UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

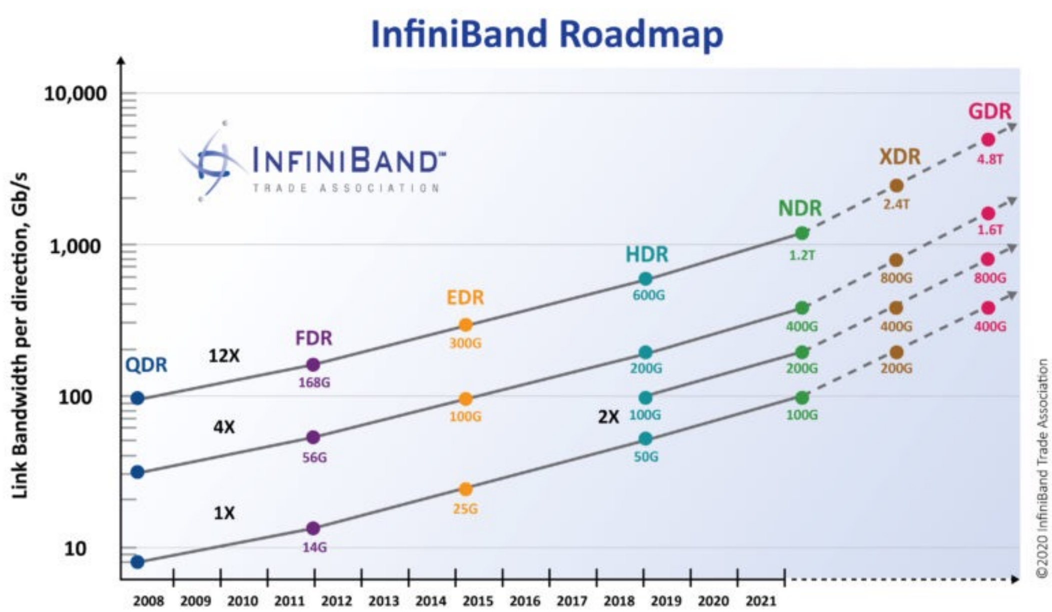
MPI One-Sided & Asynchronous Task Models

