Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand *

Jiuxing Liu Dhabaleswar K. Panda

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

Abstract

In this paper, we present a detailed study of how to design efficient and scalable flow control mechanisms in MPI over the InfiniBand Architecture. Two of the central issues in flow control are performance and scalability in terms of buffer usage. We propose three different flow control schemes (hardware-based, user-level static and user-level dynamic) and describe their respective design issues. We have implemented all three schemes in our MPI implementation over InfiniBand and conducted performance evaluation using both micro-benchmarks and the NAS Parallel Benchmarks. Our performance analysis shows that in our testbed, most NAS applications only require a very small number of pre-posted buffers for every connection to achieve good performance. We also show that the user-level dynamic scheme can achieve both performance and buffer efficiency by adapting itself according to the application communication pattern. These results have significant impact in designing large-scale clusters (in the order of 1,000 to 10,000 nodes) with InfiniBand.

1 Introduction

During the last decade, the research and industry communities have been proposing and implementing user-level communication systems to address some of the problems associated with legacy networking protocols [22, 20, 3]. The Virtual Interface Architecture (VIA) [8] was proposed to standardize these efforts. More recently, InfiniBand Architecture [11] has been introduced which combines I/O with Inter-Processor Communication (IPC).

In the area of high performance computing, MPI has been the *de facto* standard for writing parallel applications. Although InfiniBand was initially proposed as a generic interconnect for inter-processor communication and I/O, its rich feature set, high performance and scalability make it also attractive as a communication layer for high performance computing. Currently, there are multiple MPI implementations over InfiniBand publicly available [19, 13, 18, 21].

One of the key issues in designing MPI over InfiniBand is flow control. Since current MPI implementations are based on InfiniBand Reliable Connection (RC) service, multiple receive buffers have to be posted for each connection in order for the sender to have multiple outstanding messages. However, if the sender sends too fast, these buffers can be exhausted. The flow control mechanism is to prevent a fast sender from overwhelming a slow receiver and exhausting its resources such as buffer space in this case. Flow control is an important issue in MPI design because it affects both the performance and the scalability of an MPI implementation.

Two central issues in flow control are performance and scalability in terms of buffer usage. In this paper, we propose three flow control schemes: *hardware-based*, *user-level static* and *user-level dynamic* to address these issues. All three schemes are implemented in our MPI implementation over InfiniBand [14]. We have evaluated them using both micro-benchmarks and the NAS Parallel Benchmarks [17]. Performance analysis shows that for most NAS applications, only a very small number of pre-posted buffers are required for every connection in order to achieve good performance. For those applications that need more buffers, our results show that the user-level dynamic scheme can achieve both performance and buffer efficiency by adapting itself according to the application communication pattern.

*This research is supported in part by Department of Energy's Grant #DE-FC02-01ER25506, a grant from Sandia National Laboratory, a grant from Intel Corporation, and National Science Foundation's grants #CCR-0204429 and #CCR-0311542.

The main contributions of this paper are:

1. We have proposed three different flow control schemes in MPI over InfiniBand (hardware-based, user-level static and user-level dynamic). We provide detailed discussions about the design issues in these schemes and their potential for achieving scalability with good performance.

2. We have implemented all three scheme in our MPI implementation over InfiniBand.

3. We have evaluated different schemes in terms of runtime overhead and buffer efficiency using both microbenchmarks and NAS Parallel Benchmarks. We have also studied the impact of buffer space on application performance, an issue which is important to the scalability of an MPI implementation.

The rest of the paper is organized as follows: In Section 2, we present an overview of the InfiniBand Architecture. In Section 3, we describe our MPI implementation over InfiniBand and the flow control issue associated with it. We discuss designs and implementations of different flow control schemes in Section 4 and Section 5, respectively. We present performance evaluation in Section 6. Related work is discussed in Section 7. In Section 8, we conclude and briefly mention some of the future directions.

