Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits

Abhinav Vishn, Gopal Santhanaraman, Wei Huang
Hyun -Wook Jin and D. K. Panda

Technical Report
OSU-CISRC-5/05-TR33

Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits *

Abhinav Vishnu, Gopal Santhanaraman, Wei Huang,
Hyun -Wook Jin, and Dhabaleswar K. Panda

Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{vishnu, santhana, huanwei, jinhy, panda}@cse.ohio-state.edu

Abstract. In cluster computing, InfiniBand has emerged as a popular high performance interconnect with MPI as the *de facto* programming model. However, even with InfiniBand, bandwidth can become a bottleneck for clusters executing communication intensive applications. Multi-rail cluster configurations with MPI-1 are being proposed to alleviate this problem. Recently, MPI-2 with support for one sided communication is gaining significance. In this paper, we take the challenge of designing high performance MPI-2 *one sided communication* on multi-rail InfiniBand clusters. We propose a unified MPI-2 design for different configurations of multi-rail networks (*multiple ports, multiple HCAs* and *combinations*). We present various issues associated with one sided communication such as *multiple synchronization messages, scheduling of RDMA (Read, Write) operations, ordering relaxation* and discuss their implications on our design. Our performance results show that multi-rail networks can significantly improve MPI-2 one sided communication performance. Using PCI-Express with two-ports, we can achieve a peak *MPI_Put* bidirectional bandwidth of 2620 Million Bytes/s, compared to 1910 MB/s for single-rail implementation. For PCI-X with two HCAs, we can almost double the throughput and reduce the latency to half for large messages.

1 Introduction

High computational power of commodity PCs combined with the emergence of low latency and high bandwidth interconnects has led to the trend of *cluster computing*. In this area, Message Passing Interface (MPI) [8] has become the *de facto* standard for writing parallel applications. MPI-2 has been introduced as a successor of MPI-1 with *one sided communication* as one of its main additional features.

Recently, InfiniBand Architecture [10] has been proposed as the next generation interconnect for inter-process communication and I/O. Due to its open standard and high performance, InfiniBand is becoming increasingly popular for cluster computing. However, even with InfiniBand, network bandwidth can become the performance

* This research is supported in part by Department of Energy's grant #DE-FC02-01ER25506; National Science Foundation's grants #CCR-0204429 and #CCR-0311542; grants from Intel and Mellanox; and equipment donations from Intel, Mellanox, AMD, Apple and Sun Microsystems.

bottleneck for communication intensive applications. This is especially the case for clusters built with SMP (2-16 way symmetric multiprocessor systems) machines, in which multiple processes may run on a single node and must share the node bandwidth. Multi-rail [14](*multiple ports, multiple HCAs and combinations*) cluster configurations with MPI-1 are being proposed to alleviate this problem. Compared to MPI-1, MPI-2 is the next generation MPI standard with one sided operations (such as MPI_Put and MPI_Get). This leads to the following challenges.

1. *How to design support for one sided operations on multi-rail InfiniBand clusters?*
2. *How much benefits can be achieved compared to the single-rail implementation?*

In this paper, we take on these challenges. We propose a unified MPI-2 design with different configurations of multi-rail networks (*multiple ports, multiple HCAs* and *combinations*) for one sided communication. We present various issues associated with one sided communication (*multiple synchronization messages, scheduling of RDMA (Read, Write) operations, scheduling policies, ordering relaxation*) and discuss their implications on our design.

We implement our design on MVAPICH2 [1] and evaluate it with micro-benchmarks on different multi-rail configurations. Our performance results show that multi-rail networks can significantly improve MPI-2 one sided communication performance. With a two-rail InfiniBand cluster, we can achieve almost twice the *MPI_Put* bandwidth and half the *MPI_Put* latency for large messages compared to the single-rail MPI-2 implementation. Our experimental results show that, using two-ports on EM64T cluster with PCI-Express, we can achieve an *MPI_Put* bandwidth of 1500 Million Bytes/s (MB/s), and a bidirectional bandwidth of 2620 MB/s. Using two-HCAs on IA32 cluster with independent PCI-X buses, we can achieve a *MPI_Put* bandwidth of 1750 MB/s, and a bidirectional bandwidth of 1810 MB/s.

The rest of the paper is organized as follows: In section 2, we provide background information for InfiniBand, MVAPICH2 and multi-rail configurations. In section 3, we describe the multi-rail MPI-2 design for one sided communication and discuss the design issues. In section 4, we present performance results of our multi-rail MPI-2 implementation. In section 5, we present the related work. In section 6, we conclude and discuss our future directions.

2 Background

In this section, we provide background information for our work. First, we provide a brief introduction of InfiniBand. Then, we discuss some of the internals of MPI-2 one sided communication and their implementations over InfiniBand. We also present a brief overview of multi-rail InfiniBand clusters.

2.1 Overview of InfiniBand

The InfiniBand Architecture (IBA) [10] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a communication and management

[1] MVAPICH/MVAPICH2 [18] are high performance MPI-1 and MPI-2 implementations from The Ohio State University, currently being used by more than 210 organizations across 26 countries.

infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by *Host Channel Adapters* (HCA). . HCAs sit on processing nodes.

