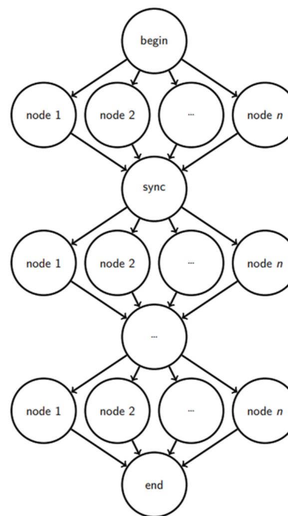
An Implementation of LCI Network Backend Using UCX

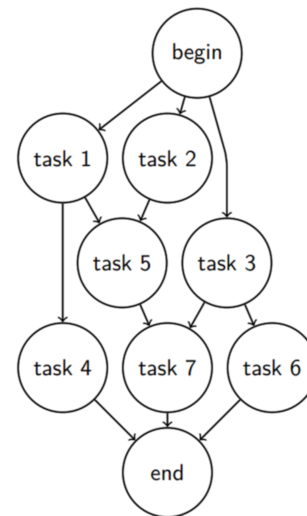
Weixuan Zheng, Jiakun Yan, Omri Mor, Marc Snir
University of Illinois Urbana-Champaign

Introduction

BSP v.s. AMTs

- MPI + Bulk-Synchronous Programming (BSP) dominates traditional HPC programming.
- BSP has its limitations
  - Unnecessary global synchronization.
  - High demands for balanced loads across processes.
- Asynchronous many-task runtime systems (AMTs) can potentially solve the limitations.
  - Computation-communication overlapping.
  - Automatic load balancing of fine-grained tasks.
  - Remove unnecessary synchronization.





New Communication Characteristics

- AMTs lead to different communication characteristics.
  - More messages.
  - Smaller messages.
  - Dynamic communication patterns.
  - More point-to-point communications.
  - Multithreaded environment.
- Not unique for AMTs, but other irregular applications.
  - Graph analytics.
  - Sparse linear algebra.
  - Fast Multipole Method.
- MPI is not optimized for these cases.

LCI Overview

LCI: Current Status

- Designed to work better in the context of new communication characteristics
- A low-level communication library with C API.
- Current backends:
  - libibverbs
  - libfabric
  - **UCX**
- Existing clients/collaborators:
  - Gluon, D-Galois, D-Ligra[1]
  - PaRSEC[2]
  - HPX[3]

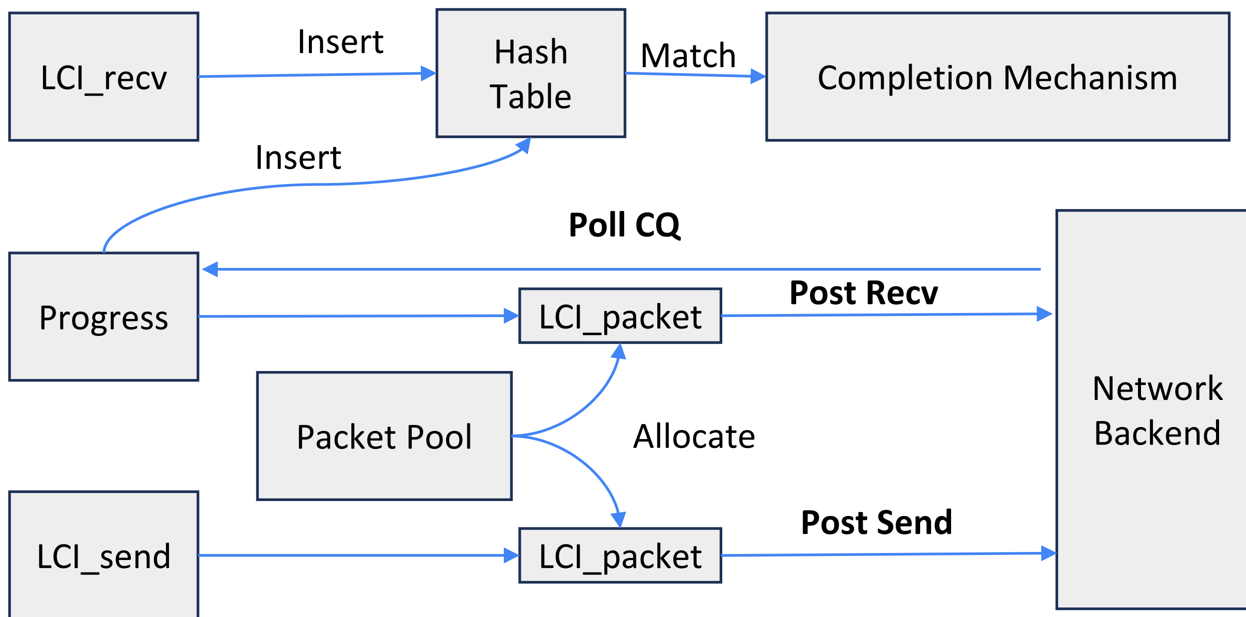
[1] Dathathri, Roshan, et al. "Gluon: A communication-optimizing substrate for distributed heterogeneous graph analytics." *Proceedings of the 39th ACM SIGPLAN conference on programming language design and implementation*. 2018.
[2] Mor, Omri, George Bosilca, and Marc Snir. "Improving the Scaling of 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.