2 Infi niBand Overview

The InfiniBand Architecture (IBA) [11] defines a high-performance network architecture 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. They generate and consume IBA packets. There are two kinds of channel adapters: Host Channel Adapter (HCA) and Target Channel Adapter (TCA). HCAs are connected to processing nodes. Their semantic interface to consumers is specified in the form of InfiniBand Verbs. TCAs connect I/O nodes to the fabric. Their interface to consumers are usually implementation specific and thus not defined in the InfiniBand specification.

2.1 Communication Model

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 the InfiniBand Architecture consists of 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 queue pair. 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. In InfiniBand, before a buffer can be used for communication, it must be registered. After communication, the buffer can be de-registered.

InfiniBand supports multiple transport services. In the current hardware, Reliable Connection (RC) and Unreliable Datagram (UD) services are implemented. In the Reliable Connection service, a connection must be set up between the queue pairs at the two parties before they can communicate. InfiniBand hardware guarantees reliability of the communication for the Reliable Connection service.

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.

In memory semantics, RDMA write and RDMA read operations are used instead of send and receive operations. These operations are one-sided and transparent to the software layer at the remote side.

2.2 Infi niBand Flow Control Mechanisms

InfiniBand provides flow control mechanisms at different level. At the link level, it defines different link rates such as 1x (2.5Gbps), 4x (10Gbps) and 12x (30Gbps). To prevent a high speed link from overrunning a low speed link, InfiniBand provides a *Static Rate Control* mechanism at the link level.

InfiniBand also has an *End-to-End Flow Control* mechanism for Reliable Connection service. When a sender is sending too fast and receive buffers are running out, the

sender HCA will slow down. If a message arrives and there is no receive buffer posted yet, the receiver will issue a *Receiver-Not-Ready* message (RNR NAK). The sender HCA will wait for a time-out and retry the send operation until corresponding receive operation is posted. The upper layer software can control this behavior by specifying parameters such as retry count and retry time-out for different queue pairs. However, these parameters must be set during initialization time and cannot be changed afterward.

3 MPI over InfiniBand

In this section, we first present a design of MPI over InfiniBand, which uses send/receive operations for transferring small data and control messages and RDMA write operations for transferring large data messages. Then we introduce the flow control issues associated with this design.

3.1 Basic MPI Design

MPI defines four different communication modes: *Standard*, *Synchronous*, *Buffered*, and *Ready*. 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 current 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 1 shows examples of typical Eager and Rendezvous protocols.

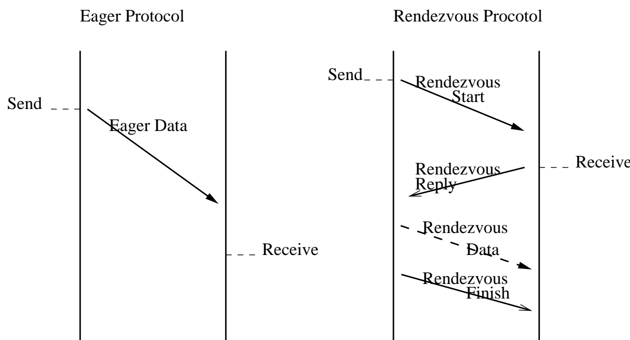

Figure 1. MPI Eager and Rendezvous Protocols

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 using RDMA write. Similar approaches have been used for implementing MPI over different interconnects [12, 7, 1].

In our basic design [14], InfiniBand Reliable Connection transport service is used. Eager protocol messages and control messages in Rendezvous protocol 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 each 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 reported through the CQ.

The 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 the sent out. 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 buffer is then re-posted to the receive queue. The communication of control messages in Rendezvous protocol also uses this buffer pool. In Rendezvous protocol, user data buffers are pinned on-the-fly and transferred directly using RDMA write operations. The buffer pinning and unpinning overheads can be reduced by using the pin-down cache technique [10].

