CSE 5194

Implementation of Programming Models and Environments on Modern Systems

Rajarshi Biswas
Papers:

- **High Performance RDMA-Based MPI Implementation over InfiniBand**
  - J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda
  - Int'l Conference on Supercomputing (ICS '03), June 2003.

- **MVAPICH-Aptus: Scalable High-Performance Multi-Transport MPI over InfiniBand**
  - M. Koop, T. Jones and D. K. Panda
  - Int'l Parallel and Distributed Processing Symposium (IPDPS), 2008

- **Efficient and Truly Passive MPI-3 RMA Using InfiniBand Atomics**
  - M. Li, S. Potluri, K. Hamidouche, J. Jose and D. K. Panda
  - EuroMPI 2013, September 2013
High Performance RDMA-Based MPI Implementation over InfiniBand

J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda
Motivation

• MPI has been the de facto standard for writing parallel applications.
• Existing designs of MPI over VIA [and InfiniBand] use send/receive operations for small data messages and control message and RDMA operations for large data messages.
• Complexity in hardware implementation and non-transparency to the remote side, send/receive operations do not perform as well as RDMA operations in current InfiniBand platforms.

• This paper we propose a method which brings the benefit of RDMA operations to not only large messages, but also small and control messages.
InfiniBand Channel and Memory Semantics

• Channel Semantics
  • send/receive operations are used for communication.
    • Sender side, the programmer initiates the send operation by posting a send descriptor
    • Receiver side, the programmer posts a receive descriptor which describes where the message should be put at the receiver side
    • Multiple send and receive descriptors can be posted and they are consumed in FIFO order.
      • Eager Send

• Memory Semantics
  • one-sided and do not incur software overhead at the other side
    • RDMA Read
    • RDMA Write
      • immediate data Variabteion
    • Faster and less overhead

Figure 1: Latency of Send/Receive and RDMA
Mapping MPI protocols

Communication Modes

• standard, Synchronous, Buffered, and Ready modes

• Eager protocol (Send/Receive Based Approach)
  • Message is pushed to the receiver side regardless of its state.
  • Small message

• Rendezvous
  • A handshake happens between the sender and the receiver via control messages before the data is sent to the receiver side
  • Large message
  • Ideal for zero-copy transfers

Figure 2: MPI Eager and Rendezvous Protocols
Mapping MPI protocols

- **Send/Receive Based Approach**
  - Eager protocol messages and control messages are transferred using send/receive operations.
  - A reliable connection is set up between every two processes.
  - Buffer pinning and unpinning overhead is avoided by using a pool of pre-pinned.
  - A credit based flow control mechanism.

- **RDMA-Based Approach**
  - RDMA write based approach for Eager protocol and control messages.
  - RDMA destination address must be known.
  - Receiver side must detect the arrival of incoming messages.
  - Persistent buffer association
    - Reduce the number of RDMA operations.
    - Persistent correspondence between each buffer at the sender/receiver.
    - Reduce the overhead of building RDMA descriptors.
  - Receiver always knows exactly where to expect the next message.
  - Polling time for multiple connection may increase.

- **Hybrid Approach**
  - RDMA write and send/receive
  - Process only communicates with a subset of all other processes.
  - RDMA polling set at the receiver side.
Design

• Basic Structure of an RDMA Channel
  • RDMA channels are unidirectional.
  • Set of fixed size, pre-registered buffers at both the sender side and the receiver side.
  • Buffers are organized in a Ring
  • Sender side, the head pointer is where the next out-going message should be copied
  • Tail pointer at the sender side is to record those buffers that are already processed at the receiver side
  • Receiver side, the head pointer is where the next incoming message should go
  • Tail pointer advances if and only if the current buffer is ready for reuse
  • Memory usage

• Polling for a Single Connection
  • sender side
    • Set data size.
    • Check the position of the tail flag. If the value is the same as the primary flag value, use the secondary value. Otherwise, use the primary value.
    • Set the head and tail flags.
    • Use RDMA write operation to transfer the buffer.
  • Receiver polls by performing the following:
    • Check to see if the head flag is set. Return if not.
    • Read the size and calculate the position of the tail flag. 3. Poll on the tail flag until it is equal to the head flag.
    • After processing, the receive side clears the head flag.
Design

• **Reducing Sender Side Overhead**
  • Before buffers can be sent out, descriptors must be allocated and all the fields must be filled.
  • After the operations are done, completion entries are generated for them in the CQ and the sender side must process them.
  • Store descriptors with the buffers.
  • Unsignalled operations in InfiniBand.

