

Paper Presentation – 4a

Karthik Gopalakrishnan

High Performance RDMA-Based MPI Implementation over InfiniBand

Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, Dhableswar K. Panda

Introduction

- The InfiniBand standard addresses shortcomings associated with traditional networking protocols.
- InfiniBand provides both channel & memory semantics.
- MPI, the de-facto standard in HPC, uses channel semantics for small messages, which does not perform as well as RDMA.
- This paper addresses these issues using techniques such as “persistent buffer association” and “RDMA polling set”.
- The design combines channel & RDMA semantics to achieve high performance & scalability.

InfiniBand Overview

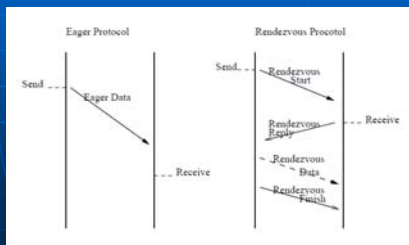
- Interconnect for Processing & IO Nodes
 - HCA – Sits on Processing Nodes
 - TCA – Sits on IO Nodes
- Queue Pairs (QP)
- Work Queue Element (WQE)
- Completion Queue (CQ)

Channel & Memory Semantics

- Channel Semantics
 - Two sided
 - Sender does a Send to send data
 - Receiver does a Recv to receive data
- Memory Semantics
 - Sender managed (One Sided)
 - Host does a RDMA_write to write data to a remote node's memory
 - Host does a RDMA_read to read data from a remote node's memory

MPI Protocols

- MPI Standard defines four communication modes:
 - Standard
 - Synchronous
 - Buffered
 - Ready



Rendezvous Protocol

- Sender handshakes with Receiver before sending messages.
- Used for large message transfers, since buffering messages or copying messages are expensive operations.
- These requirements match the properties of RDMA operations

Eager Protocol

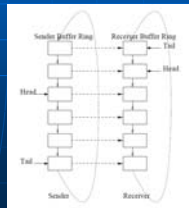
- Sender starts sending data without handshaking with the receiver.
- Beneficial for control messages & small messages since initial setup cost is avoided.
- However, receiver has to buffer the eager messages in the MPI layer till application posts a receive.
- Once application posts a receive, the message has to be copied to the application buffer.
- These requirements match the properties of Send/Receive semantics

RDMA Implementation of Eager Protocol

- RDMA operations requires the following:
 - Destination address should be known beforehand
 - Receiver should detect when incoming messages are received
- (1) is solved by using Persistent Buffer Association.
- (2) requires the receiver to poll on the buffers for completion.
- The receiver polls on all the buffers in a cyclic manner.
- Hybrid Approach - The receiver maintains a "RDMA polling set". The sender sends messages on the RDMA channel only if it is part of the receiver's polling set. Otherwise, it falls back to the Send/Receive semantic

RDMA Implementation of Eager Protocol (continued)

- Completion flag is the last byte of the message itself.
- Completion flag value is alternated for every send.



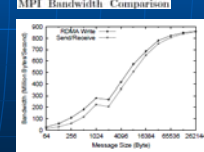
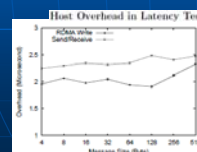
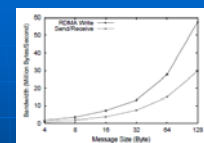
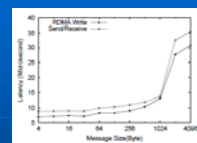
RDMA Implementation of Eager Protocol (continued)

- Sender side overhead minimized by preallocating data structures.
- Flow control uses a credit scheme.
- Credit information is piggybacked to sender.
- Senders can receive messages from the RDMA channel or from the Send/Receive channel. Hence, messages can arrive out-of-order.
- Ordering is ensured by using Packet Sequence Numbers