Design Principles

- Multithreaded performance as the first priority.
  - Minimize interference between threads.
  - Use fine-grained try locks.
  - Replace the centralized MPI matching queue with hashtable.
- Versatile Communication Interface.
  - Various communication primitives.
  - Various completion mechanisms.
- Explicit control of communication behaviors and resources.
  - Users have direct access to registered buffers.
  - Communication protocol can be chosen explicitly.
  - Progress function exposed to users.

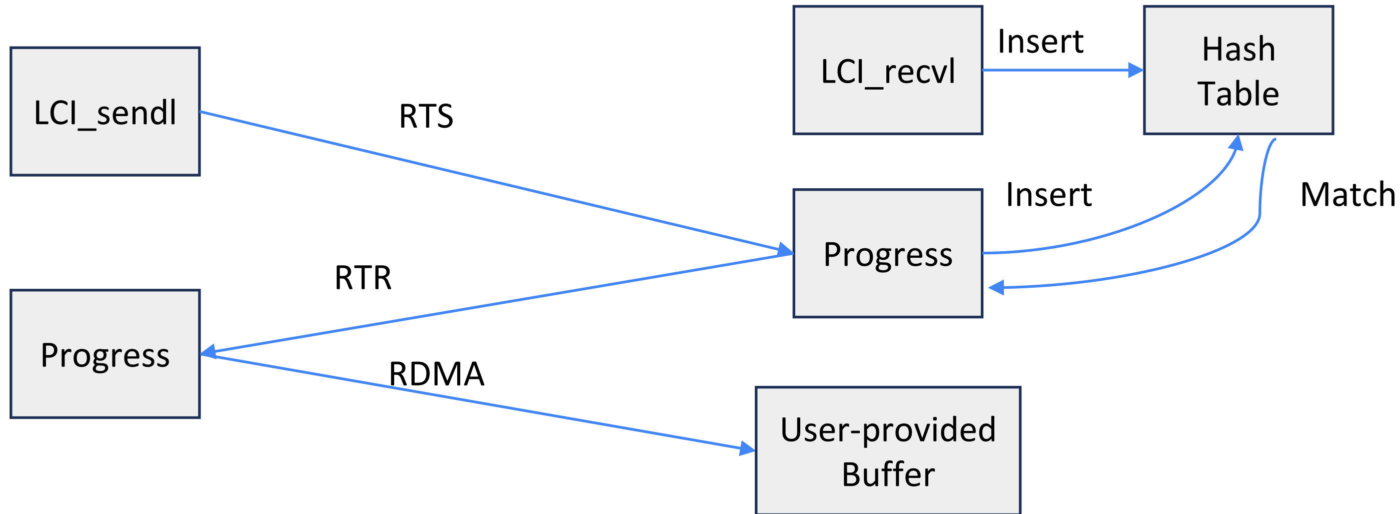
Communication: Eager Protocol

- For LCI_sendm/recvm



Communication: Rendezvous Protocol

- For LCI_sendl/recvl



The LCI Communication backends

Overview

- LCI upper layer
  - Message matching
  - Packet management
  - Rendezvous protocol
  - Various completion notification methods
- LCI communication backend layer
  - Plain message passing without tag matching supports (send/recv).
  - RDMA write with and without signal
  - Completion notification through queue polling.
- The backend layer interface are designed initially with libibverbs' functionality in mind.

