High Performance RDMA-Based MPI Implementation over InfiniBand[*]

Jiuxing Liu Jiesheng Wu Dhabaleswar K. Panda

Computer and Information Science
The Ohio State University
Columbus, OH 43210
{liuj, wuj, panda}@cis.ohio-state.edu

ABSTRACT

Although InfiniBand Architecture is relatively new in the high performance computing area, it offers many features which help us to improve the performance of communication subsystems. One of these features is Remote Direct Memory Access (RDMA) operations. In this paper, we propose a new design of MPI over InfiniBand which brings the benefit of RDMA to not only large messages, but also small and control messages. We also achieve better scalability by exploiting application communication pattern and combining send/receive operations with RDMA operations. Our RDMA-based MPI implementation currently delivers a latency of 6.8 microseconds for small messages and a peak bandwidth of 871 Million Bytes (831 Mega Bytes) per second. Performance evaluation at the MPI level shows that for small messages, our RDMA-based design can reduce the latency by 24%, increase the bandwidth by over 104%, and reduce the host overhead by up to 22%. For large messages, we improve performance by reducing the time for transferring control messages. We have also shown that our new design is beneficial to MPI collective communication and NAS Parallel Benchmarks.

1. INTRODUCTION

During the last ten years, the research and industry communities have been proposing and implementing user-level communication systems to address some of the problems associated with the traditional networking protocols [25, 3, 19, 24]. The Virtual Interface Architecture (VIA) [8] was proposed to standardize these efforts. More recently, InfiniBand Architecture [12] has been introduced which combines storage I/O with Inter-Process Communication (IPC).

InfiniBand Architecture offers both channel and memory semantics. In channel semantics, send and receive operations are used for communication. In memory semantics, Remote Direct Memory Access (RDMA) operations are used. These two different semantics raise the interesting question of how to use both of them to design high performance communication subsystems.

In high performance computing area, MPI has been the *de facto* standard for writing parallel applications. Existing designs of MPI over VIA [13] and InfiniBand [14] use send/receive operations for small data messages and control message and RDMA operations for large data messages. However, due to their 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. Thus these designs have not achieved the best performance for small data messages and control messages.

In this paper we propose a method which brings the benefit of RDMA operations to not only large messages, but also small and control messages. By introducing techniques such as *persistent buffer association* and *RDMA polling set*, we have addressed several challenging issues in the RDMA-based MPI design. Instead of using only RDMA operations for communication, our design combines both send/receive and RDMA. By taking advantage of send/receive operations and the Completion Queue (CQ) mechanism offered by InfiniBand, we are able to simplify our design and achieve both high performance and scalability.

Our RDMA-based MPI implementation currently delivers a latency of 6.8 microseconds for small messages and a peak bandwidth of 871 Million Bytes (831 Mega Bytes) per second. Our performance evaluation shows that for small messages RDMA-based design can reduce the latency by 24%, increase the bandwidth by over 104%, and reduce the host overhead by up to 22%. For large messages, we improve performance by reducing the time for transferring control messages. We have also shown that our new design benefits MPI collective communication and NAS Parallel Benchmarks.

The rest of this paper is organized as follows: In Section 2 we provide an overview of InfiniBand Architecture. In Section 3 we present how we have designed our RDMA-based protocol to support MPI. We discuss detailed design issues in Section 4. Section 5 describes our implementation. Performance evaluation is presented in Section 6. In Section

[*]This research is supported in part by Sandia National Laboratory's contract #30505, Department of Energy's Grant #DE-FC02-01ER25506, and National Science Foundation's grants #EIA-9986052 and #CCR-0204429.

.

7 we discuss related work. Conclusions and future work are presented in Section 8.

2. INFINIBAND OVERVIEW

The InfiniBand Architecture (IBA) [12] defines a System Area Network (SAN) for interconnecting processing nodes and I/O nodes. It provides the communication and management infrastructure for inter-processor communication and I/O. In an InfiniBand network, processing nodes and I/O nodes are connected to the fabric by Channel Adapters (CA). Channel Adapters usually have programmable DMA engines with protection features. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs sit on processing nodes. TCAs connect I/O nodes to the fabric.

The InfiniBand communication stack consists of different layers. The interface presented by Channel adapters to consumers belongs to the transport layer. A queue-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 that describe where received data is to be placed. Communication operations are described in Work Queue Requests (WQR), or descriptors, and submitted to the work queue. Once submitted, a Work Queue Request becomes a Work Queue Element (WQE). WQEs are executed by Channel Adapters. The completion of work queue elements is reported through Completion Queues (CQs). Once a work queue element is finished, a completion queue entry is placed in the associated completion queue. Applications can check the completion queue to see if any work queue request has been finished.

2.1 Channel and Memory Semantics

InfiniBand Architecture supports both channel and memory semantics. In channel semantics, send/receive operations are used for communication. To receive a message, the programmer posts a receive descriptor which describes where the message should be put at the receiver side. At the sender side, the programmer initiates the send operation by posting a send descriptor. The send descriptor describes where the source data is but does not specify the destination address at the receiver side. When the message arrives at the receiver side, the hardware uses the information in the receive descriptor to put data in the destination buffer. Multiple send and receive descriptors can be posted and they are consumed in FIFO order. The completion of descriptors are reported through CQs.

In memory semantics, RDMA write and RDMA read operations are used instead of send and receive operations. These operations are one-sided and do not incur software overhead at the other side. The sender initiates RDMA operations by posting RDMA descriptors. A descriptor contains both the local data source addresses (multiple data segments can be specified at the source) and the remote data destination address. At the sender side, the completion of an RDMA operation can be reported through CQs. The operation is transparent to the software layer at the receiver side. Although InfiniBand Architecture supports both RDMA read and RDMA write, in the following discussions we focus on RDMA write because in the current hardware RDMA write usually has better performance.

There are advantages for using send/receive operations. First, as long as there are receive operations posted on the receiver side, the sender can send out messages *eagerly* without specifying the destination addresses. This matches well with the usage of eager data messages and some control messages in MPI. Second, the completion of receive operations can be detected through CQs. The CQ mechanism provides a convenient way of notifying the receiver about incoming messages. On the other hand, using send/receive operations also has disadvantages. First, the operations themselves are slower than RDMA operations because they are more complex at the hardware level. Second, managing and posting descriptors at the receiver side incur additional overheads, which further reduce the communication performance. Figure 1 shows the latencies of RDMA write and send/receive operations in our InfiniBand Testbed. (Details of this testbed are described in Section 6.1.) We can see that for small message, RDMA write performs better than send/receive.

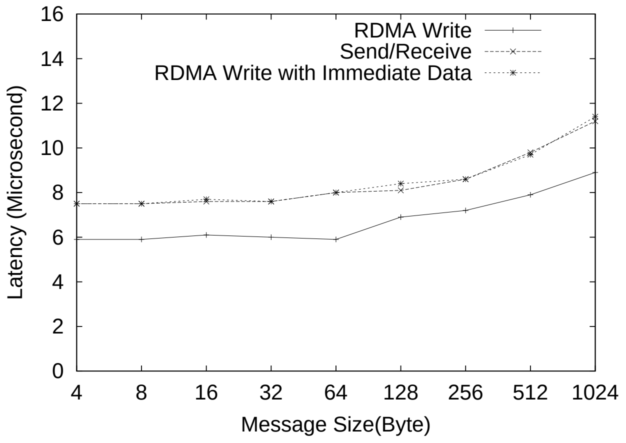

Figure 1: Latency of Send/Receive and RDMA

Similar to the send/receive operations, there are both advantages and disadvantages for using RDMA operations. The advantages include better raw performance at hardware level and less receiver overheads. However, if we use RDMA, the destination address must be known beforehand. Furthermore, the detection of incoming message must be handled explicitly at the receiver side.