The InfiniBand communication stack consists of different layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A Queue-Pair based model is used in this interface. A *Queue Pair* in InfiniBand Architecture consists of two queues: a *Send Queue* and a *Receive Queue*. The send queue holds instructions to transmit data and the receive queue holds instructions which describe where the received data is to be placed. Communication operations are described in *Work Queue Requests* (*WQR*), or *descriptors*, and submitted to the work queue. The completion of WQRs is reported through *Completion Queues* (CQs). InfiniBand supports different classes of transport services. Most of MPI implementations are based on *Reliable Connection* (RC) service provided by InfiniBand. InfiniBand Architecture supports both channel and memory semantics for RC service. In channel semantics, send/receive operations are used for communication. In memory semantics, InfiniBand supports *Remote Direct Memory Access* (*RDMA*) operations, including RDMA write and RDMA read. RDMA operations are one-sided and do not incur software overhead at the remote side. In these operations, the sender can directly access remote memory by posting RDMA descriptors. The operation is transparent to the software layer at the receiver side.

At the physical layer, InfiniBand supports different link speeds. Most of the currently available HCAs support 4x links, which can potentially achieve a peak bandwidth of 1 GB/s. 12x links are also available. However, currently they are mainly used to interconnect different switches rather than end nodes.

2.2 MPI-2 one sided communication

In many parallel scientific applications, the data distribution changes dynamically and the data access pattern is irregular. For such applications, each process computes the data it needs to access or update on other processes. However, a process may not know the location of data in its local memory, which needs to be read or updated by other processe(s). In some cases it may not even know the identification of the remote processe(s). Hence, in these situations, only one process in the communication is aware of all the parameters required to transfer the data. The emerging MPI-2 standard with one sided communication operations specifically targets such communication patterns.

In MPI-2 one sided communication, the sender can access the remote address space directly. Such one sided communication is also referred to as *Remote Memory Access* or *RMA* communication. In this model, the *origin process* (the process that issues the RMA operation) provides necessary parameters needed for communication. The area of memory on the *target process* accessible by the origin process is called a *Window*. MPI-2 specification defines various communication operations:

1. *MPI_Put* operation transfers the data to a window in the target process
2. *MPI_Get* operation transfers the data from a window in the target process
3. *MPI_Accumulate* operation combines the data movement to target with a reduce operation

As per the semantics of one sided communication, the return of the one sided operation call does not guarantee the completion of the operation. In order to guarantee the completion of one sided operation, explicit synchronization operations must be

used. In MPI-2, synchronization operations are classified as *active* and *passive*. Active synchronization involves both sides of communication while passive synchronization only involves the origin side. We mainly focus on active synchronization in this paper.

The period between two synchronization calls is called as *access epoch* and *exposure epoch* on the origin and target process, respectively. MPI-2 semantics allows multiple communication calls during an access epoch. This is done to amortize the overhead of synchronization over multiple communication operations.

2.3 MVAPICH2

MVAPICH2[18] is our high performance implementation of MPI-2 over InfiniBand. The implementation is based on MPICH2. As a successor of MPICH[8], MPICH2[1] supports MPI-1 as well as MPI-2 extensions including one sided communication. One sided communication can be implemented using a variety of approaches. One approach is to use the point to point implementation provided by MPICH2 for one sided communication.

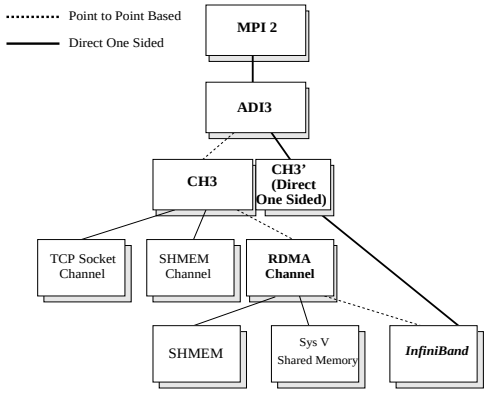

Fig. 1. Implementations of one sided communication in MVAPICH2

This approach involves the remote host for communication and synchronization operations. In the second approach, the one sided operations is implemented at the CH3 level by extending the CH3 interface [13, 12]. This approach shows benefits with respect to latency and bandwidth for regular communication patterns. It also provides better overlap between computation and communication along with scalability. We refer to the first approach as *Point to Point Based* and second approach as *Direct One Sided*. Fig. 1 shows the path taken by these approaches. In this paper, we design the *Direct One Sided* over multi-rail InfiniBand clusters implementation along with *active* mode of synchronization.

2.4 Multi-Rail InfiniBand Configurations

Multi-rail networks can be built by using *multiple HCAs* on a single node, or by using *multiple ports* in a single HCA.

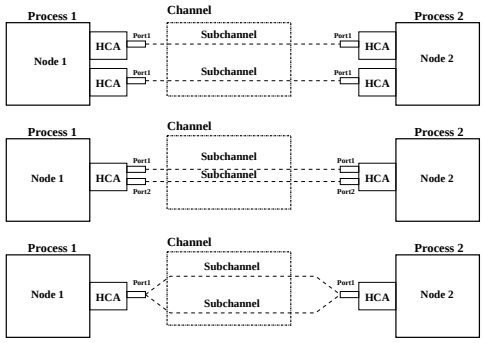

Fig. 2. Subchannel Abstraction

In an MPI application, any pair of processes can communicate with each other. This is implemented in MPICH2 designs by an abstraction called *virtual channel*. A virtual channel can be regarded as an abstract communication channel between two processes. It may not necessarily correspond to a physical connection of the underlying communication layer.