Backend Interface

- Initialization/Finalization:
    - LCISD_server_init/fina
    - LCISD_endpoint_init/fina
- Memory registration:
    - LCISD_rma_reg/dereg
- Posting communications (not exhaustive):
    - two-sided
        - LCISD_post_send
        - LCISD_post_recv
    - one-sided
        - LCISD_post_put
        - LCISD_post_putImm
- Completion polling
    - LCISD_poll_cq

Mapping to libibverbs

- Initialization/Finalization:
  - LCISD_server_init/fina ibv_device
  - LCISD_endpoint_init/fina array of ibv_qp, ibv_srq
- Memory registration:
  - LCISD_rma_reg/dereg ibv_reg_mr/ibv_dereg_mr
- Posting communications:
  - two-sided
    - LCISD_post_send ibv_post_send (IBV_WR_SEND_WITH_IMM)
    - LCISD_post_recv ibv_post_srq_recv
  - one-sided
    - LCISD_post_put ibv_post_send (IBV_WR_RDMA_WRITE)
    - LCISD_post_putImm ibv_post_send (IBV_WR_RDMA_WRITE_WITH_IMM)
- Completion polling
  - LCISD_poll_cq ibv_poll_cq

General Scheme

- During initialization, the LCI upper layer
  - Register a large memory buffer.
  - Break it into small buffers (packets).
  - Pre-post thousands of receives.
- Inside the progress engine, the LCI upper layer
  - Check for completed receives (poll_cq) and react accordingly.
  - Re-post receives.

Mapping to UCX

- Initialization/Finalization:
    - LCISD_server_init/fina ucp_context_h
    - LCISD_endpoint_init/fina ucp_worker_h
- Memory registration:
    - LCISD_rma_reg/dereg ucp_mem_map/ucp_mem_unmap
- Posting communications:
    - two-sided
        - LCISD_post_send ucp_tag_send_nbx
        - LCISD_post_recv ucp_tag_recv_nbx
    - one-sided
        - LCISD_post_put ucp_put_nbx
        - LCISD_post_putImm **?**
- Completion polling
    - LCISD_poll_cq **?**

Mimic an RDMA write with signal

- UCP does not provide a “put with signal” primitive similar to `IBV_WR_RDMA_WRITE_WITH_IMM` (in ibverbs) or `fi_writedata` (in libfabric).
- We have to use a sequence of “put + fence + send” to mimic it.
- “fence + send” is called by the callback for ucp_put_nbx

Mimic a completion queue

- All UCP communication operations are posted using UCP_OP_ATTR_FLAG_NO_IMM_CMPL except for ucp_put_nbx
- Pair each LCISI_endpoint (ucp_worker) with a queue
  - Each completion queue entry stores information of a completed operation, including
    - operation type
    - protocol number
    - message size
    - source/destination rank
    - pointer to packet
    - Communication poster
- CQ entries are:
  - Created a cq entry inside LCISD_post_* (function to post communications)
  - Pushed into CQ when related ucp operation is completed (in the function handler)
  - Passed to upper layer (LCII_progress) when polled

A Suspected Bug with UCP

- ucp_put_nbx with UCP_OP_ATTR_FLAG_NO_IMM_CMPL flag seems to have a bug
  - Callback is invoked (local completion is fine)
  - Data is never delivered to remote
- Calling ucp_worker_flush_nbx or ucp_ep_flush_nbx does not help
- Current workaround:
  - If ucp_put_nbx completes immediately, explicitly use callback after
  - In ucp_put_nbx callback, do not push to CQ, instead push it in callback for send signal
  - CQ entry for put (sender side) is not pushed in put callback, but in send callback

Coarse-grained Blocking Locking Issue

- UCP has a more coarse-grained locking scheme than ibverbs.
    - In ibverbs, queue pairs, shared receive queues, completion queues have their own locks.
        - So posting sends, posting receives, and polling cq do not interfere with each other.
        - Typically
            - posting sends are performed in worker threads
            - posting receives and polling cq are performed in progress threads
    - However, UCP uses a single lock to protect a ucp_worker
        - So worker threads and progress threads interfere with each other.
- To make it worse, they are blocking locks.
    - The progress threads (usually of a small number) can be blocked waiting for worker threads for a long time.
- We tried to use a try-lock wrapping all relevant UCP function calls to alleviate this problem.
    - It only works in some cases.

