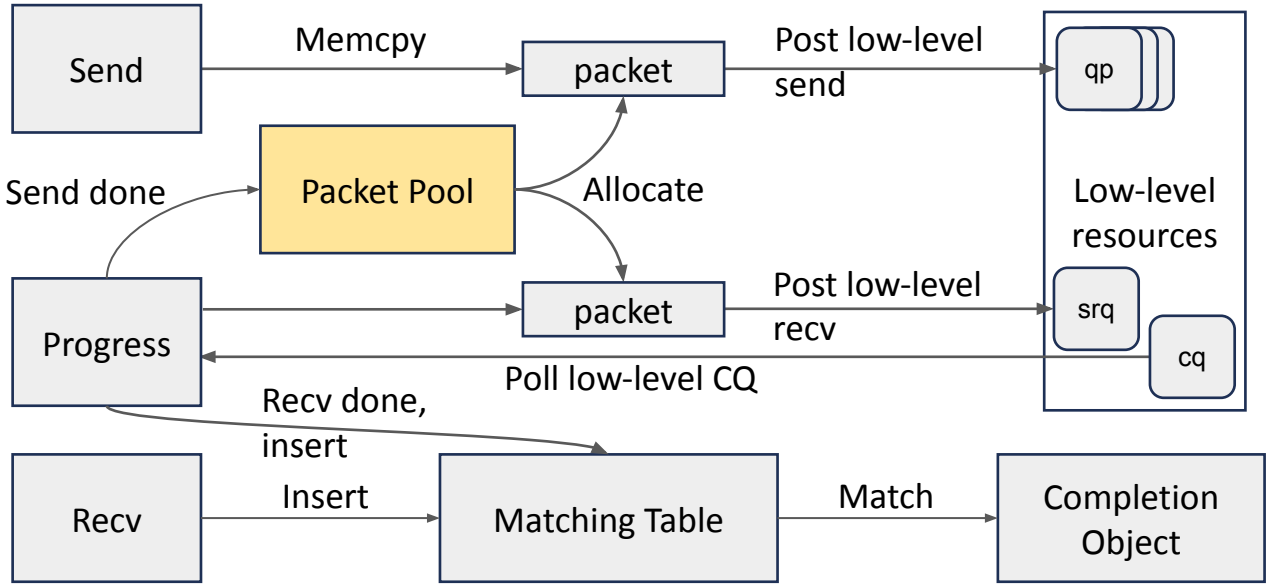
- What to support:
  - Ordering
  - Wildcard tag (MPI_ANY_TAG)
  - Wildcard source (MPI_ANY_SOURCE)
- MPI chooses to support all three options.
  - This is why MPI has to use matching queues
  - and one of the reasons multithreaded MPI is slow.
- LCI chooses to selectively support some of them.
  - No ordering guarantee.
    - If you care about ordering, just use different tags.
  - No wildcard tag.
    - Do AMTs really need it?
  - Support wildcard source (a slightly weaker form).
    - Essentially, the sender also need to agree they are wildcard-source messages.
    - So LCI can hash them to the same buckets.
- So LCI is able to use a hash table with per-bucket locks.