InfiniBand also supports a variation of RDMA write called RDMA write with immediate data. In this operation, the sender specifies an immediate data field with the RDMA write operation. Then the RDMA write operation is carried out. Additionally, a receive descriptor is consumed and a CQ entry is generated which contains the immediate data. Since this operation also involves the receiver, it has almost the same overhead as a send/receive operation, as can be seen in Figure 1.

3. MAPPING MPI PROTOCOLS

InfiniBand Architecture offers a plethora of new mechanisms and services such as channel and memory semantics, multiple transport services, atomic operations, multicast, service level and virtual lanes. Many of these mechanisms and services can be useful in designing a high performance communication subsystem. In this section, we focus on how to exploit channel and memory semantics to support MPI.

MPI defines four different communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready* modes. Two internal protocols, *Eager* and *Rendezvous*, are usually used to implement these four communication modes. In Eager protocol, the message is pushed to the receiver side regardless of its state. In Rendezvous protocol, a handshake happens between the sender and the receiver via control messages before the data is sent to the receiver side. Usually, Eager protocol is used for small messages and Rendezvous protocol is used for large messages. Figure 2 shows examples of typical Eager and Rendezvous protocols.

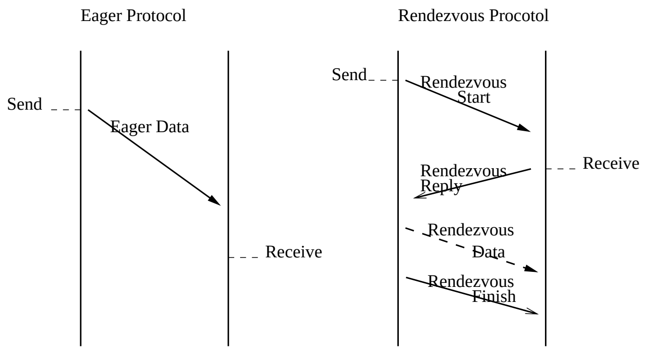

Figure 2: MPI Eager and Rendezvous Protocols

When we are transferring large data buffers, it is beneficial to avoid extra data copies. A zero-copy Rendezvous protocol implementation can be achieved by using RDMA write. In this implementation, the buffers are pinned down in memory and the buffer addresses are exchanged via the control messages. After that, the data can be written directly from the source buffer to the destination buffer by doing RDMA write. Similar approaches have been widely used for implementing MPI over different interconnects [13, 7, 2].

For small data transfer in Eager protocol and control messages, the overhead of data copies is small. Therefore, we need to push messages eagerly toward the other side to achieve better latency. This requirement matches well with the properties of send/receive operations. However, as we have discussed, send/receive operations also have their disadvantages such as lower performance and higher overhead. Next, we discuss different approaches of handling small data transfer and control messages.

3.1 Send/Receive Based Approach

In this approach, Eager protocol messages and control messages are transfered using send/receive operations. To achieve zero-copy, data transfer in Rendezvous protocol uses RDMA write operation.

In the MPI initialization phase, a reliable connection is set up between every two processes. For a single process, the send and receive queues of all connections are associated with a single CQ. Through this CQ, the completion of all send and RDMA operations can be detected at the sender side. The completion of receive operations (or arrival of incoming messages) can also be detected through the CQ.

InfiniBand Architecture requires that the buffers be pinned during communication. For eager protocol, the buffer pinning and unpinning overhead is avoided by using a pool of pre-pinned, fixed size buffers for communication. For sending an Eager data message, the data is copied to one of the buffers first and sent out from this buffer. At the receiver side, a number of buffers from the pool are pre-posted. After the message is received, the payload is copied to the destination buffer. The communication of control messages also uses this buffer pool. In Rendezvous protocol, data buffers are pinned on-the-fly. However, the buffer pinning and unpinning overhead can be reduced by using the pin-down cache technique [10].

To ensure that receive buffers are always posted when a message comes, a credit based flow control mechanism can be used. Credit information can be exchanged by piggybacking or using explicit credit control messages. For RDMA operations, flow control is not necessary because they do not consume receive descriptors.

In MPI, logically every process is able to communicate with every other process. Therefore, each process can potentially receive incoming messages from every other process. In send/receive based approach, we only need to check the CQ for incoming messages. The CQ polling time usually does not increase with the number of connections. Therefore, it provides us an efficient and scalable mechanism for detecting incoming messages. Figure 3 shows the CQ polling time with respect to different number of connections in our InfiniBand testbed.

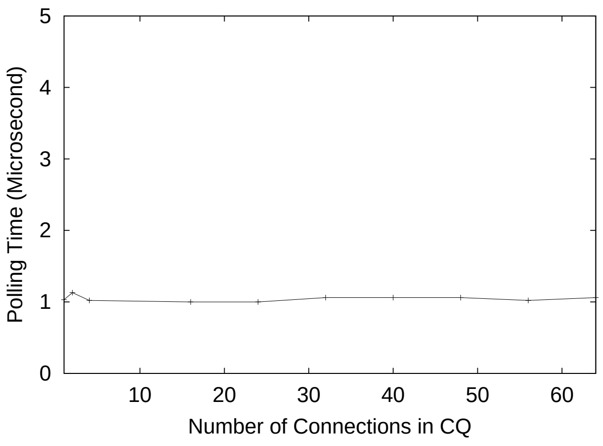

Figure 3: CQ Polling time

As we mentioned in section 2, there are also disadvantages for using the send/receive based approach. First, since the performance of send/receive is not as good as RDMA write, we cannot achieve the best latency for small data transfer and control messages. Second, we have to handle tasks such as allocating and de-allocating buffers from the pre-pinned buffer pool and re-posting receive descriptors. These tasks increase the overhead and communication latency.

3.2 RDMA-Based Approach

To overcome the drawbacks of the send/receive based approach, we have designed an RDMA write based approach for Eager protocol and control messages. In this approach, the communication of Eager protocol and control messages also goes through RDMA write operations. Therefore, we can achieve lower latency and less overhead. However, two difficulties must be addressed before we can use RDMA for data transfer:

- The RDMA destination address must be known before the communication.

- The receiver side must detect the arrival of incoming messages.

In current generation InfiniBand hardware, RDMA operations have high starting overhead. From Figure 4 we can see that the latency increases significantly if we use two RDMA write operations instead of one. Thus, it is desirable that we use as few RDMA operations as possible to transfer a message. Ideally, only one RDMA operation should be used.

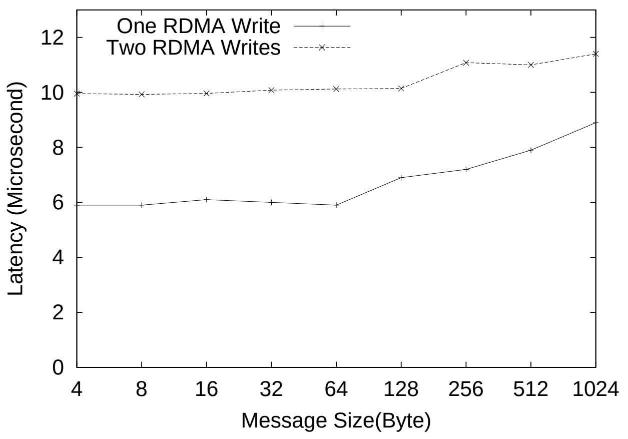

Figure 4: Latency of One RDMA Write versus Two RDMA Writes

To address the first problem, we have introduced a technique called *persistent buffer association*. Basically, for each direction of a connection we use two buffer pools: one at the sender and one at the receiver. Unlike other approaches in which the sender may use many buffers for an RDMA write operation, we have persistent correspondence between each buffer at the sender side and each buffer at the receiver side. In other words, at any time each buffer at the sender side can only be RDMA written to its corresponding buffer at the receiver side. These associations are established during the initialization phase and last for the entire execution of the program. Thus, the destination address is always known for each RDMA operation.