• **Flow Control for RDMA Channels**

• **Ensuring Message Order**
  • RDMA channel and the send/receive channel.
  • Receiver has to poll on both channels to receive messages.
    • Packet Sequence Number Expected Sequence Number

• **Polling Set Management**
  • Put first N (N is the size of the RDMA polling set) channels with incoming messages into the RDMA polling set.
  • Dynamically manage the polling set for large number of connections.
Performance Evaluation

- **Experimental setup**
- **Cluster system**
  - consisting of 8 SuperMicro SUPER P4DL6 nodes.
  - Each node has dual Intel Xeon 2.40 GHz processors with a 512K L2 cache at a 400 MHz front side bus.
  - Connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch.
  - The HCA adapters work under the PCI-X
Performance Evaluation

- Achieved a 6.8 microseconds latency for small messages
- The peak bandwidth is around 871 Million Bytes (831 Mega Bytes)/second
Performance Evaluation

- Improvements of our RDMA-based design by comparing it with the send/receive based design
- Collective Communication
Performance Evaluation

Results for IS, MG, LU, CG, FT, SP and BT programs from the NAS Parallel Bench-mark Suite on 4 and 8 nodes.
Conclusion

• Brings the benefit of RDMA to not only large messages, but also small and control messages

• Better scalability by exploiting application communication pattern and combining send/receive operations with RDMA operations
MVAPICH-Aptus

Scalable High-Performance Multi-Transport MPI over InfiniBand

M. Koop, T. Jones and D. K. Panda
Problem Statement

• This work seeks to address two main questions:

• What are the different protocols developed for MPI over InfiniBand and how do they perform at scale?

• Given this knowledge, can the MPI library be designed to dynamically select protocols to optimize for performance and scalability?
Message Channels

• Message passing is generally implemented with two modes:
  • Eager Protocol: Small messages (<8K)
  • Rendezvous Protocol: Large messages

• Multiple designs of both protocols have been implemented for InfiniBand
  • Describe and evaluate each of them to determine performance and scalability characteristics
Eager Channels

• Reliable Connection Send/Receive (RC-SR)
  • Channel built directly on the channel semantics of the RC transport of InfiniBand
  • Use of the Shared Receive Queue (SRQ) allows pooling of receive buffers to achieve better scalability

• Reliable Connection Fast Path (RC-FP)
  • Current adapters only reach their lowest latency using RDMA Write operations
  • This approach uses paired queues and last-byte polling to achieve low latency (at the cost of memory usage)

• Unreliable Datagram Send/Receive (UD-SR)
  • Built on the channel semantics of the UD transport of InfiniBand
  • Must take care of reliability, however, it is very scalable
Rendezvous Channels

• Reliable Connection RDMA (RC-RDMA)
  • Using this method an RDMA write operation is used to write directly into the application buffer without intermediate copy operations

• Unreliable Datagram Zero-Copy (UD-ZCopy)
  • Using a pool of QPs and a novel approach, data can be transferred over UD -- preventing the requirement that RC connections be created

• Copy-Based Send
  • Negotiate buffer availability, but then use the eager channels to push the data to the receiver
Performance: Eager Latency

- Classic ping-pong latency test (osu_latency)
- RC-FP delivers lowest latency
- RC-SR and UD-SR perform similarly until 2K and beyond where UD-SR requires software packetization
Throughput for RC-based channels performs poorly when the number of communicating pairs increases.