Marc Snir

Huang Vu-Dang, Omri Mor, Jiakun Yan

illinois.edu

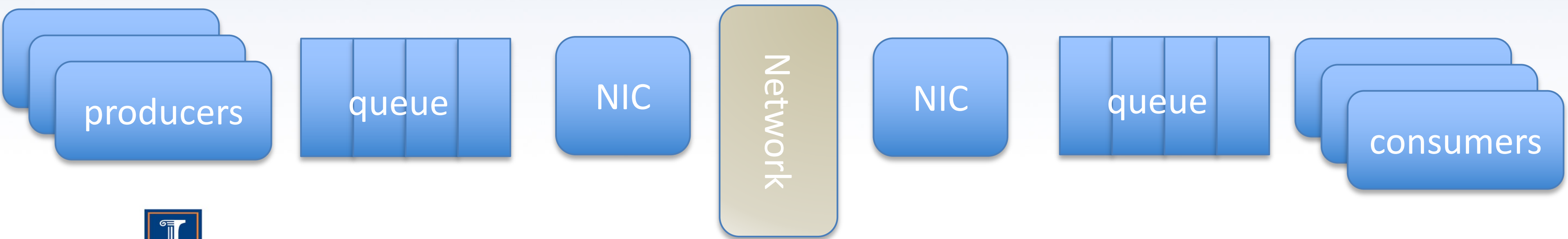
Expected NIC Hardware Evolution



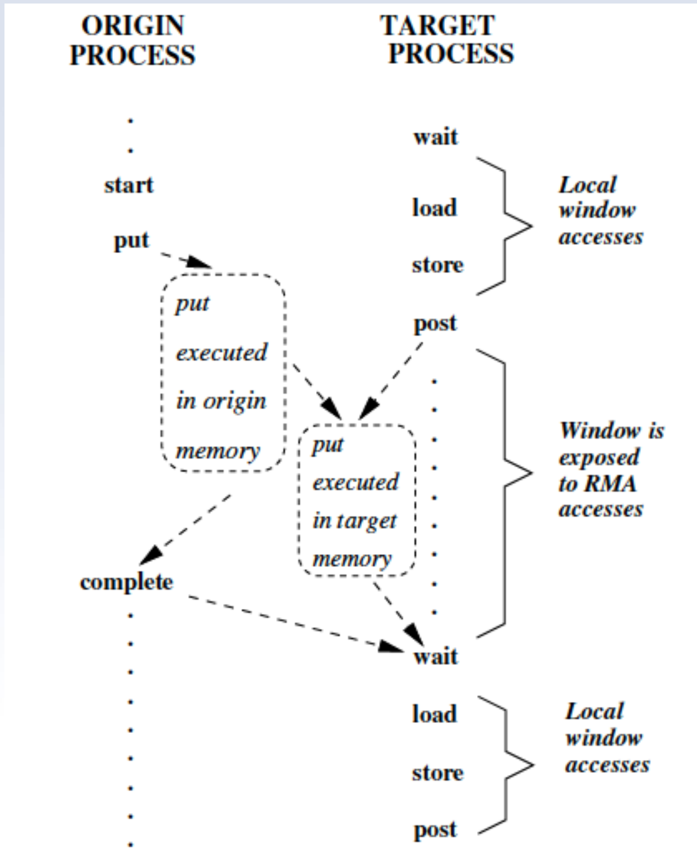
- Terabits of data
- Message rate > 1B/sec
- Faster NIC-CPU connection
- Hardware acceleration of the handling of incoming messages is essential
- Parallelism in message-handling is essential
- NIC can touch system memory with acceptable overhead

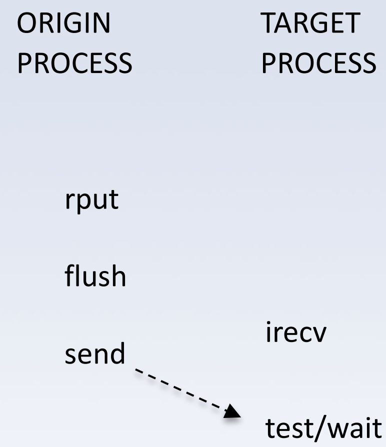
Lifetime of a message (schematic)

- Key bottleneck: Demultiplexing (matching table)
  - moving each incoming message to the right place
  - signaling consumer of message arival
- Put handles first issue (message carries destination buffer address)
- Put does not handle second issue



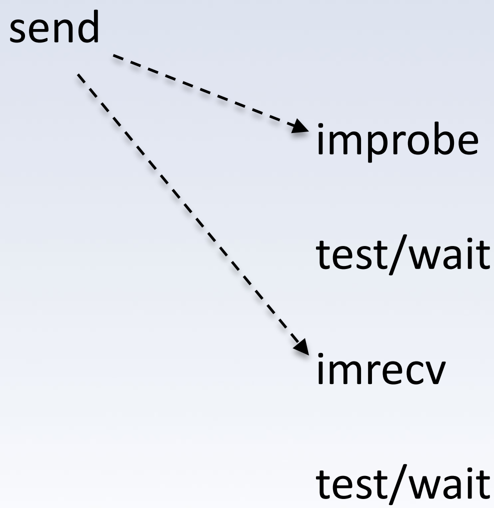
How is target process informed that a remote access is complete?





- Source process ensures completion at target, then uses 2-sided communication to inform target of completion
  - Not truly one-sided
  - Not faster than "fake" one-sided using message-passing

"Fake" One-sided communication



- 4 MPI calls at target
- Twice polling or waiting at target
- Rendezvous under the cover

One-sided communication using active-messages (GASNet)

put(data, handle)

Invoke handler

- Very general
- Hard to accelerate

- Can we accelerate specific, frequently used handlers?

- In particular, can we accelerate signaling message arrival?