Persistent buffer association can also reduce the overhead of building RDMA descriptors. All fields of the descriptors can be filled out initially except for the data size field. When we reuse an RDMA buffer, the descriptor can be reused and only the size field needs to be changed.

The second problem can be broken into two parts: First, we need to efficiently detect incoming messages for a single connection. Second, we need to detect incoming messages for all connections in a process.

For RDMA write operations, the CQ mechanism cannot be used to report the completion of communication at the receiver side. The basic idea of detecting message arrival is to poll on the content of the destination buffer. In our scheme there are a pool of buffers at both the sender side and the receiver side. If the sender could use any buffer in the pool for RDMA write, it would be very difficult for the receiver to know where to poll for incoming messages. Therefore we organize the buffers as a ring. The sender uses buffers in a fixed, circular order so that the receiver always knows exactly where to expect the next message. The details of our design will be presented in the next section.

Once we know how to detect incoming messages for a single connection, multiple connections can be checked by just polling them one by one. However, the polling time increases with the number of connections. Therefore this approach may not scale to large systems with hundreds or thousands of processes.

3.3 Hybrid Approach

As we can see, RDMA and send/receive operations both have their advantages and disadvantages. To address the scalability problem in our previous design, we have enhanced our previous design by combining both RDMA write and send/receive operations. It is based on the observation that in many MPI applications, a process only communicates with a subset of all other processes. Even for this subset, not all connections are used equally frequently. Table 1 lists the average number of communication sources per process for several large-scale scientific MPI applications [23, 26] (values for 1024 processors are estimated from application algorithm discussions in the literature). Therefore, we introduce the concept of *RDMA polling set* at the receiver side. In our scheme, each sender has two communication channels to every other process: a send/receive channel and an RDMA channel. A sender will only use the RDMA channel for small messages if the corresponding connection is in the RDMA polling set at the receiver side. If a connection is not in the polling set, the sender will fall back on send/receive operations and the message can be detected through the CQ at the receiver. The receiver side is responsible for managing the RDMA polling set. The concept of RDMA polling set is illustrated in Figure 5. Ideally, the receiver should put the most frequently used connections into the RDMA polling set. On the other hand, the polling set should not be so large that the polling time is hurting performance.

Table 1: Number of distinct sources per process

Application	# of processes	Average # of sources
sPPM	64	5.5
	1024	6
Sphot	64	0.98
	1024	1
Sweep3D	64	3.5
	1024	4
Samrai4	64	4.94
	1024	10
CG	64	6.36
	1024	11

By restricting the size of the RDMA polling set, the receiver can efficiently poll all connections which use RDMA write for Eager protocol and control messages. Messages from all other connections can be detected by polling the CQ. In this way, we not only achieve scalability for polling but also get the performance benefit of RDMA.

Having two communication channels also helps us to simplify our protocol design. Instead of trying to handle everything through the RDMA channel, we can fall back on the send/receive channel in some infrequent cases. In the next section we will discuss how this strategy helps us to simplify the flow control of RDMA channels.

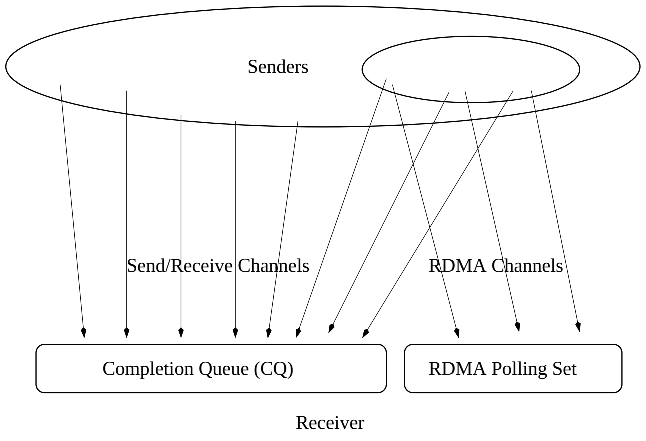

Figure 5: RDMA Polling Set

4. DETAILED DESIGN ISSUES

In this section, we discuss detailed issues involved in our design. First, we present the basic data structure for an RDMA channel. After that we discuss the communication issues for a single RDMA channel, including polling algorithm, flow control, reducing sender overhead and ensuring message order. Then we discuss how a receiver manages the RDMA polling set.

4.1 Basic Structure of an RDMA Channel

Unlike send/receive channels, RDMA channels are unidirectional. Figure 6 shows the basic structure of an RDMA channel. For each RDMA channel, there are a set of fixed size, pre-registered buffers at both the sender side and the receiver side. Each buffer at the sender side is persistently associated with one buffer at the receive side and its content can only be RDMA written to that buffer.

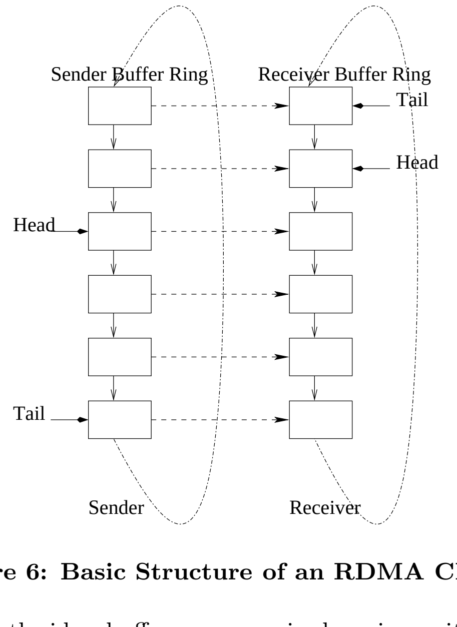

Figure 6: Basic Structure of an RDMA Channel

On both sides, buffers are organized as rings with the respective head pointers and tail pointers. The buffers run out for a sender when the head pointer meets the tail pointer. At the sender side, the head pointer is where the next outgoing message should be copied and RDMA written to the remote side. After the message is written, the head pointer is incremented. Later the receive side detects the incoming message and processes it. Only after this processing, this buffer can be used again for another RDMA write. The tail pointer at the sender side is to record those buffers that are already processed at the receiver side. The sender side alone cannot decide when to advance the tail pointer. This is done by a flow control mechanism discussed later.

At the receiver side, the head pointer is where the next incoming message should go. In order to check incoming messages, it suffices to just examine the buffer pointed by the head pointer. The head pointer is incremented after an incoming message is detected. When we have got an incoming message, the processing begins. After the processing finishes, the buffer is freed and it can be reused by the sender. However, the order in which the buffers are freed may not be the same as the order in which the messages arrive. Therefore we introduce the tail pointer and some control fields at the receiver to keep track of these buffers. The tail pointer advances if and only if the current buffer is ready for reuse.

4.2 Polling for a Single Connection

In order to detect the arrival of incoming messages for a single RDMA channel, we need to check the content of the buffer at the receiver side. In InfiniBand Architecture, the destination buffers of RDMA operations must be contiguous. Therefore a simple solution is to use two RDMA writes. The first one transfers the data and the second one sets a flag. Please note that by using two RDMA writes, we can be sure that when the flag is changed, the data must have been in the buffer because InfiniBand ensures ordering for RDMA writes.

The above scheme uses two RDMA writes, which increase the overhead as we have seen in Figure 4. There are two ways to improve this scheme. First, we can use the gather ability offered by InfiniBand to combine the two RDMA writes into one. The second way is to put the flag and the data buffer together so that they can be sent out by a single RDMA write. However, in both cases, we need to make sure that the flag cannot be set before the data is delivered. And to do this we need to use some knowledge about the implementation of hardware. In our current platform, gather list are processed one by one. And for each buffer, data is delivered in order (the last byte is written last). Thus, we need to put the flag after the data in the gather list, or to put the flag at the end of the data. Since using gather list complicates the implementation and also degrades performance (more DMA operations needed) as can been in Figure 7, we use the second approach: putting the flag at the end of the data buffer. Although the approach uses the in-order implementation of hardware for RDMA write which is not specified in the InfiniBand standard, this feature is very likely to be kept by different hardware designers.