Unmatched Receives when finalizing

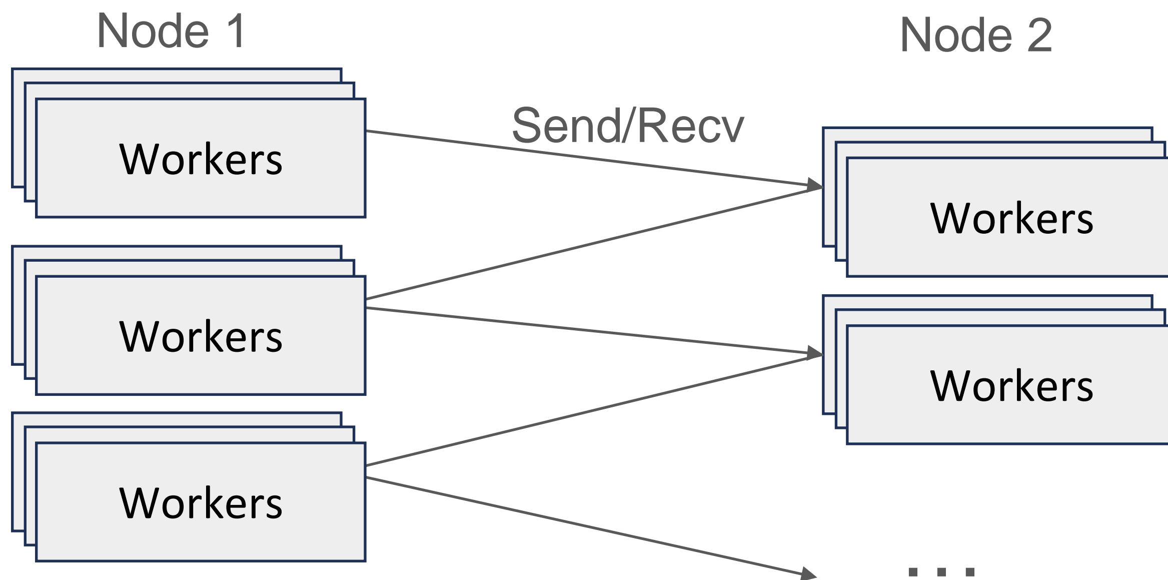
- During finalization, there can still be unmatched pre-posted receive.
- UCX will throw an assertion error if there are unmatched receive
- ibverbs does not require all receive to be matched
- Currently finalization function is blank to circumvent the error
  - Possibly keep track of all posted receive and cancel them during finalization
  - Or allow UCP to automatically cancel unmatched receive in finalization
  - Or allow a configuration option to disable the requirement in finalization

Results

Experiment Setup

- Benchmarks on SDSC Expanse
  - 2× EPYC 7742 64 cores
  - Mellanox ConnectX-6
  - HDR InfiniBand (2x50 Gbps)
- Libraries used
  - Openmpi 4.1.1 with UCX 1.15
  - LCI with ibverbs
  - LCI with UCX 1.15
- One thread per core
  - Always have one dedicated progress thread in multithreaded case