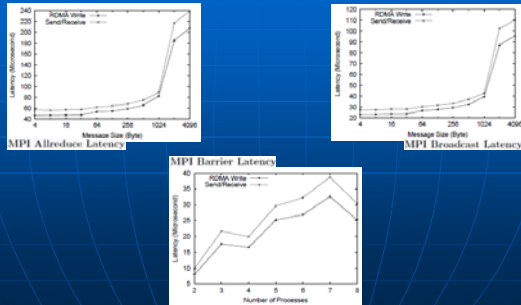
Performance Evaluation

- 8 Nodes each having
 - Dual Intel Xeon 2.4 GHz 512KB L2 Cache, 400 MHz FSB
 - Mellanox InfiniHost MT23108 DualPort 4X HCA
 - HCAs on a PCI-X 64 bit 133 MHz Slot
- Nodes connected with each other through a InfiniScale MT43132 eight 4x Port IB Switch

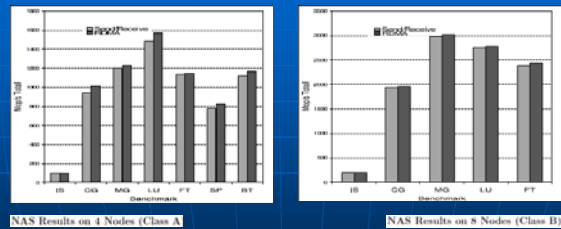
Performance Evaluation



Performance Evaluation



Performance Evaluation



Conclusion

- Design scheme brings the benefit of RDMA to large as well as small / control messages
- Achieves better scalability
- 24% reduction in latency
- 104% improvement in bandwidth
- 22% reduction in host overhead

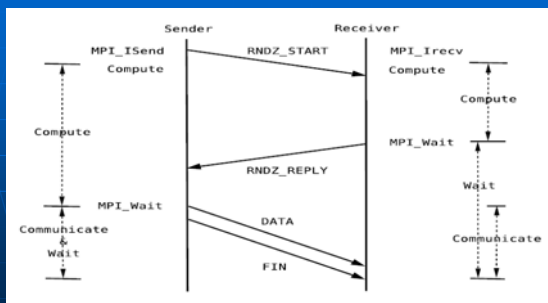
Paper Presentation – 4b

Karthik Gopalakrishnan

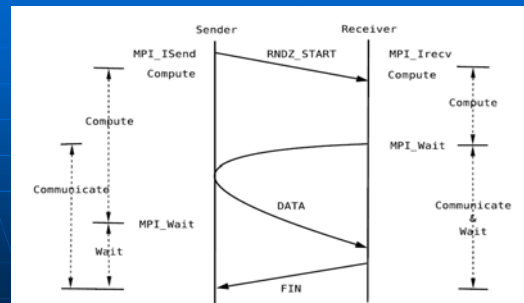
RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits

Sayantan Sur, Hyun-Wook Jin, Lei Chai, Dhableswar K. Panda

Rendezvous Protocol implementation using RDMA write



Rendezvous Protocol implementation using RDMA read



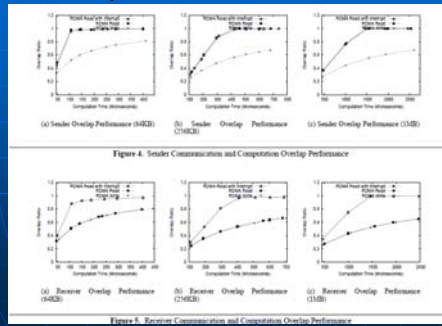
Reducing interrupt overheads

- Selective Interrupts
 - Use the `solicit_event` flag to trigger interrupts only for `RNDZ_START` & `RDMA Read complete`
- Interrupt Suppression
 - Suppress interrupts from multiple `RNDZ_START` messages
 - When the handler is invoked for the first `RNDZ_START`, disable notifications
 - Poll the CQs till there are no more completion descriptors
 - Enable notifications before exiting the handler
- Dynamic Interrupt Request
 - Enable interrupts only when pending receives are posted
 - Disable interrupts when there are no pending receives
 - Work in polling mode at other times
- Hybrid mode
 - Use a separate thread for the interrupt-based handler
 - Approach needs to be thread-safe