Putting the flag at the end of the data is slightly more complicated than it looks because the data size is variable. The receiver side thus has to know where is the end of the message and where the flag is. To do this, we organize the buffer as in Figure 8. The sender side sets three fields: head flag, data size and tail flag before doing the RDMA write. The receiver first polls on the head flag. Once it notices that the head flag has been set, it reads the data size. Based on

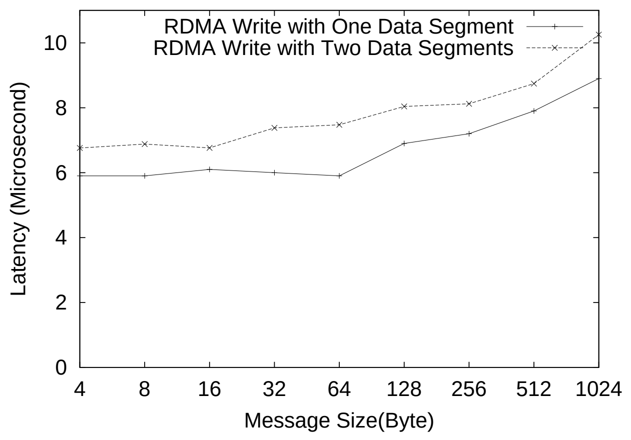

Figure 7: Latency of RDMA Write Gather

the data size, it calculates the position of the tail flag and polls on it.

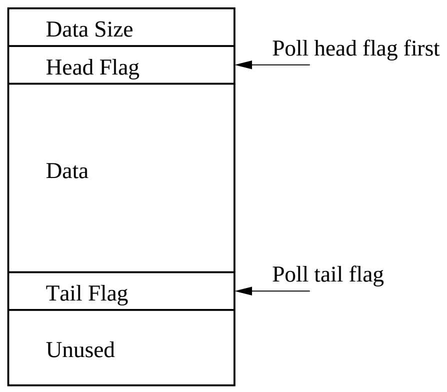

Figure 8: RDMA Buffer Structure for Polling

The above scheme has one problem. Since the receive buffer can be reused for multiple RDMA writes, it may happen that the value at the tail flag position is the same as the flag value. In this case, the send side should use two flag values and switch to another value. But how does the sender side know the value of the buffer at the receiver side? We notice that because of the persistent association between buffers, the buffer on the sender side should have the same content as the receiver side. Thus what we need to do at the sender side is the following[1]:

1. Set data size.
2. 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.
3. Set the head and tail flags.
4. Use RDMA write operation to transfer the buffer.

Initially, the head flag at the receiver side is cleared. The receiver polls by performing the following:

[1]Another approach called "bottom-fill" was used in [21].

1. Check to see if the head flag is set. Return if not.
2. 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.

4.3 Reducing Sender Side Overhead

Using RDMA write for small and control message can reduce the overhead at the receiver side because the receiver no longer needs to manage and post receive descriptors. In this section we describe how the overhead at the sender side can also be reduced by using our scheme.

At the sender side, there are two kinds of overheads related to the management of descriptors and buffers. First, before buffers can be sent out, descriptors must be allocated and all the fields must be filled. Second, after the operations are done, completion entries are generated for them in the CQ and the sender side must process them and take proper actions such as free the descriptor and the buffer.

To reduce the overheads of allocating and freeing descriptors, we store them together with the buffer. Since we have persistent association between source and destination buffers, all fields in the descriptors can be filled only once and reused except for the data size field. To deal with the overhead of completion entries in the CQ, we can use the unsignalled operations in InfiniBand Architecture. These operations will not generate CQ entries.

4.4 Flow Control for RDMA Channels

As we have mentioned in the previous subsection, before the sender can reuse an RDMA buffer for another operation, it must make sure that the receiver has already finished processing this buffer. To achieve this, a flow control mechanism is implemented for the RDMA channel:

- At the sender side, the head pointer is incremented after each send in the RDMA channel.
- At the receiver side, the head pointer is incremented after an incoming message is received.
- An RDMA channel cannot be used if its head pointer is equal to its tail pointer at the sender side. In this case, we fall back and use the send/receive channel.
- The receiver maintains a credit count for the RDMA channel. Each time a receiver RDMA buffer is freed, the credit count is increased if the buffer is pointed by the tail pointer. Then the receiver goes on and checks if the following buffers were already freed. If they were, the credit count and the tail pointer are incremented for each buffer. The checking is necessary because although the arrival of messages is in order, the buffers can be freed out of order.
- For each message sent out (either RDMA channel or send/receive channel), the receiver will piggyback the credit count.
- After the sender receives a message with a positive credit count, it increases its tail pointer.

One thing we should note is that when we run out of RDMA buffers, we fall back on the send/receive channel for communication because we have separate flow control for the send/receive channel. However, these cases do not happen often and RDMA channels are used most of the time.

4.5 Ensuring Message Order

Since we use Reliable Connection (RC) service provided by InfiniBand for our design, messages will not be lost and they are delivered in order. However, in our RDMA-based approach, there are two potential sources of incoming messages at the receiver side for each sender: the RDMA channel and the send/receive channel. The receiver has to poll on both channels to receive messages. Therefore it might receive messages out of order. This is not desirable because in MPI it is necessary to ensure the order of message delivery. To address this problem, we introduce a *Packet Sequence Number* (PSN) field in every message. Each receiver also maintains an *Expected Sequence Number* (ESN) for every connection. When an out-of-order message arrives, the receiver just switches to the other channel and delays processing of the current packet. It stays on the other channel until the PSN is equal to the current ESN.

4.6 Memory Usage for RDMA Channels

One concern for the RDMA-based design is memory usage. For each connection, we need to use two pools of prepinned buffers. If memory consumption for RDMA Channels is too large, application performance may suffer. Since the memory usage can potentially increase with the number of processes in an MPI application, it may also degrade the scalability of MPI on large-scale clusters.

There are several ways to address this problem. First, we can set up connections on demand to eliminate unnecessary connections. Since the number of connections is reduced, memory usage for RDMA channels is also decreased. Our previous work [26] implemented this mechanism for MVICH. We plan to incorporate this feature into our MPI implementation. Another way to reduce RDMA channel memory usage is to limit the number of RDMA channels. By limiting the size of the RDMA polling set, we can also effectively reduce the memory consumption.

We can decrease memory usage further by reducing the number of RDMA buffers for each connection. However, care must be taken in using this approach because too few RDMA buffers may result in reduced communication performance. We will evaluate this approach in Section 6.4.

4.7 Polling Set Management

In this subsection we describe our mechanism to manage polling sets for RDMA channels. Initially, all connections use send/receive channels. Each receiver is responsible for adding or deleting connections to the RDMA polling set. When a receiver decides to add or remove a connection, it tells the sender by piggybacking or explicitly sending a control packet. The sender side takes corresponding actions after receiving this information.

There are different policies that the receiver can use to manage the RDMA polling set (add or remove a connection). For an application with only a small number of processes, all connections can be put into the RDMA polling set because the polling time is small. For large applications, we need to limit the size of RDMA polling sets in order to reduce the polling time. A simple method is to put first N (N is the size of the RDMA polling set) channels with incoming messages into the RDMA polling set. As we can see from Table 1, this method works for many large scientific applications. For those applications which have a large number of communication destinations for each process, we can dynamically manage the RDMA polling sets by monitoring the communication pattern. Another method is to take some hints from the applications regarding the communication frequency of each connection. We plan to investigate along some of these directions.