3.2 Flow Control in MPI over InfiniBand

In Figure 1 we have shown different type of messages in Eager and Rendezvous protocols. Among these messages, *Eager Data* and *Rendezvous Start* messages are sent to the receiver regardless of its current state. Therefore, these messages are *unexpected* with respect to the receiver. Since the sender can potentially initiate a large number of send operations in MPI, it is possible that the receiver can be overwhelmed by too many unexpected messages because each of these messages consumes resources such as buffer space at the receiver side. This issue is even more important for InfiniBand because communication buffers must be pinned down and they cannot be swapped out during communication.

To accommodate unexpected messages, every process can pre-post a number of receiver buffers for each connection. Every message will be received into one of these buffer. After the receiver finishes processing a buffer, it immediately re-posts the buffer. As long as the number of outstanding unexpected messages for a connection is less than

the number of pre-posted buffer, these messages can always be received safely. However, if the number of outstanding unexpected messages is too large, a flow control mechanism needs to be implemented to stall or slow down the sender so that the receiver can keep up.

Thus, there are two important problems in flow control. First, we need to study the impact of the number of pre-posted receiver buffers. Using too many buffers will adversely affect application performance and limit the scalability of the MPI implementation. However, if the number is too small, senders may have to stall or slow down frequently and wait for the remote process to re-post the buffers. As a result, the sender and the receiver become tightly coupled in communication, leading to degraded performance. Ideally, the number should be determined by application communication pattern to achieve both performance and scalability. The second issue is what mechanism we use to stall or slow down the sender when the receiver cannot keep up. This mechanism should be effective and have negligible run-time overhead during normal communication.

4 Flow Control Design Schemes

We propose several designs to address the flow control issue. Flow control can be handled in the MPI implementation. In this case, we call it a *user-level scheme*. However, since InfiniBand itself provides end-to-end flow control, an alternative is to let InfiniBand hardware handle this task. We call this a *hardware-based scheme*. Flow control schemes can also be classified by the way they choose the number of pre-posted buffers. In a *static scheme*, this number is determined at compilation or initialization time and remains unchanged during execution of the application. On the other hand, in a *dynamic scheme*, this number can be changed during program execution. Next, we discuss these design alternatives in detail.

4.1 Hardware-Based Flow Control

In hardware-based schemes, there is no flow control at the MPI level. All outgoing messages are submitted immediately to the send queue. If too many send operations are posted, the HCA at the sender side will reduce its sending rate because of the InfiniBand end-to-end flow control. At the receiver side, when there is no posted receive for an incoming message, this message will be dropped and RNR Nak will be issued. The sender will then wait for a time-out and re-transmit the message. To ensure reliability at the MPI level, the retry count can be set to infinite. Thus eventually the message will be delivered to the receiver side when the receive buffer is posted.

One of the advantages in using hardware-based flow control is that it incurs almost no run-time overhead in normal communication when there are enough pre-posted receiver buffers. This is because there is no need to keep track of flow control information in the MPI implementation. This also means that flow control processing can be done regardless of the communication progress of applications. Therefore, hardware-based schemes also achieves better "application bypass" [4, 5]. However, InfiniBand provides very little flexibility to adjust the behavior of hardware based flow control. Since the flow control algorithm used by InfiniBand may not be optimal for every MPI application, this lack of flexibility may result in performance degradation for some applications. Further, the end-to-end flow control and transmission retries are largely transparent to the software layer and there is no information feedback for the MPI implementation to adjust its behavior. The lack of information feedback makes it very hard to implement dynamic flow control schemes in which the MPI implementation can adjust the number of pre-posted buffers for each connection based on application communication pattern.

4.2 User-Level Static Flow Control

In this subsection we discuss how to implement user-level flow control at the MPI level. First we will describe static schemes currently used in [12] and [14].