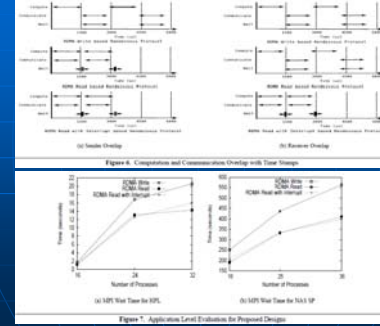
Experimental Evaluation

- Cluster A – 8 Nodes with
 - Intel Xeon 3.0 GHz 512 KB L2 Cache CPUs
 - 2 GB RAM
 - Connected to IB fabric through 64 bit 133 MHz PCI-X
- Cluster B – 32 Nodes with
 - Intel Xeon 2.66 GHz 512 KB L2 Cache CPUs
 - 2 GB RAM
 - Connected to IB fabric through 64 bit 133 MHz PCI-X
- All machines have Mellanox InfiniHost MT23108 HCAs
- Clusters connected using Mellanox MTS 14400 144 port switch

Experimental Evaluation



Experimental Evaluation



Conclusion

- Improve computation-communication overlap by using RDMA Read + Selective Interrupts
- This scheme yields better progress rate
- Gain in MPI Wait time improves with increase in system size. This has a positive impact on scalability

Paper Presentation – 4c

Karthik Gopalakrishnan

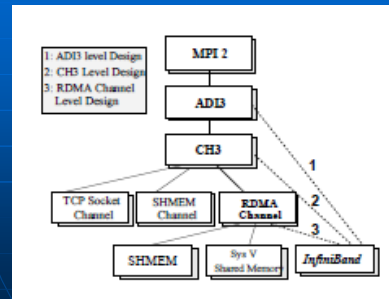
Design Alternatives and Performance Trade-offs
for Implementing MPI-2 over InfiniBand

Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin,
Dhabelaeswar K. Panda

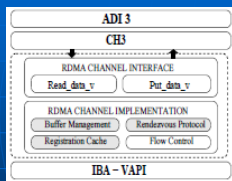
Introduction

- MPICH2 has a layered architecture to achieve portability and performance
- MPI-2 over Infiniband can be implemented at ADI3, CH3 or RDMA Channel layer
- This paper studies the complexity and performance impact of implementing MPI-2 over Infiniband at each of these layers

Layered Design of MPICH2



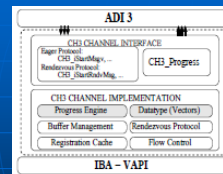
RDMA Channel Level Design & Implementation



- Arch dependant communication functions are encapsulated into a small set of interfaces
- Implements Eager & Rendezvous protocols
- User buffer registration cache is implemented in to reduce buffer registration overhead

- Communication progress is left to the upper layers
- Current CH3 stack maintains only one outstanding request in the RDMA layer
- This leads to inefficient channel utilization

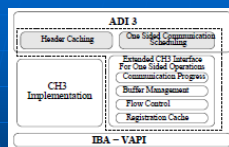
CH3 Level Design & Implementation



- The Communication progress engine has to be implemented in this layer
- CH3 layer may have to queue requests from the ADI3 layer, since the resources in the lower layer is limited

- Buffer registration cache
- Multiple rendezvous progresses can be started in parallel
- Data type communication can also be optimized
- More scope for optimization in this layer

ADI3 Level Design & Implementation



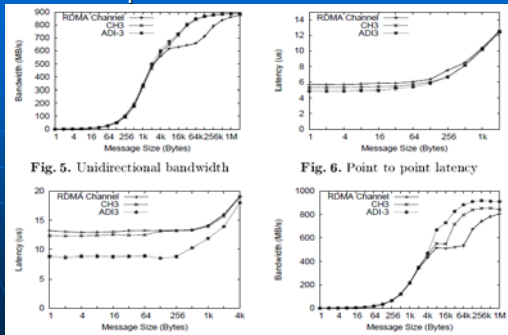
- The ADI3 layer has richer features. Hence, a complete implementation is not feasible
- Most of the implementation from the CH3 layer is reused in ADI3
- Header caching is an optimization that can be done in the ADI3 layer

- Implementing One-sided operations by using RDMA features. This cannot be done at the lower layers since we cannot distinguish data for one sided & two sided operations.