The order of polling in the RDMA polling set can be very flexible. Different algorithms such as sequential, circular and prioritized polling can be used. Polling order can have some impact on communication performance when the size of the polling set is relatively large. We also plan to investigate along some of these directions.

5. IMPLEMENTATION

We have implemented the proposed design on our InfiniBand testbed, which consists of InfiniHost HCAs and an InfiniScale switch from Mellanox [17]. The InfiniHost offers a user-level programming interface called VAPI, which is based on InfiniBand Verbs.

Currently one of the most popular MPI implementations is MPICH [9] from Argonne National Laboratory. MPICH uses a layered approach in its design. The platform dependent part of MPICH is encapsulated by an interface called Abstract Device Interface, which allows MPICH to be ported to different communication architectures. Our MPI implementation on Mellanox InfiniHost cards is essentially an ADI2 (the second generation of Abstract Device Interface) implementation which uses VAPI as the underlying communication interface. Our implementation is also derived from MVICH [13], which is an ADI2 implementation for VIA. MVICH was developed in Lawrence Berkeley National Laboratory.

In our MPI implementation, many parameters such as the size of each RDMA buffer and the threshold from Eager protocol to Rendezvous protocol can be changed at run time. By default, we have chosen 2K bytes as both the size of RDMA buffers and the Eager to Rendezvous threshold value.

6. PERFORMANCE EVALUATION

In this section we present performance evaluation for our MPI design. Base MPI performance results are first given. Then we evaluate impact of using RDMA based design by comparing it with send/receive based design. We use microbenchmarks, collective communication tests as well as applications (NAS Parallel Benchmarks [18]) to carry out the comparison. Finally, we use simulation to study the impact of number of RDMA channels on RDMA polling performance.

6.1 Experimental setup

Our experimental testbed consists of a 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. The machines were 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 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system. The Mellanox InfiniHost HCA SDK build id is thca-x86-0.1.2-build-001. The adapter firmware build id is fw-23108-rel-1_17_0000-rc12-build-001. For the tests, we compiled with Intel(R) C++ and FORTRAN Compilers for 32-bit applications Version 6.0.1 Build 20020822Z.

6.2 MPI Base Performance

Figures 9 and 10 show the latency and bandwidth of our RDMA-based MPI implementation. We have achieved a 6.8 microseconds latency for small messages. The peak bandwidth is around 871 Million Bytes (831 Mega Bytes)/second. We have chosen 2K as the threshold for switching from Eager protocol to Rendezvous protocol. Table 2 compares these numbers with the results we got from Quadrics Elan3 cards and Myrinet LANai 2000 cards in the same cluster. (Please note that Myrinet and Quadrics cards use PCI-II 64x66 MHz interface while the InfiniHost HCAs use PCI-X 133 MHz interface.) From the table we see that our implementation performs quite well compared with Quadrics and Myrinet.

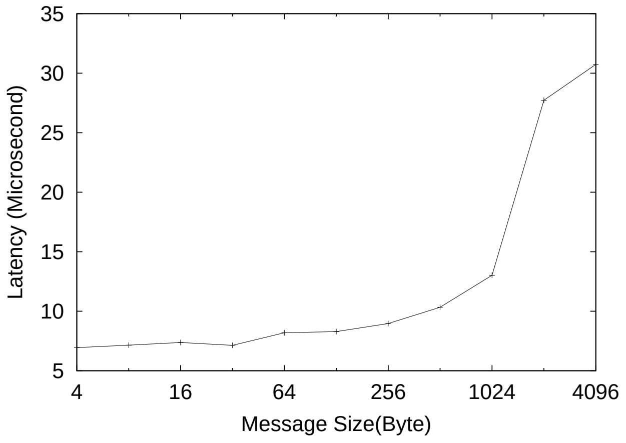

Figure 9: MPI Latency

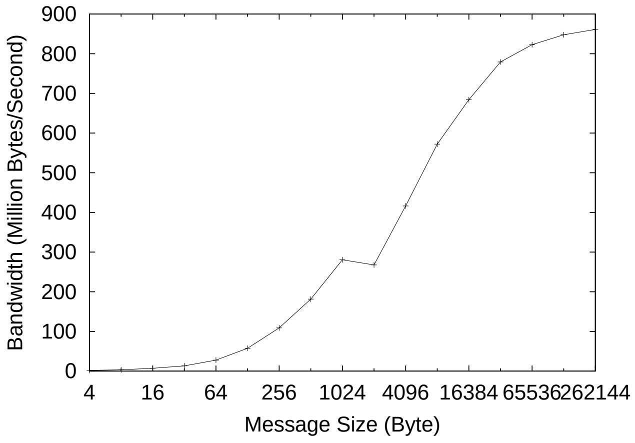

Figure 10: MPI Bandwidth

6.2.1 *Impact of Eager/Rendezvous Threshold*

In MPI applications, it is important to decide when to switch from Eager protocol to Rendezvous protocol. Usually this is determined by the size of the message. The advantage of Eager protocol is that the handshake process in Rendezvous protocol can be avoided. However, since there are extra copies in Eager protocol, it is not suitable for large messages. Another issue in using Eager protocol for large messages is that messages have to be divided into smaller chunks to fit into communication buffers.

By default, we have chosen 2K Bytes as the threshold to switch from Eager protocol to Rendezvous protocol. However, this value should be tuned in different platforms. Figure 11 shows the impact of different threshold values. Please note that the switch value for message sizes is actually a little less than the given value because in our implementation the switch value includes other overheads besides the message payload (such as the header).

We can see that using a larger threshold can lead to better latency or bandwidth for certain message sizes. To get a smooth transition in the latency graph, the threshold should be around 13K Bytes. However, as we have mentioned, increasing the threshold value also introduces more communication overhead because of message copying and fragmentation. Therefore, the best way to tune this value is probably to use real applications instead of using just the latency test alone.

Table 2: MPI Performance (Smallest Latency and Peak Bandwidth)

	Latency (us)	Bandwidth (MB/s)
This implementation	6.8	871
Quadrics	4.7	323
Myrinet/GM	7.3	242

6.3 Impact of RDMA-Based Design

In this section we show the improvements of our RDMA-based design by comparing it with the send/receive based design. In both cases, RDMA write is used for data transfer in Rendezvous protocol. Please note that since our testbed is small (8 nodes), essentially the RDMA polling set of each process contains all other processes.

6.3.1 *Micro-Benchmarks*

The latency test was carried out in a ping-pong fashion and repeated for 1000 times. In Figure 12, we can see that RDMA-based design improves MPI latency. For small messages the improvement is more than 2 microseconds, or 24% of the latency time. For large messages which go through the Rendezvous protocol, we can still reduce the latency by saving time for control messages. The improvement for large messages is more than 6 microseconds.

The bandwidth test was conducted by letting the sender push 100 consecutive messages to the receiver and wait for a reply. Figures 13 and 14 present the bandwidth comparison. It can be seen that RDMA-based design improves bandwidth for all message ranges. The impact is quite significant for small messages, whose performance improves by more than 104%.

LogP model for parallel computing was introduced in [6], which uses four parameters *delay, overhead, gap* and *processors* to describe a parallel machine. The overhead in communication can have significant impact on application performance, as shown by previous studies [16]. In Figure 15 we present the host overhead for the latency tests in Figure 12. We can see that the RDMA-based design can also reduce the host overhead for communication. For small messages, the RDMA-based approach can reduce the host overhead by up to 22%.