In [14], we have proposed enhanced virtual channel abstraction to provide a unified solution to support multiple HCAs, multiple ports, and multiple paths in a single port. In our proposed design, a virtual channel can consist of multiple *virtual subchannels* (referred as subchannels from here on-wards). Each subchannel refers to a path of communication between end nodes.

This enhanced abstraction eases the way to deal with all the multi-rail configurations (multiple ports, multiple HCAs and combinations). In the case of each node having multiple HCAs, subchannels for a virtual channel correspond to connections that go through different HCAs. If we would like to use multiple ports of the HCAs, we can set up subchannels so that there is one connection for each port. Similarly, different subchannels/connections can be set up in a single port that follow different paths. Once all the connections are initialized, the same subchannel abstraction is used for communication in all cases. Therefore, there is essentially no difference for all the configurations except for the initialization phase. The subchannel abstraction can also easily deal with cases in which we have a combination of multiple HCAs, multiple ports, and multiple paths in a single port. This idea is further illustrated in Figure 4.

3 Multi-rail Layer Design for MPI-2 One Sided Communication

In this section, we present the design issues involved with MPI-2 one sided communication on multi-rail InfiniBand clusters.

3.1 Basic Architecture

The basic architecture of our design to support multi-rail networks for MPI-2 one sided communication is shown in Figure 3. In the figure, we can see that besides the MPI-2, Direct One Sided layer and InfiniBand layer, our design consists of an intermediate layer, *Multi-rail Layer*.

This layer takes the responsibility of scheduling messages on the available subchannels. Besides this, it takes care of the correctness issues like *Handling Multiple HCAs* and *Multiple Synchronization Messages* and efficiency issues like *Scheduling Policies*, *Ordering Relaxation* and *Scheduling of RDMA Read and RDMA Write Operations*.

In [14], we focused on virtual channel abstraction to unify different multi-rail configurations (*multiple ports, multiple adapters and combinations*). We incorporate a similar

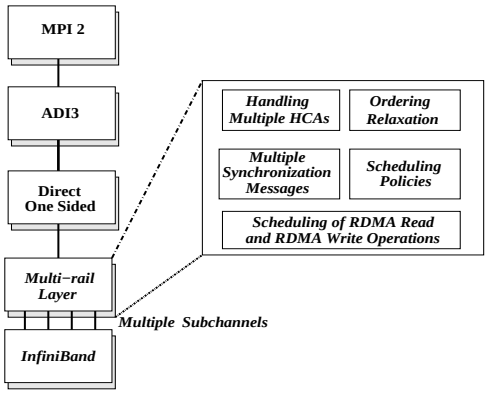

Fig. 3. Basic Architecture

design to unify multi-rail configurations for MPI-2 in this paper. At the channel level, we maintain a set of data structures to coordinate between different subchannels.

3.2 Detailed Design Issues

Our multi-rail MPI-2 implementation is based on MVAPICH2 [18], our MPI-2 implementation over InfiniBand. MVAPICH2 is derived from MPICH2, which is developed at Argonne National Labs and one of the popular MPI-2 implementations. The one sided operations have been implemented at the CH3 layer, by extending the CH3 interface.

In this section, we discuss the design challenges involved for multi-rail MPI-2 design associated at the Multi-rail Layer.

Multiple Synchronization Messages: In order to initiate the one sided communication, the origin process calls *win_start* to open a window. The target process *posts* the buffers for the window. Once the one sided communication is done, a synchronization message needs to be sent to the target process. The receipt of synchronization message guarantees the data transfer of previously issued RMA operations. However, when multiple subchannels are used, data transfer on one subchannel might not have finished even though other subchannels would have received the synchronization message. Hence, we need to issue synchronization messages on each subchannel. Note that when the load on subchannels is balanced, the transfer of synchronization messages along multiple subchannels takes place in parallel, incurring very small overhead.

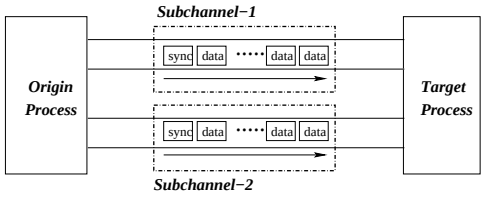

Fig. 4. Multiple Synchronization Messages

Scheduling of RDMA Read and RDMA Write Operations: In MPI-1, usually the two sided communication uses either RDMA Write or RDMA Read for data

transfer in InfiniBand. For many MPI-2 applications, in one sided communication, the *MPI_Put* and *MPI_Get* operations are implemented using RDMA Write and RDMA Read respectively. Since, RDMA Read and RDMA Write utilize bandwidth in different directions, it is important to schedule them independently with respect to each other's load on different subchannels.

In order to achieve this, we propose a load based fragmentation policy discussed in the next section, which maintains independent queues of *MPI_Put* and *MPI_Get* operations issued in an *epoch*. Trivally, this policy would fragment the messages equally on all subchannels in the presence of only one kind of one sided operation. In presence of a combination of one sided operations, each having the same size, this policy would fall back to equal fragmentation.

Scheduling Policies Classification based on Message Size: In this paper, we classify the policies used for scheduling based at different layers. As proposed in [9], we use *reordering* and *no reordering* policies at the CH3' (Direct One Sided) Layer.