# Optimizing Multithreaded Performance

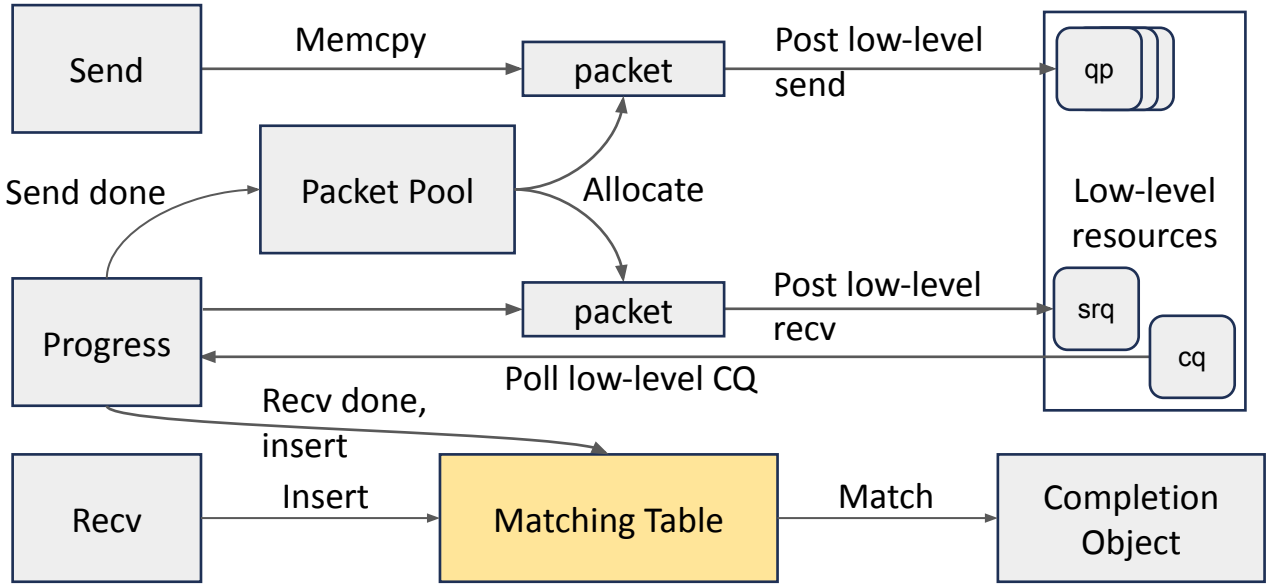
- Four data structures come into play.
  - Low-level network resources (encapsulated in LCI devices).
  - Packet pools.
  - Matching tables.
  - Completion objects.

We carefully designed these data structures and their interactions so there is no centralized thread contention point at LCI.

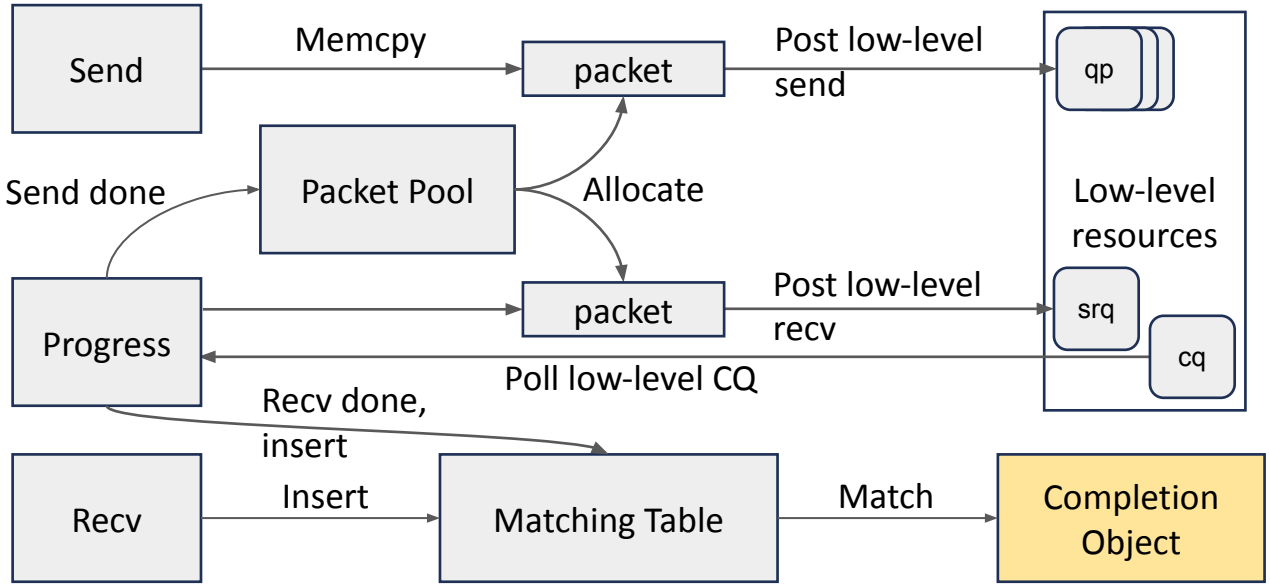
## Packet pool: thread-local queues with random packet stealing