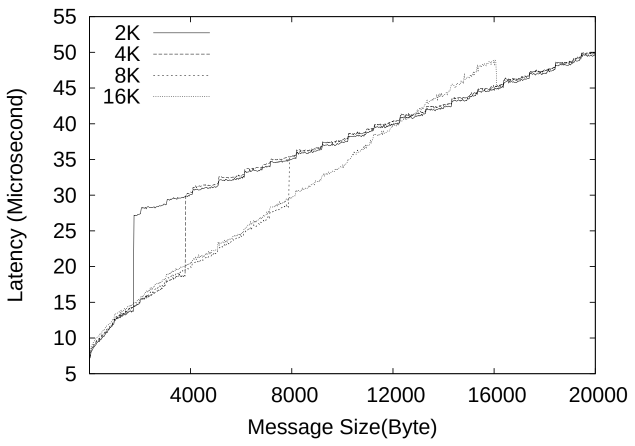

Figure 11: Impact of Eager/Rendezvous Threshold

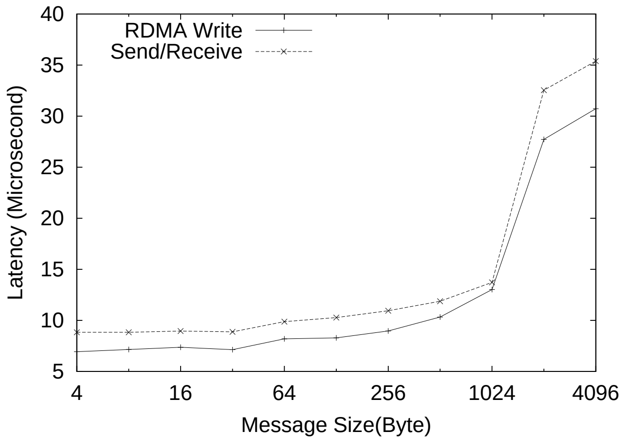

Figure 12: MPI Latency Comparison

6.3.2 MPI Latency Timing Breakdowns

To provide more insights into where time has been spent in our RDMA based design, we have obtained timing breakdowns for the latency test through instrumentation. Based on the implementation of MPICH, we have divided the latency time into seven components: sender MPI to ADI overhead (s_mpitoadi), sender ADI layer protocol overhead (s_protocol), sender RDMA posting time (s_post), InfiniBand RDMA transfer time (IBA), receiver completion overhead (r_completion), receiver ADI layer protocol overhead (r_protocol) and receiver ADI to MPI overhead (r_aditompi).

In Figure 16 we show the results for different message sizes. In each pair, the left bar represents the RDMA based design and the right bar represents the send/receive based design. We can see that with the RDMA based design, we have reduced the ADI layer protocol processing overhead, especially for the receiver. Another major improvement comes from the better performance of the underlying InfiniBand RDMA operations compared with send/receiver operations.

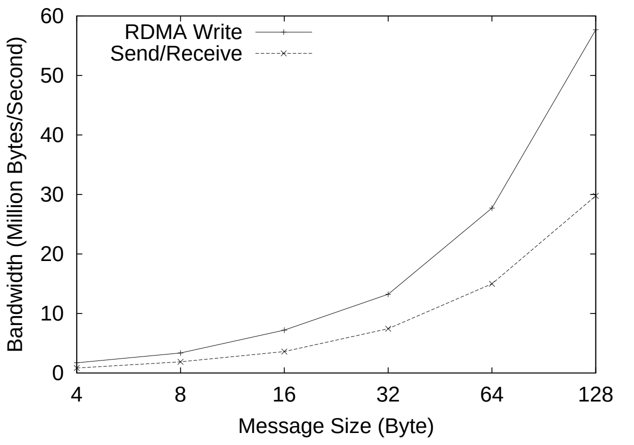

Figure 13: MPI Bandwidth Comparison (Small Messages)

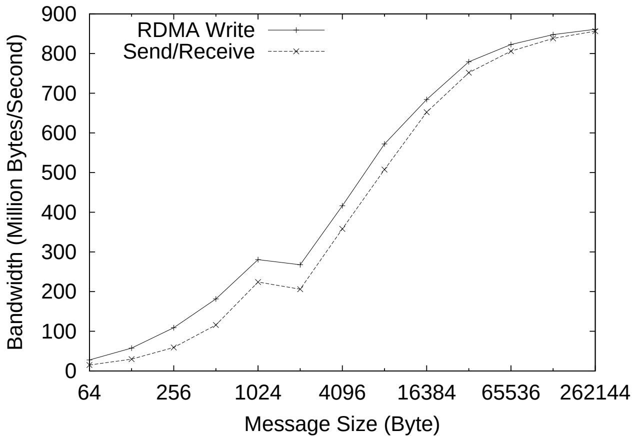

Figure 14: MPI Bandwidth Comparison

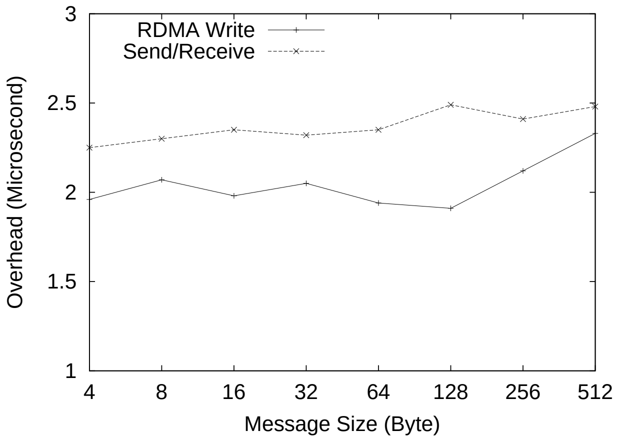

Figure 15: Host Overhead in Latency Test

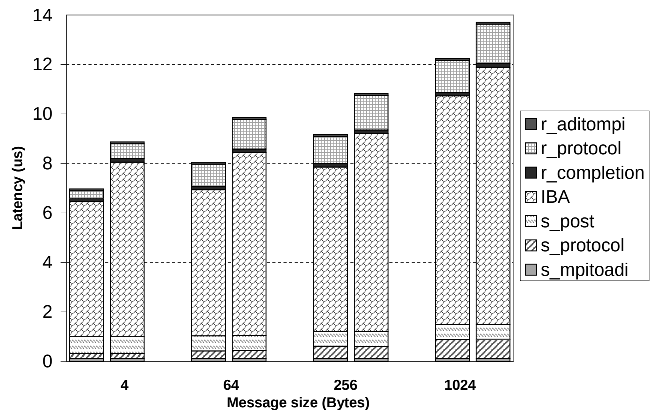

Figure 16: MPI Latency Timing Breakdowns

6.3.3 *Collective Communication and NAS Parallel Benchmarks*

We have also conducted tests using collective communication and NAS Parallel Benchmarks. For collective communication tests, we have chosen barrier, allreduce and broadcast, which are among the most frequently used operations in large scientific applications [23]. The test programs we have used for barrier and allreduce are the Pallas MPI Benchmarks [20].

In the barrier test shown in Figure 17, we change the number of nodes participating in the barrier and measure the barrier time. We can see that by using RDMA, the barrier latency can be reduced significantly. In the allreduce and broadcast tests shown in Figures 18 and 19, the number of processes is kept at 8 and we change the message size of the collective communication. From the figures we can see that RDMA-based design improves the performance of both allreduce and broadcast operations.

In Figures 20 and 21 we show the results for IS, MG, LU, CG, FT, SP, and BT programs from the NAS Parallel Benchmark Suite on 4 and 8 nodes. (Class A results are shown for 4 nodes and class B results are shown for 8 nodes.) SP and BT require the number of processes to be a square number. Therefore, their results are not shown for 8 nodes. Program IS uses mostly large messages and the improvement of the RDMA-based design is very small. For all other programs, the RDMA-based design brings improvements as high as 7% in overall application performance.

6.4 Impact of RDMA Channel Memory Usage

In this subsection, we study the impact of RDMA Channel memory usage on application performance. As we have mentioned, decreasing the number of RDMA buffers for each connection can reduce memory consumption of our RDMA based design and can potentially lead to better scalability. However, too few RDMA buffers may also result in degraded communication performance because the sender and the receiver are tightly coupled in communication.

