Short Messages

• For short messages
  ▪ `memcpy()` into / out of pre-registered buffers
  ▪ “short” = “memcpy cost does not matter”

• Once copied in, do one of two things:
  ▪ Use “eager” RDMA
  ▪ Use send / receive semantics
Short Message Protocol

- **MPI_SEND(short_message, ...)**

User buffer

Pre-registered buffer queue
Short Message Protocol

- \texttt{MPI\_SEND(\textit{short\_message}, \ldots)}
Short Message Protocol

- \texttt{MPI\_SEND(short\_message, ...)}
Short Message Protocol

• `MPI_SEND(short_message, ...)`

![Diagram showing User buffer, User buffer copy, Pre-registered buffer queue, and Send / RDMA]
Short Message Protocol

- \texttt{MPI\_SEND(\textit{short\_message}, ...)}

- User buffer
- User buffer copy
- Pre-registered buffer queue

Return to queue when send completes
Short: RDMA vs. Send

- RDMA semantics
  - Sender specifies [remote] target buffer address
  - Requires N pre-registered buffers for each peer
  - Quickly becomes non-scalable

- Send / receive semantics
  - Receiver specifies target buffer address
  - Can use common pool of pre-registered buffers
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer:

- **Peer A**
  - Msg 1
  - Msg 2

- **Peer B**
  - Msg 1
  - Msg 2
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer A

Peer A

Msg 2

Msg 3

Msg 4

Buffers for peer B

Peer B

Msg 1

Msg 2

Buffers

Peer A

Msg 1

Msg 2

Buffers

Peer B

Msg 1

Msg 2
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer A and peer B with messages exchanged between Peer A and Peer B.
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer A

Peers:
- Peer A
- Peer B

Messages:
- Msg 1
- Msg 2
- Msg 3
- Msg 4
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer A

Peer A

- Msg 2
- Msg 3
- Msg 4

Buffers for peer B

Peer B

- Msg 1
- Msg 2

Peer A

- Msg 1
- Msg 2
- Msg 3
- Msg 4

Peer B

- Msg 2
Short: RDMA vs. Send

- **RDMA**
  - Requires initial setup (exchange addresses)
  - Completely hardware driven

- **Send / receive**
  - Less initial setup
  - Driver picks buffer
  - Involves remote software

Buffers for peer A and B:
- Peer A: Msg 2, Msg 3, Msg 4
- Peer B: Msg 1, Msg 2

Or it might RNR!
RDMA vs. Send

- RDMA
  - Buffers for each peer: must be (MxN)
  - Exchange addresses
  - MPI maintains accounting/flow control
  - MPI must notice new received messages
  - Unordered
  - "One-sided"
  - More work for MPI

- Send / receive
  - Pool of buffers -- can be less than (MxN)
  - Network maintains accounting / [some] flow control
  - Network notifies of new received messages
  - Ordered
  - Two-sided (ACK’ed)
  - Less work for MPI
Limiting Short RDMA

• Open MPI allows N “short” RDMA peers
  ▪ (N+1)th peer will use send/receive for short
  ▪ Receiver’s choice
• If M buffers posted for each RDMA peer
  ▪ Total registered memory for short RDMA
    \[ M \times N \times \text{short}\_size \]
[Shared] Receive Queue

- How to receive messages?
- Per-peer resources
  - Flow control
  - Never error
- Pooled resources
  - No flow control
  - Better utilization
  - Can play stats game
    - Potential for retransmits

Per-peer receive queues

For peer 1:

For peer 2:

For peer 3:

For peer 4:

Shared receive queue
[Shared] Receive Queue

- How to receive messages?
- Per-peer resources
  - Flow control
  - Never error
- Pooled resources
  - No flow control
  - Better utilization
  - Can play stats game
    - Potential for retransmits

Per-peer receive queues
- For peer 1:
- For peer 2:
- For peer 3:
- For peer 4:

Shared receive queue

Less than NxM buffers
Long Messages

- For long messages
  - Pipelined protocol
  - Fragment the message, event-driven queue
    - Register
    - Send / receive
    - Unregister
  - (...skipping many details...)
- More complicated if message is not contiguous (not described here)
Long Message Protocol

- **MPI_SEND**(long_message, ...)

  Long message

- Sent in 3 phases:
  - Eager / match data
  - Send / receive data
  - RDMA data
Long Message Protocol

- **MPI_SEND**(long_message, …)

- 1st phase: Find receiver match
  - Use (copy + send/receive) for first fragment
  - Only send enough to confirm receiver match; do not overwhelm receiver resources
  - Typical rendezvous protocol
Long Message Protocol

• **MPI_SEND**(long_message, ...)  
  
  1 2

• 2nd phase: Hide receiver register latency
  - Use (copy + send/receive) for next few frags
  - Allows overlap of memory registration

• Pipeline subject to max depth setting
  - Conserve resources
Long Message Protocol

- **MPI_SEND**(long_message, ...)

  1. **Eager send** + send until RDMA offset
  2. Send(s) complete
  3. Unregister

  Time

  - Register + Write
  - Unregister
  - Unregister + Complete

  1 2 3
Why So Complex?

- Why not a single register / send / deregister?
  - Each time is directly proportional to buffer length
- Total time is longer
- Many details skipped
  - See paper on [www.open-mpi.org](http://www.open-mpi.org)
fork() Support

- fork() and registered memory problematic
  - Must differentiate between parent / child registered memory
- Not properly supported until:
  - OFED v1.2
  - Open MPI v1.2.1
  - Kernel 2.6.16 or later (but some Linux distros have backported, e.g., RHEL4U6)
- Query MCA param:
  - btl_openib_have_fork_support
Striping

- Automatic:
  - Short messages round robin across lowest latency BTL module
  - Long message fragments round robin across all available BTL modules
  - Size of long message fragments proportional to network capacity
- Developers investigating more complex scheduling scenarios
How to Activate Striping?

• Do nothing (!)

  mpirun -np 4 a.out

• If Open MPI detects multiple active OF ports, it will use them all
  - Assuming each peer port is “reachable”
  - Determined by subnet ID
Warning: Default Subnet ID

- Subnet ID uniquely identifies an IB network
  - All Cisco IB switches ship with the default ID
    - FE:80:00:00...
  - To compute peer process reachability, Open MPI must have different subnet IDs
    - But still, ambiguities exist
Warning: Default Subnet ID

Good

Bad -- OMPI can’t compute reachability

Bad -- OMPI can’t compute reachability, but will warn
More Information

- Open MPI FAQ
  - General tuning
    http://www.open-mpi.org/faq/?category=tuning
  - InfiniBand / OpenFabrics tuning
    http://www.open-mpi.org/faq/?category=openfabrics