Remote signaling of RDMA Completion

- Target NIC "knows" when RDMA is complete at target
  - E.g., Infiniband can enqueue entry to completion queue at target
  - Needed for error recovery
- MPI does not seem to take advantage of this

Asynchronous Task Model

- Hybrid dataflow computation model: Graph of
  - Nodes: Sequential tasks
  - Edges: Dependencies
  - Task is executed when all the dependencies are satisfied.
- Graph and mapping of tasks to nodes is statically defined or expanded dynamically ahead of its evaluation.
  - PaRSEC, HPX, Legion…
- Graph analysis libraries have similar logic
- Communication is not symmetric (not ping-pong) and can be very irregular
- Number of pending communications can be very large

Communication Requirements

1. Instantiating and mapping dataflow graph
2. Communicating data for internode dependencies

- One-sided is good match for 2nd requirement
- But need to know when all dependencies are satisfied

- No need to wake-up a task when its dependencies are satisfied. One only need to mark the task as runnable
  - The scheduler is invoked by a thread when it becomes idle
  - The scheduler picks a runnable task for execution and assign it to the thread for execution
- Signaling RDMA Completion =(possibly) marking a task as runnable – **The consumer of the signal is the scheduler**

Marking a Task as Runnable Current Practice

- A master thread polls for incoming message and keeps track of satisfied dependencies (send-receive)
- When a task is runnable it is moved to runnable queue
- An idle thread communicates with the master thread to get work

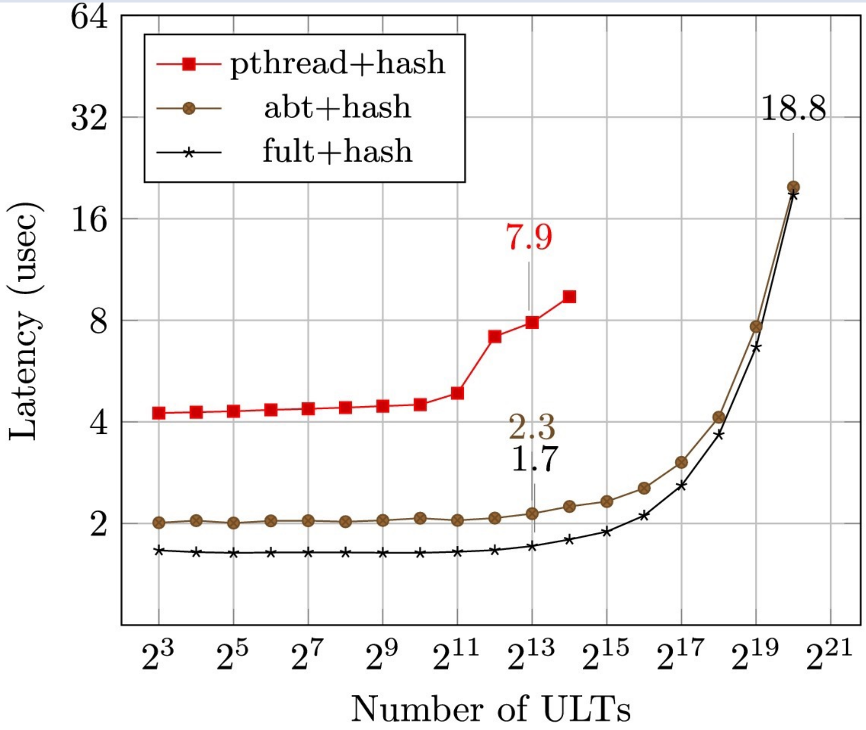
- MPI polls IB queue; Master thread polls requests

Marking a task as runnable

- Simple most case: Message arrival is signaled by decrementing a counter; task is runnable if counter is zero.
- Basic data structure: vector of (short) counters
  - Scalability achieved using vector of vectors of counters.
- One queue per thread, with work stealing for load balancing
- Three main operations:
  - Set counter (done by the runtime when task is created)
  - Decrement counter (done by the communication library when a message arrives)
  - Find a zero counter (done by the scheduler to schedule a runnable task)
    - Accelerated using count leading zero (vector) hardware