In our MPI implementation, the default value we have chosen for the number of RDMA buffers for each connection is 100. However, this value can be easily changed based on the size and the communication pattern of different MPI applications. In Figure 22 we show the performance of class A NAS applications IS, CG, and MG running on 8 nodes when we change the number of RDMA buffers. We can see that

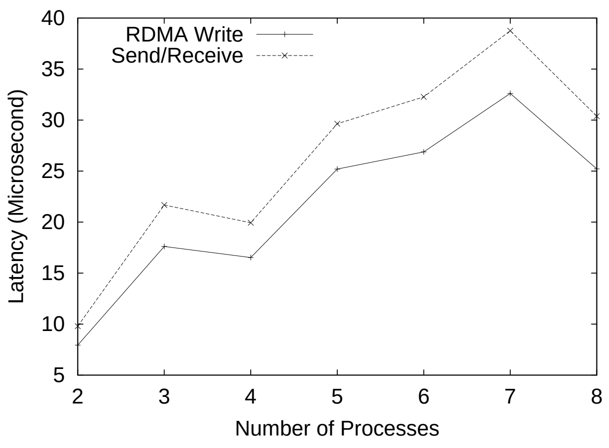

Figure 17: MPI Barrier Latency

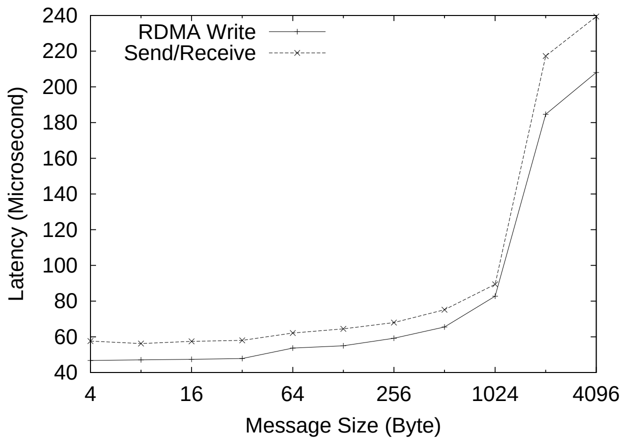

Figure 18: MPI Allreduce Latency (8 Nodes)

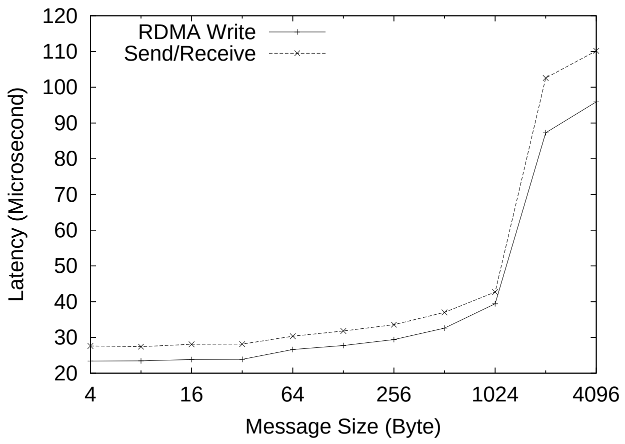

Figure 19: MPI Broadcast Latency (8 Nodes)

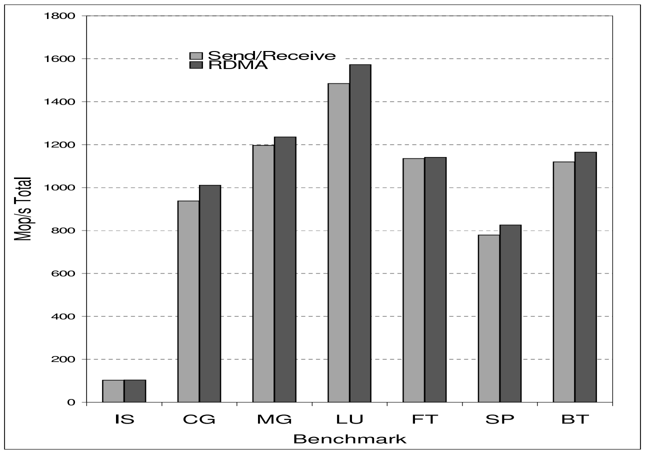

Figure 20: NAS Results on 4 Nodes (Class A)

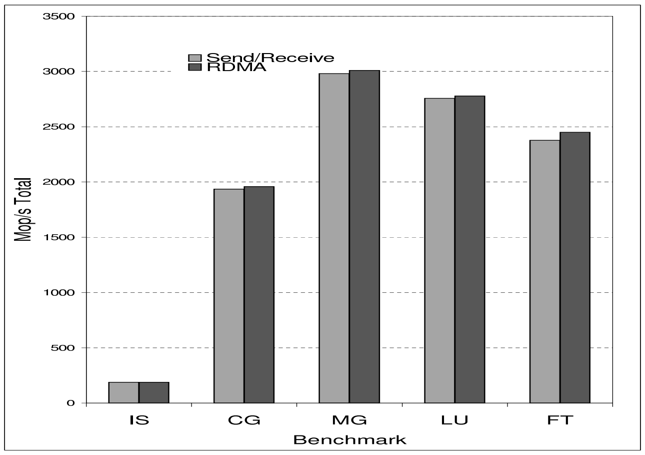

Figure 21: NAS Results on 8 Nodes (Class B)

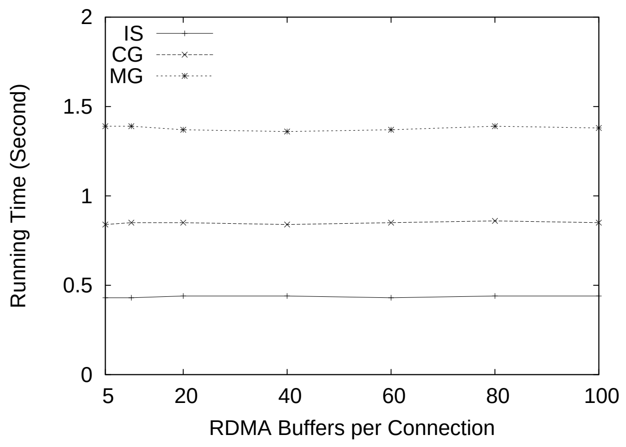

Figure 22: Effect of Changing the Number of RDMA Buffers for Each Connection

reducing the number of RDMA buffers does not degrade application performance. Even with only 5 RDMA buffers for each connection, the applications can still perform very well. We have observed similar patterns for other NAS applications. If these applications keep the same communication pattern, then only 20M Bytes memory is needed at every process for a 1024 process MPI application. This indicates that in these cases memory usage for RDMA channels will not pose any scalability problems for large scale MPI applications.

6.5 RDMA Channel Polling Time

In order to study the behavior of the RDMA-based design for large systems, we have simulated the polling time with respect to different numbers of connections in the RDMA polling set. Figure 23 shows the results. We can see that even though the polling time increases with the number of connections, the time to poll a connection is very small. Even with 128 connections, the polling time is only about 1.3 microseconds. This small polling time means that the size of an RDMA polling can be relatively large without degrading performance. For applications shown in Table 1, a polling set with 16 connections is enough even for 1024 processes. The polling time for 16 connections is only 0.14 microseconds. Thus, our proposed design demonstrates potential for being applied to large systems without performance degradation.