The basic idea of user-level flow control is to use a credit-based scheme. During MPI initialization, a fixed number of receive buffers are pre-posted for each connection. Initially, the number of credits for each sender is equal to the number of pre-posted buffers at the corresponding receiver. Whenever a sender sends out a message that will consume a receiver buffer, its credit count will be decremented. When the credit count reaches zero, the sender can no longer post send operations that consume credits. Instead, these operations will be stored in a *backlog queue*. The operations in the backlog queue will be processed later in FIFO order when credits are available.

At the receiver side, the receiver will re-post a receive buffer after it has finished processing it. The credit count for the corresponding sender will then be incremented. However, since this information about new credits is only available at the receiver side, we must have some kind of mechanism to transfer it to the sender side. Two methods can be used for this purpose: *piggybacking* and *explicit credit messages*. To use piggybacking, we add a credit information field to each message. An MPI process can use this field to notify the other side about credit availability. If the communication pattern is symmetric, each sender will get credit information updates frequently and be able to make communication progress. Explicit credit messages can be used when the communication pattern is asymmetric. When a process has accumulated a certain number of credits and there is still no message sent by the MPI layer to the other

side, a special credit message can be sent to transfer the credit information. In the MPI implementation, small messages are usually transferred using Eager protocol. However, when there are no credits, only Rendezvous protocol is used. Because of the handshaking process in Rendezvous protocol, credit information can be exchanged through piggybacking, which can speed up the processing of the send operations in the backlog queue.

Because of possible deadlock situation, explicit credit messages must be used carefully. In [12] and [14], these messages themselves will consume credits because they are implemented using send operations. To prevent deadlock, we proposed an *optimistic scheme* for credit messages. Basically, we do not impose flow control for explicit credit messages. Thus, explicit credit messages are not subject to user-level flow control. These messages are always posted directly without going through the backlog queue even though no credit is available. In this case, the hardware-level flow control mechanism will ensure that the message will be delivered. Since credit messages can always be sent, deadlock will not happen.

Using user-level flow control requires the MPI implementation to manage credit information and take appropriate actions. Therefore, it may have larger run-time overhead than hardware-based schemes. The use of explicit credit messages may increase network traffic in some cases. (We should note that hardware-based schemes may also increase network traffic because of NAK and re-transmission.) However, these overheads can be reduced by an optimized implementation. Another disadvantage of user-level schemes is that flow control processing relies on communication progress. Therefore, it achieves less "application bypass" compared with hardware-based schemes. The major advantage of user-level flow control is that various information regarding flow control is available to the MPI layer. Based on this information, an MPI implementation can adjust its behavior to achieve better performance and scalability. Next, we will discuss a dynamic user-level flow control scheme that takes advantage of this information.

4.3 User-Level Dynamic Flow Control

To achieve both performance and scalability, we propose a dynamic user-level flow control scheme. This scheme uses credit-based flow control at the MPI level, which is similar to the static scheme. The difference is that each connection starts with a small number of pre-posted buffers. During program execution, the number of pre-posted buffers can be gradually increased based on the communication pattern using a feedback-based control mechanism. In this scheme, two important issues must be addressed:

- How to provide feedback information?
- What to do when feedback information is received?

The feedback mechanism should notify the receiver when more pre-posted buffers are needed. We notice that if there are not enough credits, a message will enter the backlog queue and be processed later. Therefore, this information can be used to provide feedback. We add a field to each message indicating whether it has gone through the backlog. When a process receives a message that has gone through the backlog queue, it takes action to increase the number of pre-posted buffers for the corresponding sender. The increase can be linear or exponential depending on the application. If communication pattern changes are relatively slow compared with the time to increase the number of pre-posted buffers, this mechanism can achieve both good performance and buffer efficiency.

In addition to increasing the number of buffers, a dynamic scheme can also decrease the number of buffers when the application no longer needs so many buffers. This may be beneficial to long-running, multi-phase MPI applications whose communication pattern changes in different phases. Currently we only allow increasing the number of buffers. We plan to investigate more along this direction in the future.