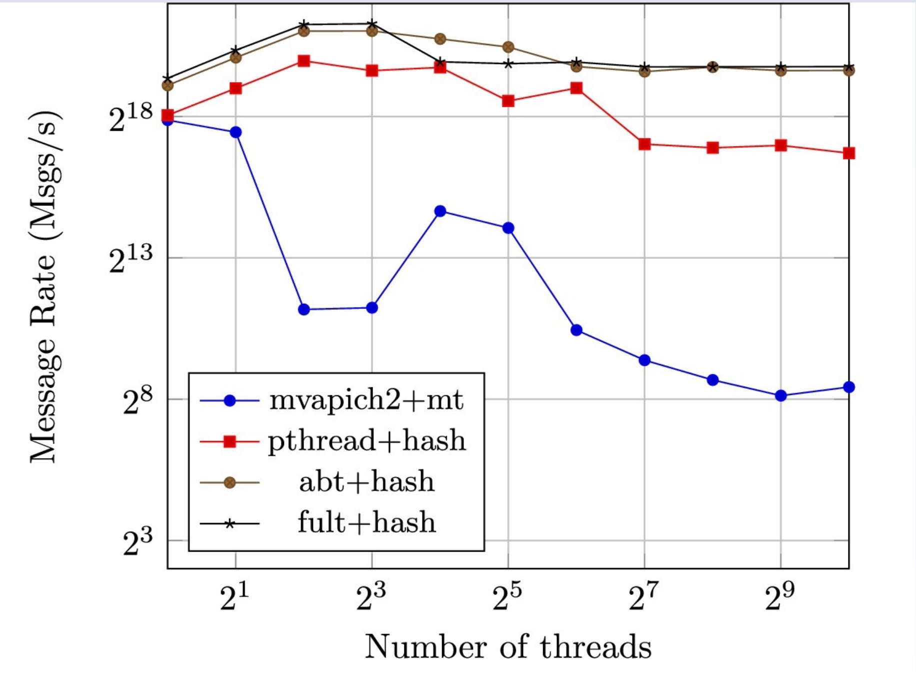
- HV **Dang**, M **Snir** -Fult: Fast user-level thread scheduling using bit-vectors

Some (old) results – Latency (blocking send/recv pingpong)



- ~2018

Message rate (64 byte)



- MPI without ANY_SOURCE, ANY_TAG
  – no central queues needed

- Unlike MPI, ANY_SOURCE is supported, but indicated by the send

Marking a task as runnable (2)

- More conventional design: Message arrival is signaled by decrementing a counter; if the counter is zero then an entry is appended to a queue
- Basic data structure: linked list
- Basic operation done by communication library:

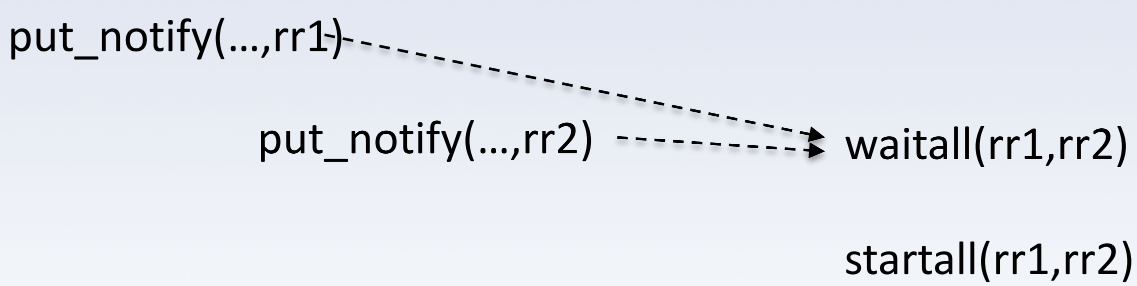
```
decrement counter;
 if (counter == 0)
    atomic exchange of two pointers
```

  – Easy to move to NIC (?)

Can this be retrofitted into MPI?

- First thought

```
put_notify(…, remote_request)
```



- If communication pattern repeats then need two copies of each persistent request in order to avoid races

MPI Fundamental Issue with Blocking Calls

- MPI "knows" that a MPI_Wait is satisfied; the scheduler does not know that
  - The scheduler does not know that a blocked task has become runnable
  - Significant performance issue when there is a large number of tasks blocked on MPI communications
- An event that completes a blocking call should communicate to the scheduler to mark the blocked task as runnable
  - Or manipulate by itself the scheduler data structures

Can this be retrofitted into MPI?