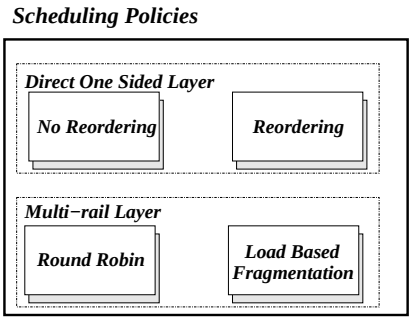

Fig. 5. Scheduling Policies at Different Layers

At the multi-rail layer, we do a classification of the policies based on the message size.

We employ the following policies:

- *Round Robin*
- *Load Balanced Fragmentation*

For small messages, we employ round robin policy. In this policy, the complete message is sent using one of the available subchannels in a round robin fashion. Fragmentation incurs overhead of posting descriptors on multiple subchannels, which is significant for small messages. Hence, we employ a *switchover threshold*, messages of size less than this threshold are scheduled in a round robin fashion. For large messages, we primarily use Load Balanced Fragmentation policy. In this policy, we divide the message in chunks and schedule them, so that the load on all subchannels is balanced. This policy leads to optimal utilization of all subchannels for large messages. In case of one sided communication, the switchover threshold can be reduced significantly in comparison to two-sided communication, due to the absence of rendezvous protocol. As a result, medium size messages can also benefit from fragmentation. We present the actual benefits in the performance evaluation section.

Ordering Relaxation: One sided communication imposes no ordering requirements for messages within an *epoch*, by the definition from the semantics. As a result, the one sided approach does not need to maintain ordering at the receiver side. We simplify our design by incorporating this fact, reducing the overhead of bookkeeping at the receiver side.

Handling Multiple HCAs: To initiate data transfer on multiple HCAs, we need to take care of the following issues:

- *Multiple Buffer Registrations*
- *Multiple Completion Queues*

RDMA Write and RDMA Read operations require the buffer to be registered with the NIC in order to initiate the data transfer. Since, we mainly employ the *fragmentation* policy for large messages, we register the complete buffer with all the available HCAs. The cost of registration is reduced by using a registration cache [15].

In addition to multiple registrations, we need to use multiple completion queues for multiple HCAs. On completion of a signaled RDMA Write or RDMA Read, a completion queue entry is generated, along with the status of the completion such as success/failure. At least, one completion queue needs to be associated with every InfiniBand HCA. We employ a simple round robin polling scheme to process the completion entries generated on multiple completion queues.

4 Performance Evaluation

In this section, we evaluate the performance of our multi-rail MPI-2 design over InfiniBand. We show the performance benefit we can achieve with multi-rail design compared to the single-rail implementation.

4.1 Experimental Testbed

We evaluated our implementation with multiple HCAs on IA32 systems comprising of independent PCI-X buses, and on EM64T systems comprising of PCI-Express bus and multiple ports per adapter. Our experimental testbed comprises of two clusters.

IA32 Cluster with Multiple HCAs: This cluster consists of two SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets. Each node has dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. We have used InfiniHost MT23108 Dual-Port 4x HCAs from Mellanox. The ServerWorks GC LE chipsets have two separate I/O bridges and three PCI-X 64-bit 133 MHz bus slots. To reduce the impact of I/O bus contention, the two HCAs are connected to separate PCI-X buses connected to different I/O bridges. The kernel version we used is Linux 2.4.22smp. The IBGD version is 1.6.1 and HCA firmware version is 3.3.2. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 2 GB of PC2100 DDR-SDRAM.

EM64T Cluster with Multiple ports: This cluster consists of two EM64T nodes having 8X PCI Express slots. Each node has two Intel Xeon CPUs running at 3.4 GHz processors, 512 KB L2 cache and 1 GB of main memory. This cluster uses III Generation MT25208 4X Dual Port HCAs from Mellanox. A combined unidirectional bandwidth of 8X can be used, when both ports are used for communication. The kernel version we used is Linux 2.4.21-15.EL. The IBGD version is 1.6.1 and HCA firmware version is 4.6.2. The Front Side Bus (FSB) of each node runs at 800MHz.

4.2 One Sided Communication Micro-Benchmarks

In this section, we briefly introduce the micro-benchmarks we used to evaluate the MPI-2 one sided operation performance on multiple HCAs. We use normal benchmarks, such as uni- and bi-directional bandwidth, as well as micro-benchmarks with other communication patterns.

Two processes are involved in uni-directional bandwidth test. The origin process starts a window access epoch, issues a number of RMA operations (MPI_Put for MPI_Put bandwidth test, MPI_Get for MPI_Get bandwidth test), and ends the access epoch. The target process just starts and ends a window exposure epoch. We measure the time taken by the whole process, divide it to the total amount of data being sent, and get the bandwidth performance. We refer to the number of one sided operations being issued in one access epoch as *window size*. In this paper, the window size we use is 16.

In MPI_Put bi-directional bandwidth test, each side starts and ends a window exposure epoch, meanwhile they start a window access epochs, each issue a number of MPI_Put operations to remote window, and end the access epoch. The window size used here is still 16. The total amount of data sent is divided by the total time of the whole process is reported as the bi-directional bandwidth of MPI_Put operation.