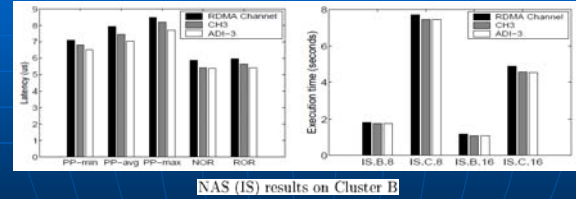
Performance Evaluation

- Cluster A – 8 Nodes with
 - Intel Xeon 3.0 GHz 512 KB L2 Cache CPUs
 - 2 GB RAM
 - Connected using InfiniScale MTS2400 Switch
- Cluster B – 32 Nodes with
 - Intel Xeon 2.66 GHz 512 KB L2 Cache CPUs
 - 2 GB RAM
 - Connected using MTS 14400 Switch
- All machines have Mellanox InfiniHost MT23108 HCAs

Experimental Evaluation



Experimental Evaluation



Conclusion

- Implementation at the CH3 & ADI3 layers provide substantial improvement compared to RDMA layer implementations.
- Progress Engine implementation is the main design complexity at the CH3 layer.
- Full blown ADI3 implementation is too complicated, but optimizations like header caching can be done here.
- Complexity of moving to ADI3 layer is well justified by the improvement in performance.

Paper Presentation – 4d

Karthik Gopalakrishnan

MPI over uDAPL - Can High Performance and Portability Exist Across Architectures

Wei Huang, Gopalakrishnan Santhanaraman, Hyun-Wook Jin, Dhabelaeswar K. Panda

Introduction

- The Top 500 list of super computers is a good scale to measure the performance of a computer with respect to processing large scale applications
- The current trends show that clusters are the preferred Hardware for such High Performance Computing
- And MPI is in de-facto standard for HPC applications
- Hence, MPI has to be portable over a large variety of hardware & should work with a wide array of interconnects

Introduction

- There are native implementations of MPI on various interconnects:
 - MPICH-MX for Myrinet
 - MVAPICH for InfiniBand
 - MPI/Elan for Quadrics
- These MPI implementations differ in applications they have been tested for.
- Hence, new deployments of applications over different MPI implementation may expose latent problems in the MPI library.
- Using a portable MPI library can result in reduced cost for the MPI library developer

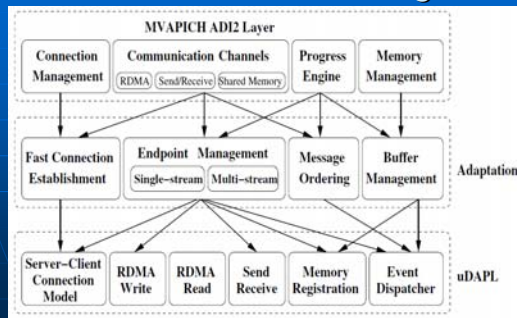
uDAPL

- uDAPL - User-Level Direct Access Transport APIs
- Specification drafted by the Direct Access Transport (DAT) Collaborative
- Provides light weight APIs
- Exports minimal RDMA functionality to the applications
- It provides a common terminology for VIA, IB & IWARP APIs

uDAPL

- HPC applications need to be rewritten to use the uDAPL APIs
- This is impractical for existing large critical applications
- A better approach would be to retain the application & rewrite the MPI library to use uDAPL underneath the MPI calls
- Idea is to provide close to native performance with the uDAPL device
- Achieved by mapping MVAPICH components to uDAPL services in the most efficient manner

uDAPL – Overall Design



uDAPL – Overall Design

- Efficient Connection Establishment
 - uDAPL only supports RC
 - Hence, All-to-All connections should be established during initialization
 - Use the Client-Server model provided by uDAPL
 - Initial approach was to create a Server Thread for every process to listen for incoming connections
 - Higher ranking process will act as a server
 - This approach was abandoned due to overhead of thread creation & portability
 - Second approach is to establish connections in (n-1) steps