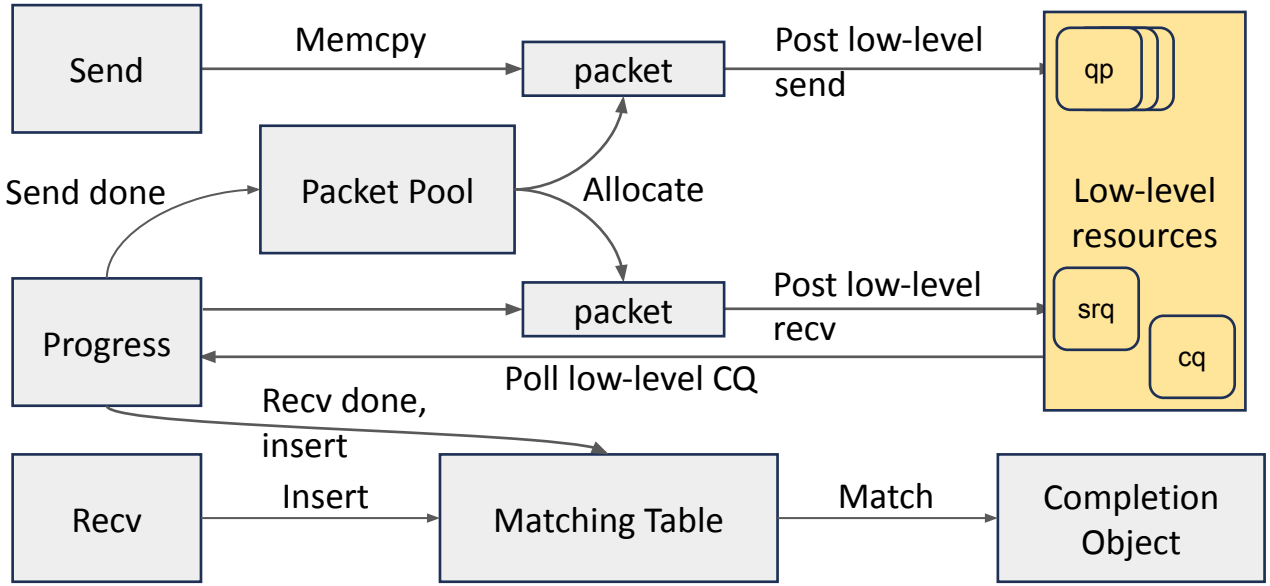
## Matching Table: hash table with per-bucket locks



Completion object: Atomic-based synchronizer/queue; Replication.



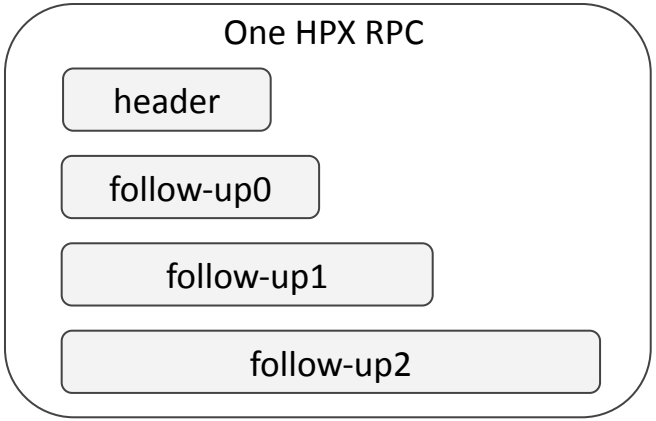
Low-level resources: No coarse-grained locks; Replication.