4.3 Performance Benefits of Multi-Rail Design

To evaluate the performance benefits of our multi-rail MPI-2 design, we compare it with our original MVAPICH2 design, which can only use only one-port of a nic. In the multi-rail design, we use *load balanced fragmentation* for large messages and *round robin* scheme for small messages. We present performance comparisons using latency, bandwidth and bidirectional bandwidth for *MPI_Put* and *MPI_Get* operations.

Microbenchmark Evaluation for *MPI_Put* Operation In Figures 6, 8 and 10 we present the results for *MPI_Put* bandwidth, bidirectional bandwidth and latency respectively for the IA32 cluster with multiple HCAs. We show the results for EM64T with two-ports on PCI-Express in Figures 7, 9 and 11.

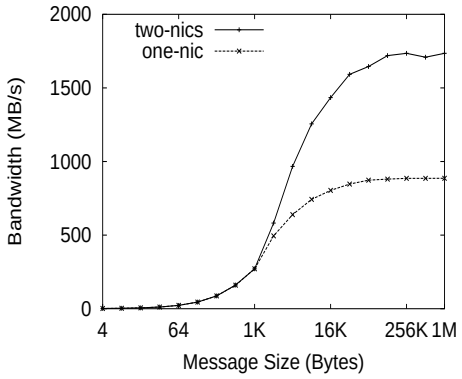

Fig. 6. MPI_Put Bandwidth on the IA32 Cluster

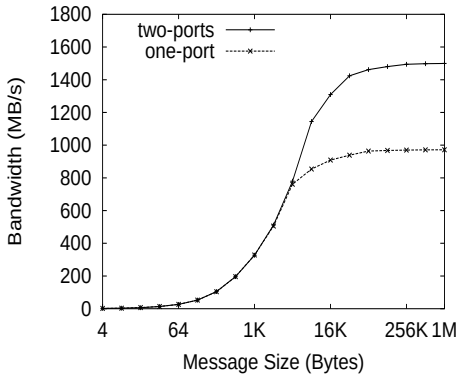

Fig. 7. MPI_Put Bandwidth on the EM64T Cluster

In Figure 6, we observe that for small messages (less than or equal to 1KBytes) , both multi-rail design and the original implementation perform comparably. For large messages, multi-rail design outperforms the original implementation considerably. With multi-rail design, we can achieve a maximum peak unidirectional *MPI_Put* bandwidth of 1750 MB/s in comparison to 880 MB/s for our original implementation. We also notice, that due to the absence of rendezvous protocol, medium size messages (2KB - 16KB), can take advantage of load balanced fragmentation policy for multi-rail design.

We observe a similar trend for dual-port on EM64T in Figure 7. For messages of size greater than 8KBytes, we use fragmentation policy. We can achieve a peak bandwidth of 1500 MB/s using multi-rail design, in comparison to 971 MB/s for the original implementation capable of using only one-port of a nic.

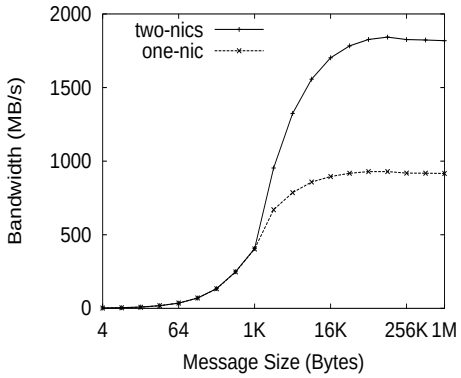

Fig. 8. MPI_Put Bidirectional Bandwidth on the IA32 Cluster

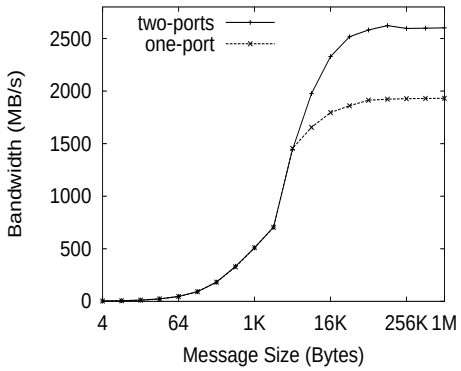

Fig. 9. MPI_Put Bidirectional Bandwidth on the EM64T Cluster

In figures 8 and 9, we compare the performance of *MPI_Put* bidirectional bandwidth for IA32 cluster and EM64T cluster, respectively. For IA32 cluster, due to the bottleneck of PCI-X, we can achieve only 941 MB/s for original implementation. However, using multi-rail design we can achieve a peak bidirectional bandwidth of 1810 MB/s. For EM64T cluster, we can achieve a peak bidirectional bandwidth of 2620 MB/s with two-ports in comparison to 1910 MB/s using the original implementation.