uDAPL – Overall Design

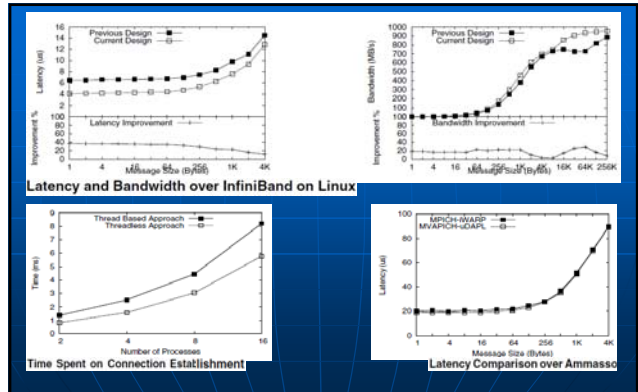
- Multichannel Communication
 - In a single stream design, there is only one connection between every process pair. We can use multiple connections for higher performance.
 - Use RDMA and Send-Receive channel for inter-node communication.
 - Use Shared Memory channel for intra-node communication.
 - uDAPL requires buffers to be registered with the Interface Adapter for RDMA operations. Since this is an expensive operation, it is taken out of the critical path by using preregistered buffers for small messages.
 - If buffers run out, we fall back to Send-Receive operations.
 - Buffers for large messages are registered on the fly.

uDAPL – Overall Design

- Multi Stream MPI Design
 - High performance can be obtained on Linux using the schemes described above. On Solaris however, one connection is not allowed to take up the entire bandwidth due to QoS reasons.
 - Hence, a Multi Stream design is used to overcome this restriction.
 - The approach used is to establish multiple EPs between each process pair.
 - All EPs use buffers from a shared buffer pool to improve buffer utilization & simplify flow control implementation
 - Eager messages use RDMA first & then Send / Receive
 - Use a fixed EP for small messages
 - Rendezvous messages select EPs on a Round Robin basis
 - Rendezvous Finish control message sent using Send Receive on the same EP

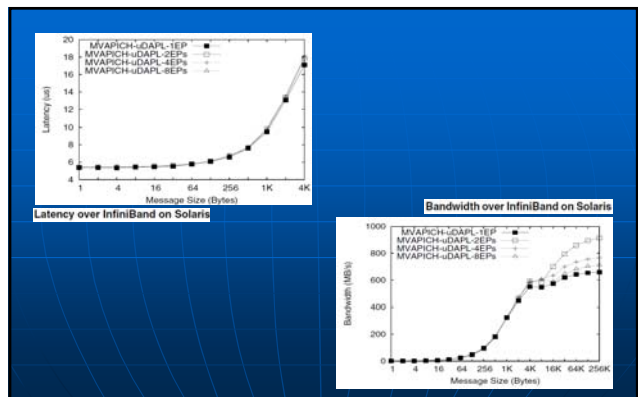
Performance Evaluation on Linux

- Experimental Setup for InfiniBand consisted of a 8 Node EM64T cluster with each node having
 - Dual 3.4 GHz Intel Xeon CPUs
 - 2 GB RAM
 - MT25208 HCA on PCI Express Bus
- Experimental Setup for Ammaso GigE consisted of a 8 Node IA32 cluster with each node having
 - Dual 3.0 GHz Intel Xeon CPUs
 - GigE adapter on PCI-X Bus



Performance Evaluation on Solaris

- Experimental Setup for InfiniBand consisted of a 8 Node Opteron cluster with each node having
 - Dual 2.2 GHz CPUs
 - 2 GB RAM
 - MT23108 HCA on PCI-X Bus
 - Solaris 10 Operating System
- SilverStorm 3032 switch used to connect the cluster together



Conclusion

- The uDAPL implementation on Solaris shows a 30% improvement in bandwidth for large messages.
- Upto 11% improvement in application performance.
- 30% improvement in start up time
- Performance over GigE is comparable to native.