UD-SR remains scalable in performance.
Scalability: Memory Usage

- **RC-FP** requires a significant amount of memory resources
- **RC-SR** is much more scalable in memory, but can still have issues at scale
- **UD-SR** remains very scalable with near-
- constant memory usage
Scalability: Latency

- Due to the memory polling used in RC-FP only a few channels can be allocated before latency increases.
- The InfiniBand HCA has only a limited number of QPs that can be active in the on-card cache.
## Summary

<table>
<thead>
<tr>
<th>Type</th>
<th>Channel</th>
<th>Transport</th>
<th>Latency</th>
<th>Throughput</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eager</td>
<td>RC-SR</td>
<td>RC</td>
<td>Good</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td></td>
<td>RC-FP</td>
<td>RC</td>
<td>Best*</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td></td>
<td>UD-SR</td>
<td>UD</td>
<td>&lt;2K, Good</td>
<td>&lt; 2K, Best</td>
<td>Best</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&gt;2K, Poor</td>
<td>&gt; 2K, Poor</td>
<td></td>
</tr>
<tr>
<td>Rendezvous</td>
<td>RC-RDMA</td>
<td>RC</td>
<td>-</td>
<td>Best</td>
<td>Fair</td>
</tr>
<tr>
<td></td>
<td>UD-ZCopy</td>
<td>UD</td>
<td>-</td>
<td>Good</td>
<td>Best</td>
</tr>
<tr>
<td></td>
<td>Copy-Based</td>
<td>RC/UD</td>
<td>-</td>
<td>Poor</td>
<td></td>
</tr>
</tbody>
</table>

No eager or rendezvous channel has all of the desired features.
MVAPICH-Aptus Design

• As seen from the previous evaluation results, no single channel for either eager or rendezvous is always best

• General Goal:
  • Use a combination of message channels and transports to optimize for performance and scalability

• Design Challenges:
  • When should a channel be created? ▪ When should a channel be used?
Channel Allocation

• Some channels perform well when only a limited number of them are created, but quickly deteriorate
  • RC Transports (RC-SR/RC-FP/RC-RDMA)
    • Each RC connection requires additional memory usage
    • Cache on HCA can be overflowed quickly
  • RC-FP
    • Too many channels increases polling time
    • Memory scalability is poor

• Strategy:
  • Create up to a configurable number of channels of each type ▪
    • 16 RC QPs
    • 8 RC-FP connections
  • Setup after a certain number of “qualified” messages are transferred
Channel Usage

• As found earlier, some channels also perform differently given message size and other features

• We allow a flexible form of matching when sending a message:

• Take the first match where both the conditional is true and the channel is allocated to the destination peer

Sample Configuration

<table>
<thead>
<tr>
<th>MSG_SIZE</th>
<th>Channel Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 2048</td>
<td>RC-FP</td>
</tr>
<tr>
<td>&lt;= 2008</td>
<td>UD-SR</td>
</tr>
<tr>
<td>&lt;= 8192</td>
<td>RC-SR</td>
</tr>
<tr>
<td>&lt;= 8192</td>
<td>UD-SR</td>
</tr>
<tr>
<td>TRUE</td>
<td>RC-RDMA</td>
</tr>
<tr>
<td>TRUE</td>
<td>UD-ZCopy</td>
</tr>
<tr>
<td>TRUE</td>
<td>Copy-Based</td>
</tr>
</tbody>
</table>
Evaluation

• MVAPICH

• Evaluated Configurations:

<table>
<thead>
<tr>
<th></th>
<th>RC-SR</th>
<th>RC-RDMA</th>
<th>RC-FP</th>
<th>UD-SR</th>
<th>UD-ZCopy</th>
</tr>
</thead>
<tbody>
<tr>
<td>RC</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVAPICH0.9.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UD</td>
<td></td>
<td></td>
<td>Available</td>
<td>Available</td>
<td></td>
</tr>
<tr>
<td>MVAPICH-UD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aptus</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td>Available</td>
<td></td>
</tr>
</tbody>
</table>

Experimental Testbed:
- 70 node, 560-core InfiniBand Linux cluster
- Dual 2.3GHz “Clovertown” quad-core processors
- Mellanox MT25208 DDR HCA
- OpenFabrics OFED 1.2
- We evaluate the following application benchmarks
  - NAS Parallel Benchmarks: CFD application kernels
  - NAMD: Molecular dynamics application
  - SMG2000: Multigrid solver (ASC Benchmark)
- In addition to collecting the wallclock performance measurement, we also evaluate other characteristics:
  - Channels created
  - Message and data volume over each channel
In all results we see that the hybrid UD/RC design is able to outperform or match either mode used exclusively.