# Case Study: HPX LCI parcelport

# Background

- HPX has a communication abstraction similar to RPCs/Active Messages.
  - Invoke a function with provided arguments on remote processes.
  - All threads can make such RPC calls.
- After the argument serialization, the job is to transfer the following messages per RPC invocation:
  - One header message:
    - A short message with a size upper bound.
    - Mainly for transferring control information.
    - Can piggyback small data.
  - Optional multiple follow-up messages:
    - For data transfer.
    - Of arbitrary size.

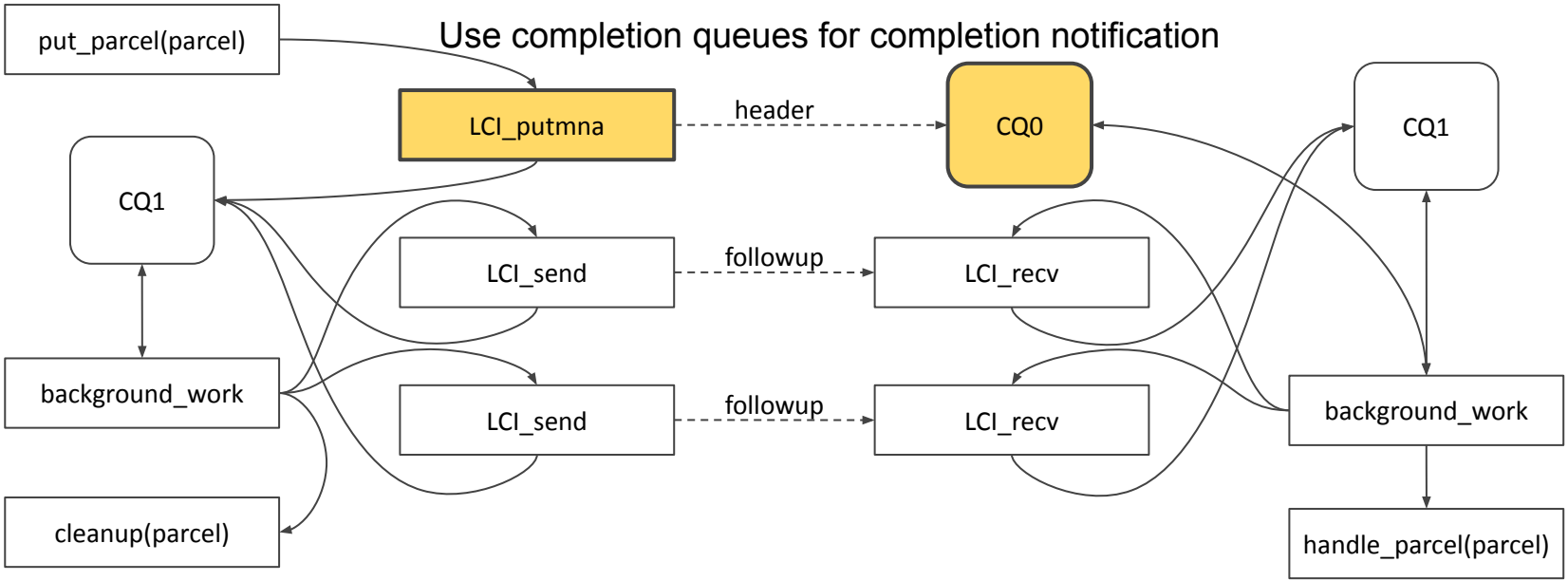


# LCI Parcelport

**Use LCI_putmna (eager put w. target buffer allocation “no-copy”) for header message.**

Use LCI_send/recv for follow-up messages

Use completion queues for completion notification

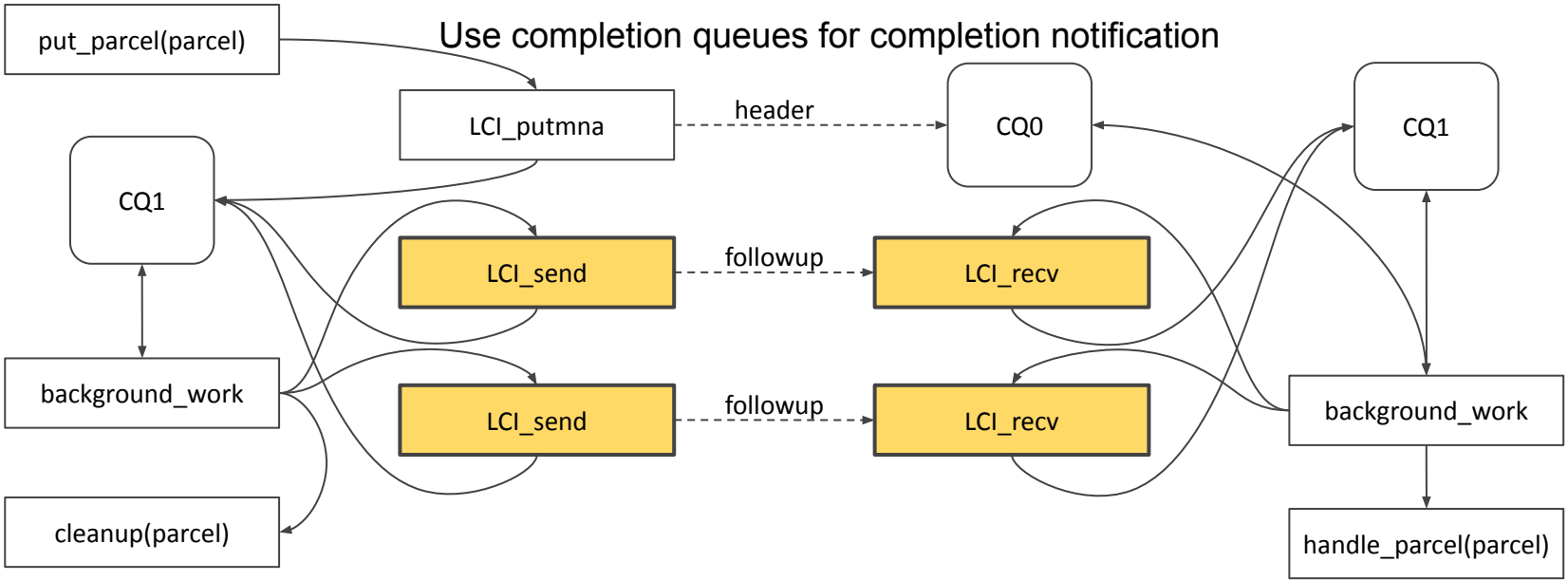


# LCI Parcelport

Use LCI_putma (eager put w. target buffer allocation “no-copy”) for header message.