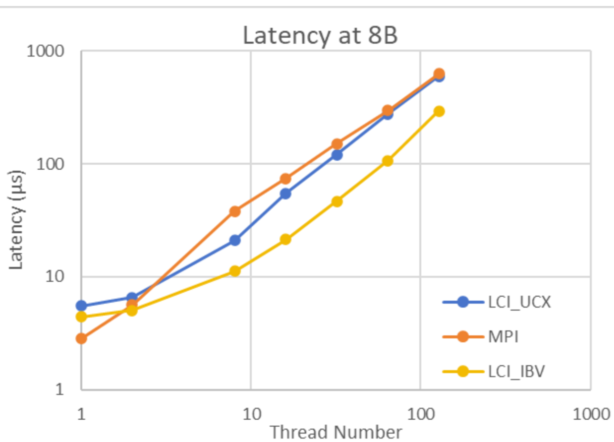
Multithreaded Ping-pong Benchmark

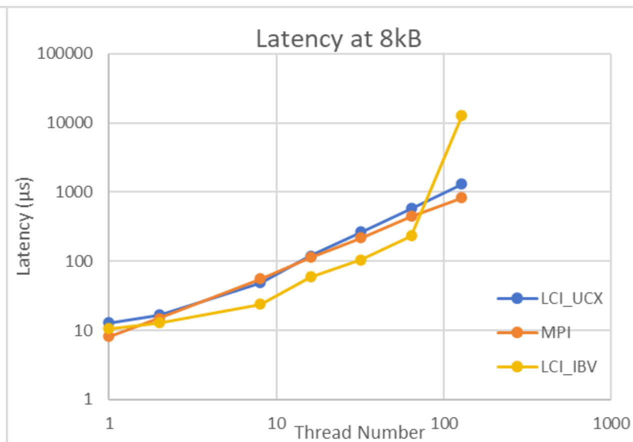


- Small send window (only 1 send/recv per thread), large number of steps (1000)

Results: Latency vs. Thread Number

- Latency as a function of thread number
    - Thread number: 1, 2, 4, 8, 16, 32, 64, 128
- Libraries compared:
    - LCI with UCX backend (without try-lock)
    - LCI with ibverbs backend
    - MPI with UCX backend





Latency at 256kB

Latency (µs)

Thread Number

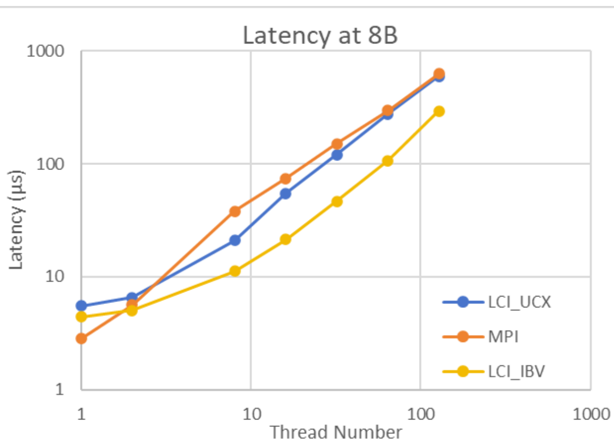
10000 1000 100 10 1

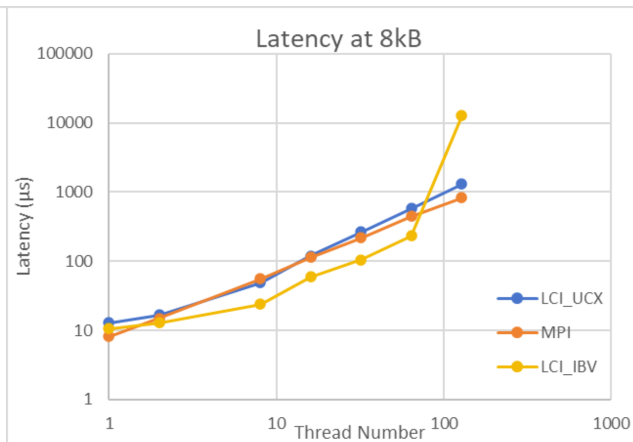
1 10 100 1000

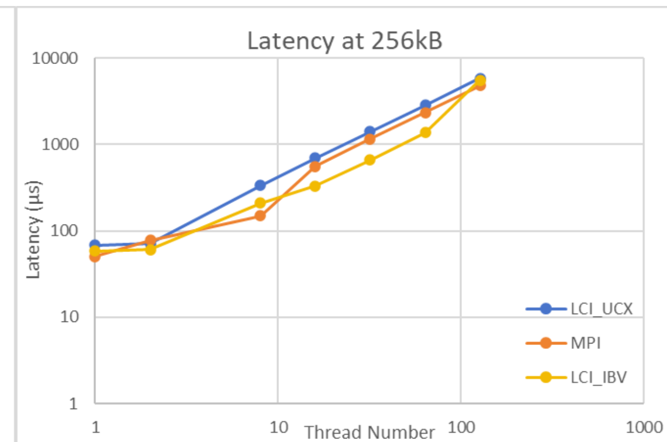
LCI_UCX MPI LCI_IBV

Discussion of Results

- LCI with ibverbs outperforms LCI with UCX and MPI with UCX
  - IB uses finer-grained locks
  - UCP progress, send/recv share the same lock