- 512/484 processes

Breakdown shows Aptus dynamically has setup the fewest channels needed.
Breakdown of message transfers by channel show *good utilization* of “expensive” channels, despite allocating only a few of them.
Conclusions

• As clusters continue to scale, the MPI library must be scalable in memory as well as performance

• Previously a UD-based MPI showed superior scalability, but lower performance in some applications

• In this work we bridge the gap between RC and UD designs

• We are working towards
  • Looking into the new eXtended Reliable Connection (XRC) transport provided in ConnectX adapters
  • Release of the Aptus (UD/RC) design in an upcoming version of MVAPICH
  • Investigate support for dynamic communication patterns
Efficient and Truly Passive MPI-3 RMA Using InfiniBand Atomics

M. Li, S. Potluri, K. Hamidouche, J. Jose and D. K. Panda
Motivation

- MPI Remote Memory Access (RMA) Model
- Minimizing communication overheads is key as applications scale to millions of processes/cores
- RMA model offers an alternative to Send/Recv based message passing model
  - Communication Epochs
    - Period between 2 synchronizations
    - One-sided communication
    - Windows area
- Promises better latency hiding, asynchronous progress and reduced synchronization overheads
- MPI-3 offers several extensions to provide more flexibility

MPI-3 RMA Passive Synchronization

- RMA offers flexible synchronization alternatives
  - Active: Fence and Post-Wait/Start-Complete
  - Passive: Lock/Unlock, Lock_all/Unlock_all
  - Shared/Exclusive (Lock/Unlock) and (Only Shared) (Lock_all/Unlock_all)
- Passive synchronization does not require involvement of target process
  - Less synchronization
  - Better overlap
- However, current implementations are based on two-sided operations
- Desirable to have a truly one-sided design offering
  - Performance (no remote polling)
  - Fairness (FIFO)
Problem Statement

• Can a truly passive locking mechanism be designed for InfiniBand Clusters?

• How can this design provide:
  - Performance (no remote Polling) - Fairness (FIFO => no starvation)

• Can the new locking mechanism benefits the performance of applications?
# Existing Passive Synchronization Semantics over IB

<table>
<thead>
<tr>
<th></th>
<th>Shared</th>
<th>Exclusive</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State-of-the art</strong></td>
<td>Send/Recv</td>
<td>Send/Recv</td>
<td>Restrict asynchronous progress</td>
</tr>
<tr>
<td>MPI Libraries</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Jiang et.al</strong></td>
<td>Atomics</td>
<td>Atomics</td>
<td>High network Traffic due to remote polling</td>
</tr>
<tr>
<td>(Compare_and_swap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Jiang et.al</strong></td>
<td>--</td>
<td>Atomics/Put</td>
<td>Shared mode of locking is not handled</td>
</tr>
<tr>
<td>(MCS based)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Santhanaraman</strong></td>
<td>Send/Recv</td>
<td>Atomics</td>
<td>Restrict asynchronous progress. High network Traffic</td>
</tr>
<tr>
<td><strong>et.al</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficient and Truly Passive Synchronization scheme

• Lock Data Structures
• Our locking mechanism depends on IB atomics to implement shared and exclusive mode of locking
• IB requires 64 bits buffer for atomic operations
• This 64 bits region is divided into three parts to handle different lock requests:
  • Shared Counter: count of the processes that own or have requested a shared lock
  • Exclusive Tail: rank of the process which is tail of the distributed queue
  • Exclusive Head: rank of the process which is head of the distributed queue
• In order to handle all possible lock requests, a distributed lock queue is maintained to ensure FIFO and avoid remote polling
• Data structures to implement the distributed lock queue:
  • – Wait-for array: used when shared lock comes after exclusive lock. This exclusive lock knows the list of processes that request shared lock after it
  • – Signal-to array: used when shared lock comes after exclusive lock. This exclusive lock wakes up pending processes that are waiting for the shared lock
  • – Exclusive-next: two element integer array. Used by processes requesting exclusive lock to form a distributed lock queue
  • – Exclusive-prev: one integer flag. Used by a process unlocking an exclusive lock to wake up another process waiting for an exclusive lock
Exclusive Locking Only

- RDMA operations: `compare_and_swap` and `Put`
- Lock requests are ordered in distributed queue
- Exclusive locks are granted in FIFO order
Shared Locking Only

• Atomic operation: Fetch_and_add. To decrement we add the MAX value

• Each process requires shared lock is able to get it after its atomic operation completes

• Each process releases shared lock by decrementing shared lock counter by 1
Interleaved Shared and Exclusive Locking

- Shared followed by exclusive lock: Process gets exclusive lock after all previously granted shared locks have been releases.