**Use LCI_send/recv for follow-up messages**

Use completion queues for completion notification

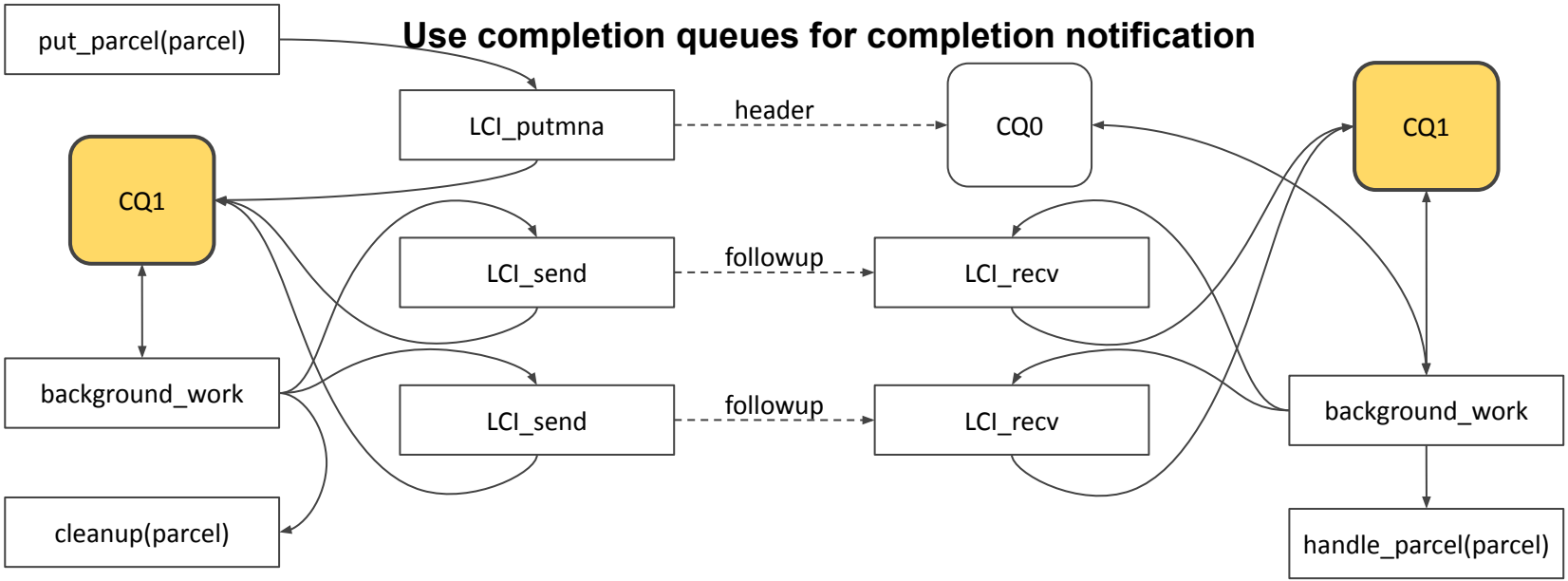


# LCI Parcelport

Use LCI_putma (eager put w. target buffer allocation “no-copy”) for header message.

Use LCI_send/recv for follow-up messages

**Use completion queues for completion notification**



# LCI Parcelport

In addition,

- Explicit progress control.
  - LCI parcelport can spawn dedicated progress threads.
- Low-level network resource replication.
  - LCI parcelport can allocate multiple LCI devices and split communication among them.