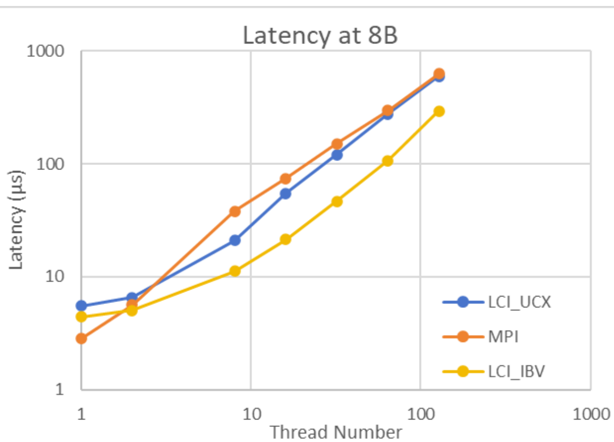


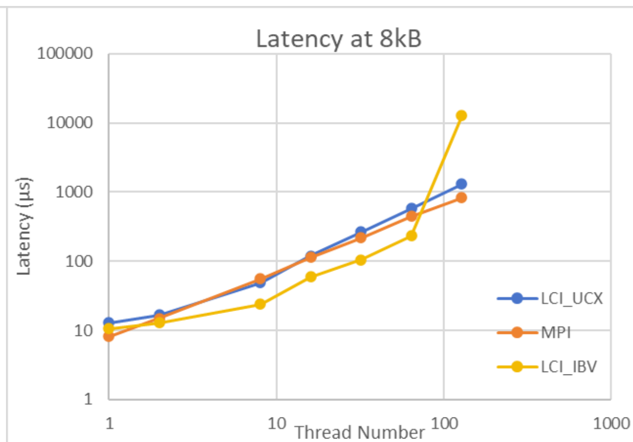


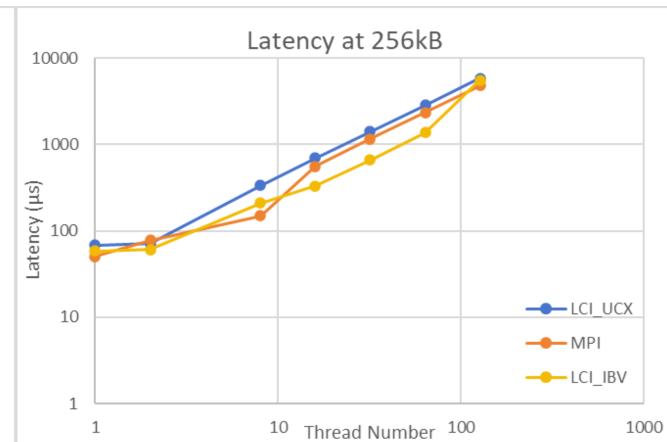


Discussion of Results

- Latency jump of LCI with ibverbs at 128 threads, 8kB message
  - Threads are spread in 2 different sockets
  - Only 1 progress thread, half of workers will go across sockets
  - need to experiment with numactl --interleave