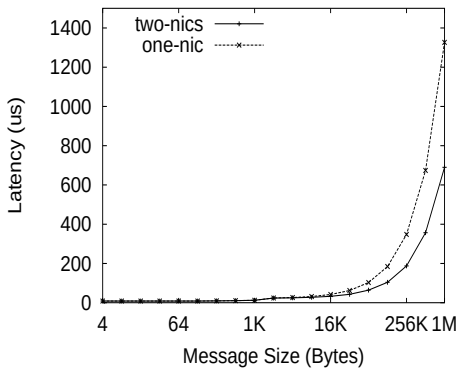

Fig. 10. MPI_Put Latency on the IA32 Cluster

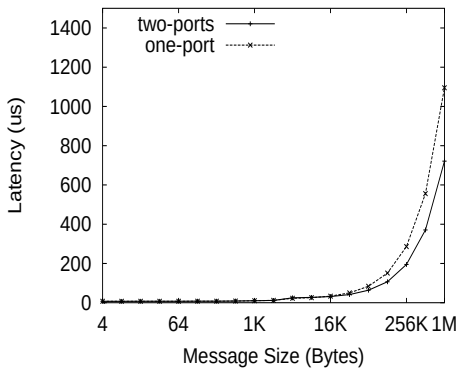

Fig. 11. MPI_Put Latency on the EM64T Cluster

In figures 10 and 11, we present the results for *MPI_Put* latency for IA32 and EM64T cluster, respectively. We observe that we perform almost similar with the original implementation for small messages. For large messages, we can improve the latency by 46% for IA32 cluster and 32% for EM64T cluster by using the multi-rail design.

Micro-Benchmark Performance Evaluation for *MPI_Get* Operation Figure 12 shows the bandwidth for *MPI_Get* operation for the IA32 cluster. We observe similar trends in performance as *MPI_Put* operation. Using the multi-rail design, we can achieve a peak bandwidth of 1705 MB/s, compared to 881 MB/s for the original implementation. For the EM64T cluster, we can achieve a peak bandwidth of 1494 MB/s, compared to 969 MB/s for the original implementation.

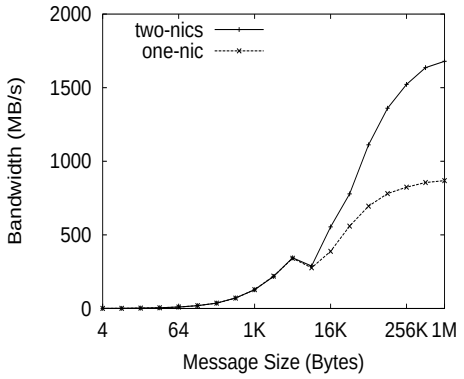

Fig. 12. MPI_Get Bandwidth on the IA32 Cluster

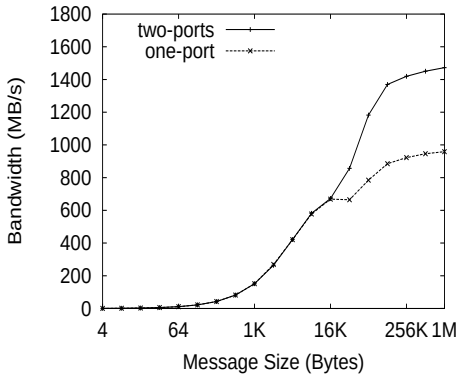

Fig. 13. MPI_Get Bandwidth on the EM64T Cluster

In figures 14 and 15, we present the results for *MPI_Get* latency for IA32 and EM64T cluster, respectively. We observe that we perform almost similar with the original implementation for small messages. For large messages, we can improve the latency by 45% for IA32 cluster and 33% for EM64T cluster by using multi-rail design.

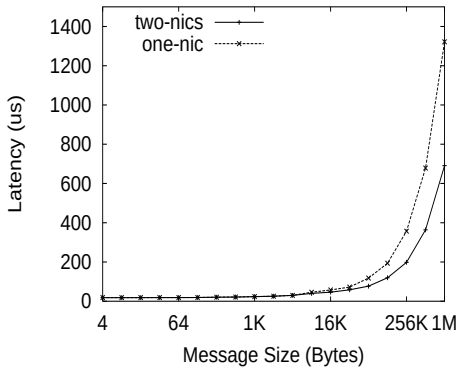

Fig. 14. MPI_Get Latency on the IA32 Cluster

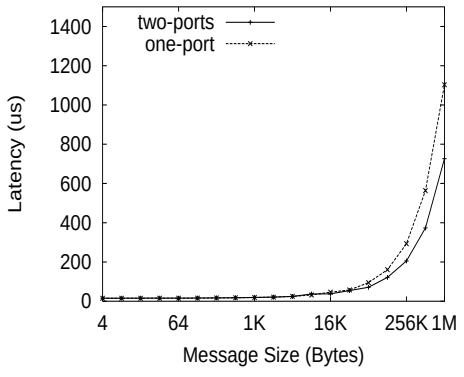

Fig. 15. MPI_Get Latency on the EM64T Cluster

One Sided Interleaving Test To show the impact of re-ordering of one sided communication, we use a throughput test which involves two processes. The first process starts a window access epoch and then issues 16 MPI_Put and 16 MPI_Get operations of the same size. The second process just starts an exposure epoch. The same sequence of operations are repeated for several iterations and we measure the maximum throughput we can achieve (in terms of Million Bytes/sec).

Fig. 16 shows the case where CH3' (Direct One Sided) layer issues all one sided operations in order. For MPI_Put, the data mainly flows from the origin process to target process, while for MPI_Get, the data mainly flows from the target process to origin process. So only one direction of the network link is fully used at one time. We can use the network bandwidth more efficiently if we re-order the one sided operations, as shown in Fig.17. Here we interleave the put and get operations so that we can almost achieve bi-directional bandwidth provided by the link.