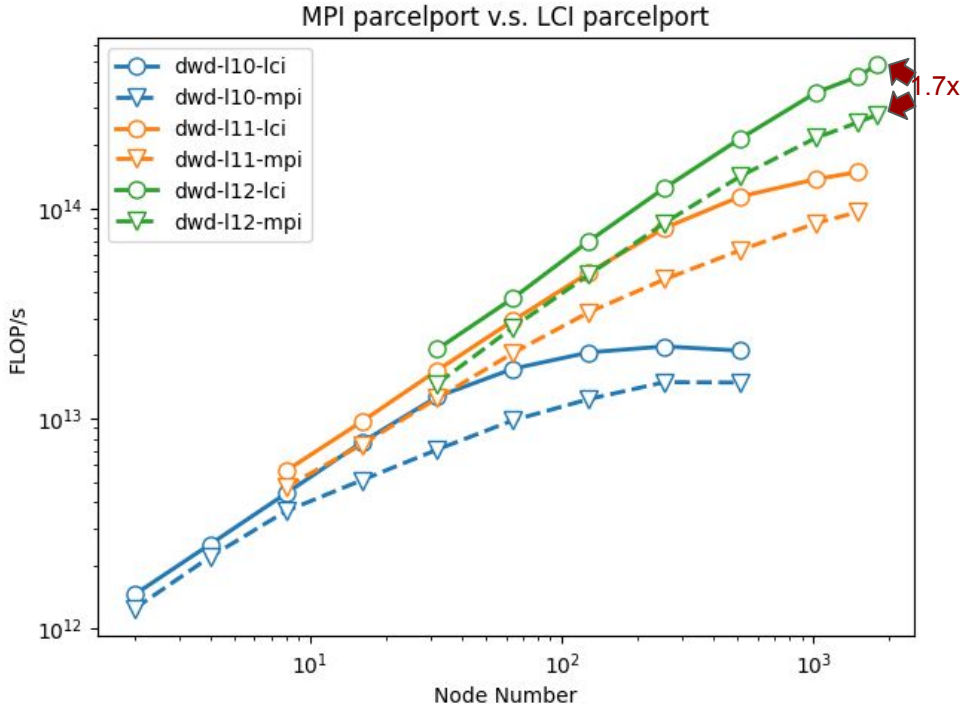
# Comparing MPI Parcelport

- Have to use preposted receives + request testing for header messages.
  - Additional tag matching, ordering, memory copy overheads.
- Have to acquire locks when testing the shared preposted receive.
  - Could be worse when considering progress engine.
- Have to use a request pool to manage a large number of pending operations.
  - Locking overhead.
  - Overhead of testing individual requests.
- MPI typically uses coarse-grained locks on low-level resources.
  - Lost concurrency.
- Difficulties of replicating low-level resources.
  - No portable way across MPI vendors.
  - Scalability issues.

# Performance

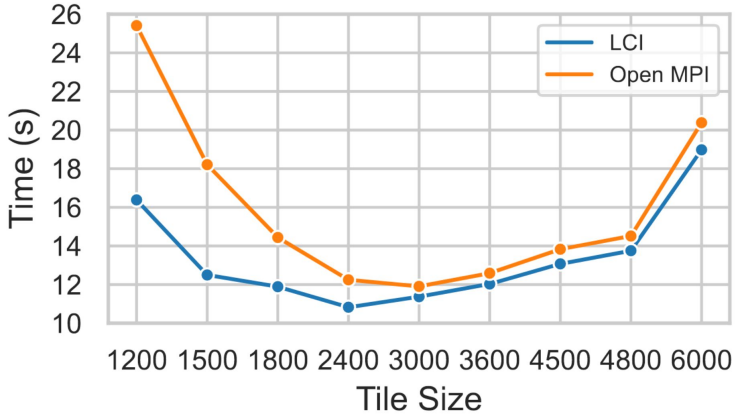
# Octo-Tiger (on top of HPX+MPI/LCI)

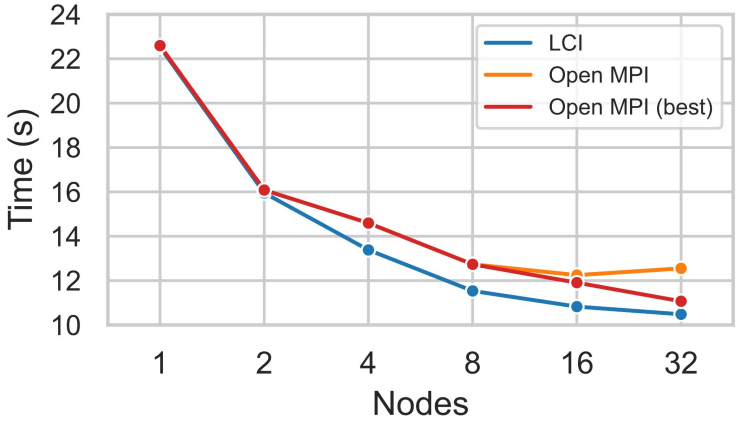
- Overview:
  - Astrophysical application simulating steller merger.
  - Adaptive mesh refinement + Fast Multipole method.
  - Built on top of HPX + Kokkos.
- Full-system runs (1720 GPU nodes) on Perlmutter
  - Production settings.
  - LCI achieves **1.7x** speedup over MPI
- Similar speedups (1.5x-2.5x) on other platforms
  - SDSC Expanse (128 nodes), Frontera (256 nodes), NCSA Delta (64 nodes), Ookami (128 nodes)



# HiCMA (on top of PaRSEC+MPI/LCI)

- Overview
  - Tile-based low-rank Cholesky approximation.
  - Used in applications such as geostatistical modeling.
  - Built on top of PaRSEC
- LCI outperforms MPI by 12%.



# Lightweight Communication Interface:

## High-Performance Communication Support for Asynchronous Many-Task Systems

Source code & Document & Examples: https://github.com/uiuc-hpc/lci

If you are interested in porting your system onto LCI, let us know!

**Q&A**: Jiakun Yan (jiakuny3@illinois.edu)