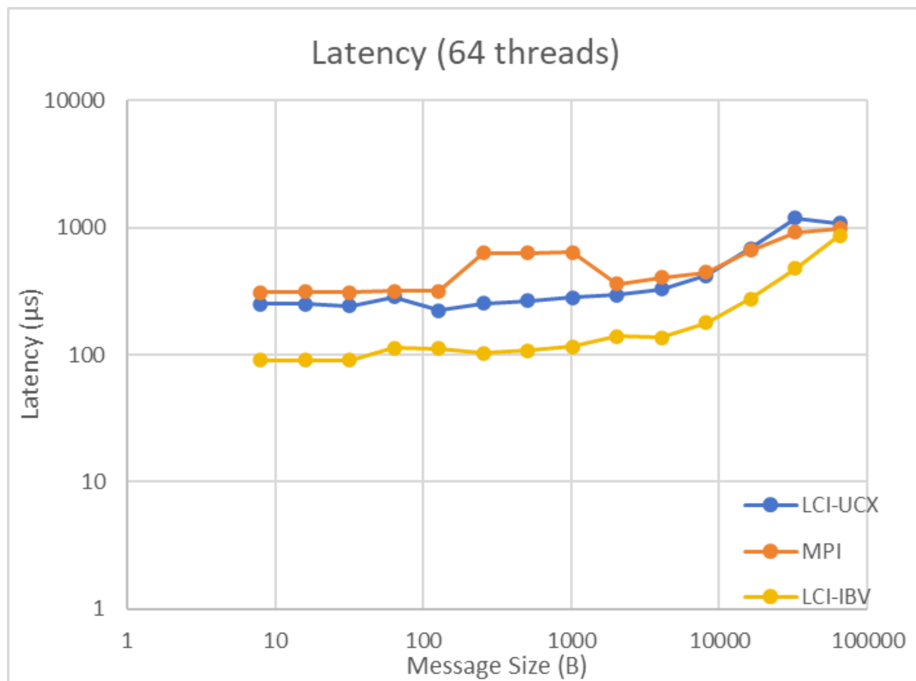






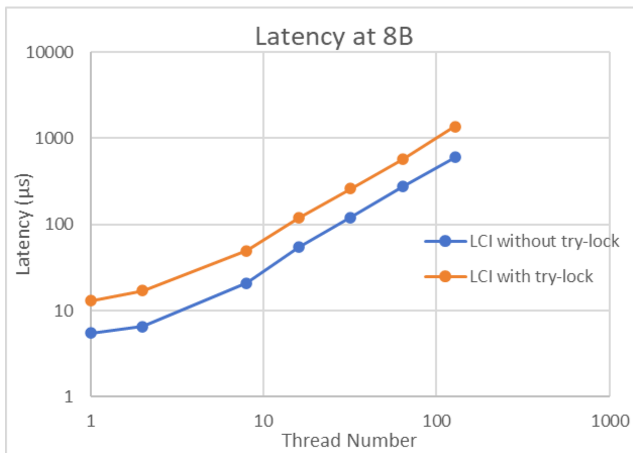
Latency vs. Thread Number and Latency vs. Message Size

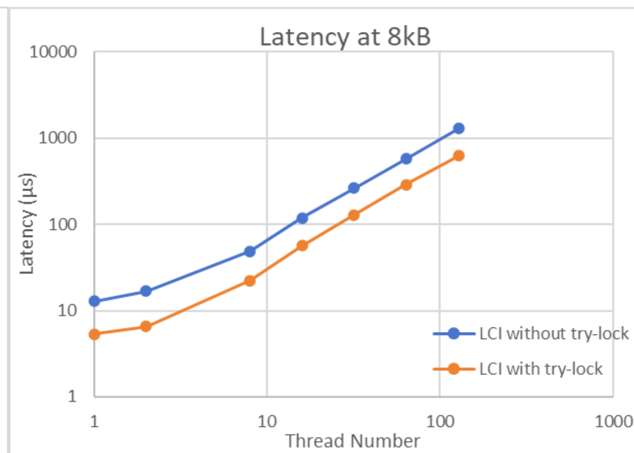
- As message size gets larger, performance gets similar
  - Hardware bandwidth becomes the bottleneck