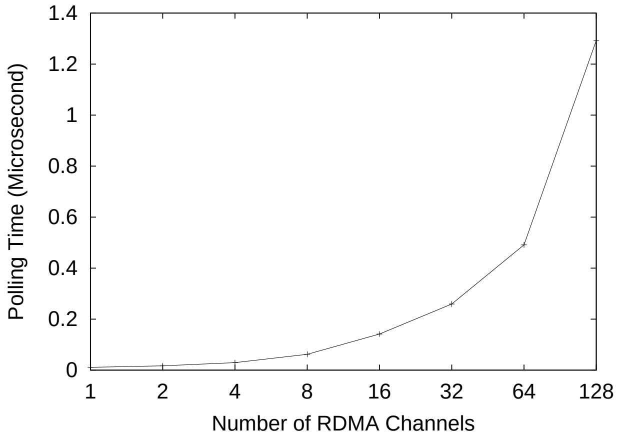

Figure 23: Polling Time of RDMA Channels

7. RELATED WORK

Being the *de facto* standard of writing parallel applications, MPI has been implemented for numerous interconnects, including those with remote memory access abilities [22, 2, 11, 4]. [2] relies on the active message interface offered by LAPI. [11] uses PIO for small messages. Work done in [4] implemented MPI for Cray T3D based on the SHMEM interface. [22] describes an MPI implementation over Sun Fire Link Interconnect, which is based on PIO. MPI over Sun Fire Link uses a sender-managed buffer scheme for transferring messages. In this approach, the sender can choose any buffer at the receiver side for doing remote write. To let the receiver know where the data has been written, another PIO is used to write the buffer address to a pre-specified location. This extra PIO has very little overhead. However, the RDMA operations in InfiniBand Architecture have larger overhead. Therefore, one of our objectives is to use as few RDMA operations as possible. Another difference is that InfiniBand offers both channel and memory semantics and we have shown that it is possible to combine them to achieve scalability. However, none of the existing implementation has information regarding the use of RDMA operations for small data messages and control messages, nor are the scalability issues in RDMA discussed in these references.

RDMA operations have been used to implement MPI collective operations. Work in [21] focuses on how to construct efficient algorithms to implement collective operations by using RDMA operations. Our work can be used in conjunction with their work to efficiently transfer short data messages and control messages.

RDMA operations have also been used to design communication subsystems for databases and file systems [27, 15]. These studies do not address the issue of using RDMA for control messages. [5] evaluated different communication schemes for implementing a web server on a cluster connected by VIA. Some of their schemes use RDMA write for transferring flow control messages and file data. However, their schemes differ from ours in that they have used a sender-managed scheme which is similar to [22].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a new design of MPI over InfiniBand which brings the benefit of RDMA to not only large messages, but also small and control messages. We have proposed designs to achieve better scalability by exploiting application communication pattern and combining send/receive operations with RDMA operations. Our performance evaluation at the MPI level shows that for small messages, our RDMA-based design can reduce the latency by 24%, increase the bandwidth by over 104%, and reduce the host overhead by up to 22%. For large messages, we improve performance by reducing the time for transferring control messages. We have also shown that our new design also benefits MPI collective communication and NAS Parallel Benchmarks.

There are several directions for our future work. First, we would like to expand our current work by evaluating it with larger testbeds and more applications. We also plan to use similar techniques for designing other communication subsystems such as those in cluster-based servers and file systems. We are also working on taking advantage of other InfiniBand features, such as QoS [1], multicast, different classes of services and atomic operations to improve MPI performance. Another direction we are currently pursuing is to design new RDMA-based algorithms and protocols to improve the performance of MPI collective communication.

Software Availability

Our implementation of MPI over InfiniBand described in this paper is publicly available. It can be downloaded from http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html.

Acknowledgments

We would like to thank Dr. Pete Wyckoff from Ohio Supercomputer Center for valuable discussions. We thank Sushmitha Kini, Darius Buntinas, Weikuan Yu and Balasubraman Chandrasekaran for their help with the experiments. Our appreciation is also extended to Jeff Kirk, Ezra Silvera and Kevin Deierling from Mellanox Technologies for their insight and technical support on their InfiniBand hardware and software.

9. REFERENCES

[1] F. J. Alfaro, J. L. Sanchez, J. Duato, and C. R. Das. A Strategy to Compute the Infiniband Arbitration Tables. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[2] M. Banikazemi, R. K. Govindaraju, R. Blackmore, and D. K. Panda. MPI-LAPI: An Efficeint Implementation of MPI for IBM RS/6000 SP Systems. *IEEE Transactions on Parallel and Distributed Systems*, pages 1081–1093, October 2001.

[3] M. Blumrich, C. Dubnicki, E. W. Felten, K. Li, and M. R. Mesarina. Virtual-Memory-Mapped Network Interfaces. In *IEEE Micro*, pages 21–28, Feb. 1995.

[4] R. Brightwell and A. Skjellum. MPICH on the T3D: A Case Study of High Performance Message Passing. In *1996 MPI Developers Conference*, July 1996.

[5] E. V. Carrera, S. Rao, L. Iftode, and R. Bianchini. User-Level Communication in Cluster-Based Servers. In *Proceedings of the Eighth Symposium on High-Performance Architecture (HPCA'02)*, February 2002.

[6] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Principles Practice of Parallel Programming*, pages 1–12, 1993.

[7] R. Dimitrov and A. Skjellum. An Efficient MPI Implementation for Virtual Interface (VI) Architecture-Enabled Cluster Computing. http://www.mpi-softtech.com/publications/, 1998.

[8] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, pages 66–76, March/April 1998.

[9] 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.

[10] H. Tezuka and F. O'Carroll and A. Hori and Y. Ishikawa. Pin-down Cache: A Virtual Memory Management Technique for Zero-copy Communication. In Proceedings of 12th IPPS.

[11] P. Husbands and J. C. Hoe. MPI-StarT: Delivering Network Performance to Numerical Applications. In *Proceedings of the Supercomputing*, 1998.

[12] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.0, October 24 2000.

[13] Lawrence Berkeley National Laboratory. MVICH: MPI for Virtual Interface Architecture. http://www.nersc.gov/research/FTG/mvich/index.html, August 2001.

[14] J. Liu, J. Wu, S. P. Kinis, D. Buntinas, W. Yu, B. Chandrasekaran, R. Noronha, P. Wyckoff, and D. K. Panda. MPI over InfiniBand: Early Experiences. Technical Report, OSU-CISRC-10/02-TR25, Computer and Information Science, the Ohio State University, January 2003.

[15] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of USENIX 2002 Annual Technical Conference, Monterey, CA*, pages 1–14, June 2002.

[16] R. Martin, A. Vahdat, D. Culler, and T. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proceedings of the International Symposium on Computer Architecture*, 1997.

[17] Mellanox Technologies. Mellanox InfiniBand InfiniHost Adapters, July 2002.

[18] NASA. NAS Parallel Benchmarks.

[19] S. Pakin, M. Lauria, and A. Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM). In *Proceedings of the Supercomputing*, 1995.

[20] Pallas. Pallas MPI Benchmarks. http://www.pallas.com/e/products/pmb/.

[21] R. Gupta, P. Balaji, D. K. Panda, and J. Nieplocha. Efficient Collective Operations using Remote Memory Operations on VIA-Based Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '03)*, April 2003.

[22] S. J. Sistare and C. J. Jackson. Ultra-High Performance Communication with MPI and the Sun Fire Link Interconnect. In *Proceedings of the Supercomputing*, 2002.

[23] J. S. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '02)*, April 2002.

[24] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: A User-level Network Interface for Parallel and Distributed Computing. In *ACM Symposium on Operating Systems Principles*, 1995.

[25] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *International Symposium on Computer Architecture*, pages 256–266, 1992.

[26] J. Wu, J. Liu, P. Wyckoff, and D. K. Panda. Impact of On-Demand Connection Management in MPI over VIA. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[27] Y. Zhou, A. Bilas, S. Jagannathan, C. Dubnicki, J. F. Philbin, and K. Li. Expericences with VI Communication for Database Storage. In *In Proceedings of International Symposium on Computer Architecture'02*, 2002.