- Exclusive followed by shared lock: Process gets shared lock after the previous exclusive lock releases its lock
Intra-node Locking Design

• For intra-node locking, native loopback that needs a number of queue pairs \( (p+(p*(p-1))/2) \) is not an efficient implementation (\( P \) = number of process on a node)

• If the lock-unlock 64 bits data structures are allocated in the shared memory region, the number of queue pairs used is decreased from \( (p+(p*(p-1))/2) \) to \( p \)
  • Based on the intra-node locking design, if one process wants to acquire a lock from other process in the same node, it issue atomic operation to itself (loopback)
  • The locking/unlocking mechanisms are the same as discussed earlier
Lock_All/Unlock_All Implementation

• Lock_all and Unlock_all introduced in MPI-3 use only shared lock.
• In our design, they are implemented based on the lock/unlock mechanism discussed earlier.
• If MPI_MODE_NOCHECK is used, then they are implemented as No.Op
• Inside Lock_all function, call win_lock is explicitly called for every processes in the communicator
• For Unlock_all, the same mechanism is used to call unlock for every process in the communicator
Performance Evaluation

Experimental Setup

• Cluster A
- Xeon Dual quad-core processor (2.67 GHz) with 12GB RAM
- Mellanox QDR ConnectX HCAs (32 Gbps data rate) with PCI_Ex Gen2 interface
• Software stack
- Implemented on MVAPICH2-1.9 will be in future releases
- http://mvapich.cse.ohio-state.edu Latest releases: MVAPICH2-2.0a
- High Performance open-source MPI Library for InfiniBand, 10Gig/iWARP, and RDMA over Converged Enhanced Ethernet (RoCE)
  - MVAPICH (MPI-1), MVAPICH2 (MPI-2.2 and MPI-3.0), Available since 2002
  - MVAPICH2-X (MPI + PGAS), Available since 2012
  - Used by more than 2,077 organizations (HPC Centers, Industry and Universities) in 70 countries
MPI_Get with Lock-Unock

- For one MPI_Get latency:
  - Small messages: atomic based design incurs an overhead compared to two-sided based design: two-sided design coalesces the 3 operations in one message
  - Large messages: Amortized the overhead and have similar performance
- For eight MPI_Get latency, the overhead is amortized and we see similar performance with both designs
MPI_Get with Lock_all-Unlock_all

- Same trend for small messages
- The design could benefit large messages by asynchronously issuing lock/unlock requests from different processes
Overlap Benchmark

- Achieves almost optimal computation/communication overlapping
• This modified version of Splash LU Kernel does dense LU factorization
• Our design outperforms the two-sided approach by a factor of 49% and 35% on 4 and 32 processes
Conclusion

• Proposed Locking mechanism to implement both shared and exclusive lock with RDMA InfiniBand Atomics:
  • - No remote polling - FIFO order.
  • Show optimal computation communication overlap
  • Demonstrated up to 49% improvement using Splash LU Kernel
  • Evaluate our designs with more applications/systems
  • Provide RDMA based-designs for MPI-3 RMA over IB
References:

• http://nowlab.cse.ohio-state.edu/static/media/publications/abstract/liujiics03.pdf
• http://nowlab.cse.ohio-state.edu/static/media/publications/slide/koop-ipdps08.pdf
• http://web.cse.ohio-state.edu/~panda.2/5194/slides/5c_eurompi13_rma.pdf
Thank You

- Network-Based Computing Laboratory
  - http://nowlab.cse.ohio-state.edu/
- MVAPICH Web Page
  - http://mvapich.cse.ohio-state.edu/