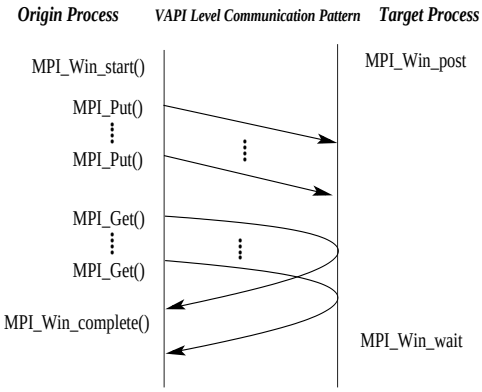

Fig. 16. Ordered issue of one sided operations

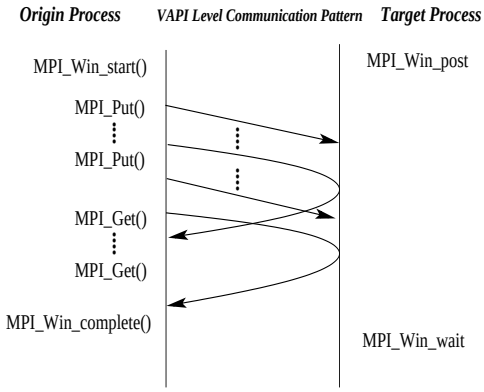

Fig. 17. Re-ordered issue of one sided operations

Figure 18 shows the performance achieved by a combination of policies at the CH3' layer and Multi-rail layer. At the multi-rail layer we use load balanced fragmentation policy. At the CH3' layer, we compare impact of reordering with no reordering, when combined with the multi-rail policy specified above.

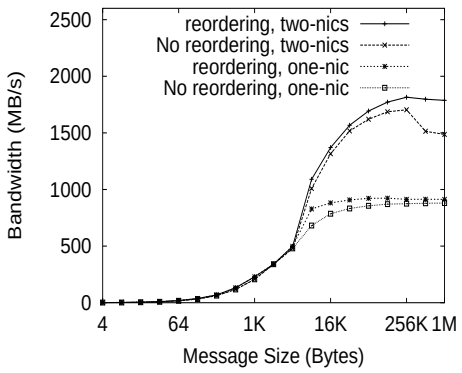

Fig. 18. Interleaved throughput on the IA32 Cluster

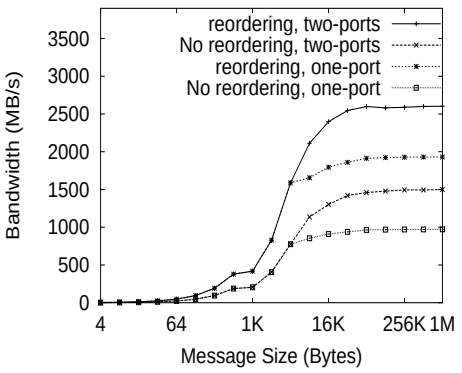

Fig. 19. Interleaved throughput on the EM64T Cluster

For IA32 cluster using two-nics, we can achieve almost 1703 MB/s without reordering, which is close to the multi-rail peak unidirectional bandwidth. With single-rail implementation, we can achieve a peak bandwidth of 880 MB/s without reordering. We notice, that with reordering for two-nics, we can almost achieve 1800 MB/s, almost the peak bidirectional bandwidth with two-nics. With single-rail implementation, due to the limitation of PCI-X, we can achieve only 907 MB/s.

In figure 19, we evaluate the performance of CH3' layer reordering, compared to the no reordering policy for the EM64T cluster. We use the load balanced fragmentation at the multi-rail layer. Using two-ports and reordering, we can achieve 2604 MB/s, which is almost the peak bidirectional bandwidth available with two-ports. It is interesting to notice, that reordering with single-rail implementation outperforms the combination of no reordering with multi-rail implementation. We attribute it to the fact that, PCI-Express can achieve 8X bidirectional bandwidth with one-port. However, due to the contention at the NIC, we cannot achieve a combined 8X unidirectional bandwidth using two-ports. Using reordering with single-rail implementation, we can achieve 1900 MB/s. However we can only achieve a peak bandwidth of 1474 MB/s using multi-rail implementation with no reordering. With no reordering for single-rail implementation, we can achieve 962 MB/s, which is close to the unidirectional bandwidth available with the single-rail implementation.

5 Related Work

In this section we discuss related work on one sided communication model as well as multi rail networks. There are several studies regarding implementing one sided communication in MPI-2.Some of the MPI-2 implementations which implement one sided communication are WMPI [17], NEC [11] and SUN-MPI [3]. In [22], reordering of one sided operations is proposed to reduce the cost of lock synchronization operation. Besides MPI, some other programming models which provide one sided communication are ARMCI [19], GASNET [2] and BSP [6].

Using interconnection networks for different topologies has been studied in [5]. Using multirail networks to build high performance clusters is proposed in [4]. The paper proposed different allocation schemes in order to eliminate conflicts at end points or I/O buses. However, the main interconnect focused in the paper was Quadrics [21] and the performance evaluation was done using simulation. In this paper, we focus on software support at the end points to build InfiniBand multirail networks for MPI-2 and present experimental performance data.