- Second thought

```
put_notify(…, remote_semaphore)
```

sem.P() executed by the task that needs the data or code that generates the task

put_notify(…,sem) ⇢ MPI executes sem.V() upon message arrival

- **Scheduler is aware of semaphore logic**
- May prefer a "restricted" binary semaphore, were P and V calls always alternate, in order to enable a more efficient implementation

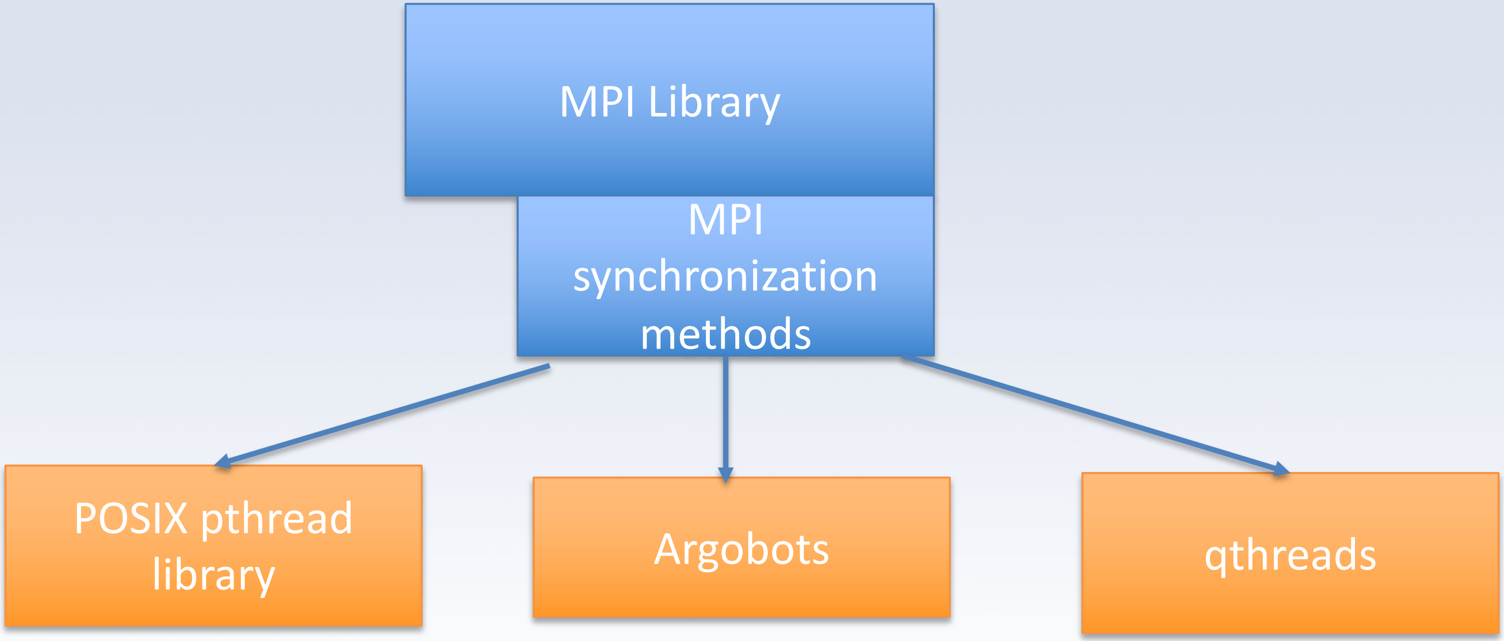
Counting Depdences: Need variant of counting semaphore

- `sem.P(count):` sets the semaphore to `count` and blocks until `count == 0`
- `sem.V():` decrements the counter by one
- Restricted use: a `sem.P(count)` is succeeded by `count sem.V()` operations

Same approach works with other communications

- For example
  - `recv(…,sem)`
- Caller blocks. When receive is satisfied, MPI executes the prescribed synchronization operation
- This is a specialized active message handler, that can be implemented in software, or could be accelerated by NIC

Packaging



MPI Library
MPI synchronization methods
POSIX pthread library
Argobots
qthreads

Summary

- MPI was designed as a pure communication library
- End-to-end performance is affected by awkward interfacing to other runtime components.
  - Scheduler (CPU manager)
  - Memory manager
  - Energy manager ??
- In order to reduce the effective communication latency, one needs to consider not only buffer to buffer delay, but also producer to consumer signaling delay.
  - And running benchmarks where CPUs are not only busy communicating
- **MPI needs to interact with (CPU/GPU) scheduler, in order to avoid polling**

The End