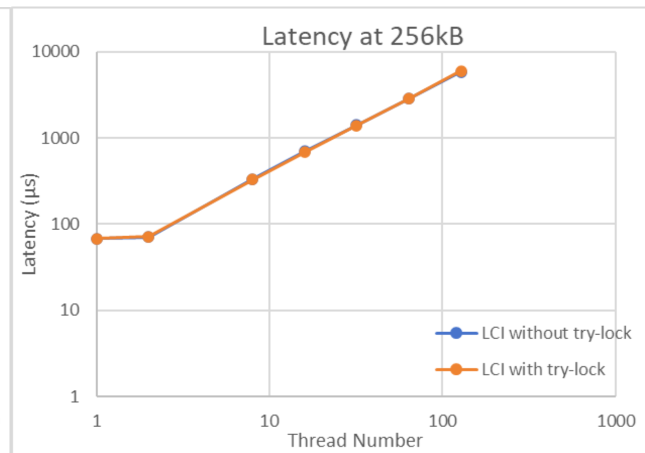


Results: With vs. Without Try-lock

- LCI with UCX backend, with and without try-lock wrapping around all UCP functions
- When unable to obtain the lock, directly return with LCI_ERR_RETRY

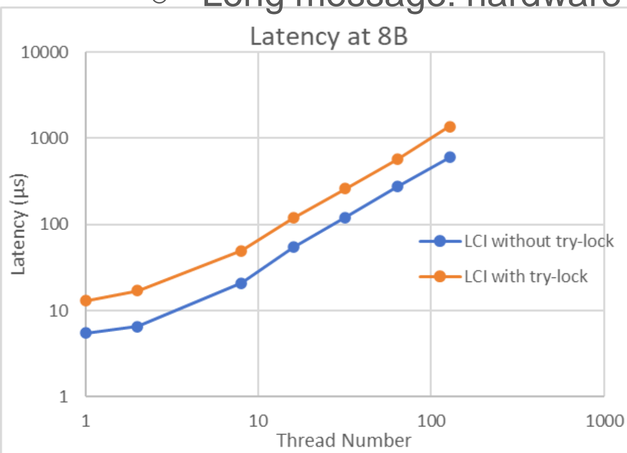


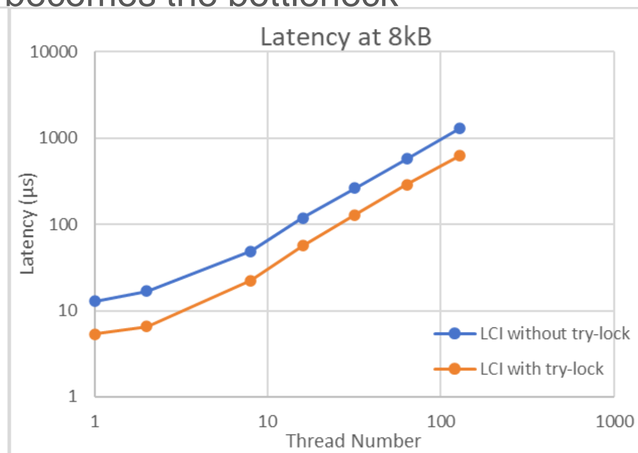


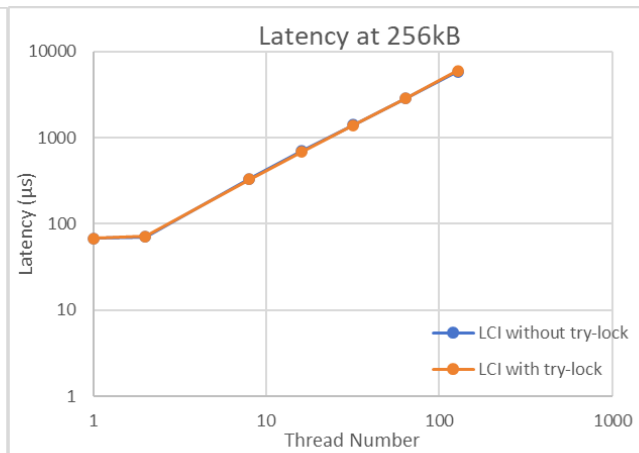


Discussion of Results

- Degrades performance for short messages, Improves performance for medium messages, no change for long messages
- Hypothesis: try-lock makes progress function obtains the lock more often
  - Small message: progress function has little task
  - Medium message: progress function has more work
  - Long message: hardware becomes the bottleneck







Future Work

- Optimize current implementation
- Test with multiple progress threads/devices
- Implement and test the configuration without dedicated progress thread
- Implement the backend using lower-level API: UCT
- Implement LCI API using UCP directly and compare performance

Questions?