VMI2 [20] is a messaging layer developed by researchers at NCSA. An MPI implementation over VMI2 is also available [20]. VMI2 runs over multiple interconnects. [20] briefly mentions VMI2's ability to stripe large messages across different network interconnects. Instead of using a separate messaging layer, our design has integrated the multirail support with MPI-2 one sided protocols. LA-MPI [7] is an MPI implementation developed at Los Alamos National Labs. LA-MPI was designed with the ability to stripe message across several network paths. LA-MPI design includes a *path scheduler*, which determines the path a message will use for transfer. This design bears some similarity with our approach.

However, none of the above works have focussed on design of MPI-2 one-sided communication operations with multirail InfiniBand clusters.

6 Conclusions and Future Work

In this paper, we have presented the challenges *(Multiple synchronization messages, handling multiple HCAs, scheduling policies, ordering relaxation)* associated with designing MPI-2 one sided communication over multirail Infiniband networks. We have implemented our design and presented the performance evaluation for microbenchmarks. We have observed that multirail InfiniBand clusters can significantly improve the performance for one sided communication. Using a two rail cluster, we have achieved almost doubled the throughput and reduced the latency to half with *MPI_Put* and *MPI_Get* operations for large messages. We have also observed that reordering policy can significantly improve the performance for communication patterns with a mix of one sided operations.

In future, we plan to evaluate our implementation on large scale clusters for applications with one sided communication. We also plan to evaluate the scheduling policies in depth, to take care of different communication patterns for one sided communication.

7 Software Distribution

As indicated earlier, the open-source MVAPICH2 [16] software is currently being used by more than 210 organizations world-wide. The latest release is 0.6.0. The proposed MPI-2 multirail one sided communication solution will be available in the 0.7.0 release.

8 Acknowledgements

We would like to thank Sayantan Sur, Amith R Mamidala, Sundeep Narravula, Lei Chai, Pavan Balaji, Karthikeyan Vaidyanathan, and Weikuan Yu for their feedback on technical issues.

References

1. Argonne National Laboratory. MPICH2. http://www-unix.mcs.anl.gov/mpi/mpich2/.
2. Dan Bonachea. GASNet Specification, v1.1. Technical Report UCB/CSD-02-1207, Computer Science Division, University of California at Berkeley, October 2002.
3. Stephen Booth and Fernando Elson Mourao. Single Sided MPI Implementations for SUN MPI. In *Supercomputing*, 2000.
4. Salvador Coll, Eitan Frachtenberg, Fabrizio Petrini, Adolfy Hoisie, and Leonid Gurvits. Using multirail networks in high-performance clusters. In *CLUSTER '01: Proceedings of the 3rd IEEE International Conference on Cluster Computing*, page 15, Washington, DC, USA, 2001. IEEE Computer Society.
5. J. Duato, S. Yalamanchili, and L. Ni. *Interconnection Networks: An Engineering Approach.* The IEEE Computer Society Press, 1997.
6. M. Goudreau, K. Lang, S. B. Rao, T. Suel, and T. Tsantilas. Portable and Effcient Parallel Computing Using the BSP Model. *IEEE Transactions on Computers*, pages 670–689, 1999.
7. Richard L. Graham, Sung-Eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski. A network-failure-tolerant message-passing system for terascale clusters. *Int. J. Parallel Program.*, 31(4):285–303, 2003.

8. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.
9. Wei Huang, Gopal Santhanaraman, Hyun Wook Jin, and Dhabaleswar K. Panda. Scheduling of MPI-2 One Sided Operations On InfiniBand. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '05)*.
10. InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.
11. J. Traff and H. Ritzdorf and R. Hempel. The Implementation of MPI-2 One-Sided Communication for the NEC SX. In *Proceedings of Supercomputing*, 2000.
12. W. Jiang, J.Liu, H. W. Jin, D. K. Panda, D. Buntinas, R.Thakur, and W.Gropp. Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. EuroPVM/MPI, September 2004.
13. J. Liu, W. Jiang, Hyun-Wook Jin, D. K. Panda, W. Gropp, and Rajeev Thakur. High Performance MPI-2 One-Sided Communication over InfiniBand. International Symposium on Cluster Computing and the Grid (CCGrid 04), April 2004.
14. Jiuxing Liu, Abhinav Vishnu, and Dhabaleswar K. Panda. Building multirail infiniband clusters: Mpi-level design and performance evaluation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 33, Washington, DC, USA, 2004. IEEE Computer Society.
15. Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. High Performance RDMA-Based MPI Implementation over InfiniBand. In *17th Annual ACM International Conference on Supercomputing*, June 2003.
16. Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kinis, Darius Buntinas, Weikuan Yu, Balasubraman Chandrasekaran, Ranjit Noronha, Peter Wyckoff, and Dhabaleswar K. Panda. MPI over InfiniBand: Early Experiences. Technical Report, OSU-CISRC-10/02-TR25, Computer and Information Science, the Ohio State University, January 2003.
17. Fernando Elson Mourao and J Gabriel Silva. Implementing MPI's One-Sided Communications for WMPI. In *EuroPVM/MPI*, September 1999.
18. Network-Based Computing Laboratory. MVAPICH: MPI for InfiniBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html, January 2003.
19. J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586, 1999.
20. Scott Pakin and Avneesh Pant. VMI 2.0: A dynamically reconfigurable messaging layer for availability, usability, and management.
21. F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002.
22. Rajeev Thakur, William Gropp, and Brian Toonen. Minimizing Synchronization Overhead in the Implementation of MPI One-Sided Communication. In *EuroPVM/MPI*, September 2004.