5 Implementation

Currently one of the most popular MPI implementations is MPICH [9]. The platform dependent part of MPICH is encapsulated by an interface called Abstract Device Interface (ADI), which allows MPICH to be ported to different communication architectures. Our MPI implementation over InfiniBand is essentially an ADI2 (the second generation of Abstract Device Interface) implementation which uses InfiniBand as the underlying communication interface. Our implementation [14] is also derived from MVICH [12], which is an ADI2 implementation for VIA.

Our original MPI implementation over InfiniBand [14] uses a user-level static flow control scheme. We incorporated our optimistic scheme for deadlock avoidance. We have also implemented the hardware-based scheme and the user-level dynamic scheme. In the user-level dynamic scheme, linear increasing is used when more pre-posted buffers are needed. In all implementations, the size of each pre-posted buffer is 2 KBytes.

6 Performance Evaluation

In this section, we present performance evaluation of different flow control schemes using both micro-benchmarks and applications. The micro-benchmarks are latency and bandwidth tests. The applications we use are the NAS

Parallel Benchmarks [17]. In the performance evaluation, we concentrate on two aspects of different flow control schemes: normal condition (with plenty of pre-posted buffers or credits) and flow control condition (when the number of outstanding messages exceeds the number of pre-posted buffers or there are not enough credits). Another issue we are interested in is how many pre-posted buffers are generally needed by applications in order to achieve best performance.

6.1 Experimental Testbed

Our experimental testbed consists of a cluster system of 8 SuperMicro SUPER P4DL6 nodes. Each node has dual Intel Xeon 2.40 GHz processors with 512K L2 cache and 400 MHz front side bus. The machines are connected by Mellanox InfiniHost MT23108 DualPort 4X HCA adapters through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch [15]. The HCA adapters work under the PCI-X 64-bit 133MHz interfaces. We used the Linux Red Hat 7.2 operating system. The compilers we used for the tests are Intel(R) C++ and FORTRAN Compilers for 32-bit applications Version 6.0.1 Build 20020822Z.

6.2 Micro-Benchmarks

6.2.1 Latency

The latency test is carried out in a ping-pong fashion. The sender sends a message of a certain data size to the receiver and waits for a reply of the same size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) are used in the tests.

In the latency test, the communication pattern is very symmetric. Since the sender and the receiver send back a message only after processing the previous one, there are always enough receive buffers posted at both sides. For user-level schemes, the credit information is always transferred in time through piggybacking. Therefore, this test shows how different schemes perform under normal conditions.

In the latency test, the hardware-level scheme has the least overhead because there is no need to keep track of information related to flow control. However, from Figure 2 we can see that this bookkeeping overhead is negligible and all three schemes perform comparably[1].

[1] In this paper, we have based our study on the Send/Recv based MPI implementation, which does not use RDMA for small and control messages. Our RDMA-based MPI implementation described in [13] achieves a latency of 6.8μs.

6.2.2 Bandwidth

The bandwidth tests are carried out by having the sender send out a number of back-to-back messages to the receiver and then waiting for a reply from the receiver. The number of back-to-back messages is referred to as *window size*. The receiver sends the reply only after it has received all messages. The above procedure is repeated multiple times and the bandwidth is calculated based on the elapsed time and the number of bytes sent by the sender. We use two different versions of bandwidth tests. In the blocking version, MPI_Send and MPI_Recv functions are used. MPI_Isend and MPI_Irecv are used in the non-blocking version.

In the first group of our tests, we have chosen a fixed message size (4 bytes). The tests are conducted for both blocking version and non-blocking version. The numbers of pre-posted buffers we have chosen for the tests are 10 and 100. Different results are obtained by varying the window size of the bandwidth tests.

Figures 3 and 4 show the results with 100 pre-posted buffers for small message transfers. In these tests the window size does not exceed the number of pre-posted buffers. We can see that with enough buffers or credits, all three schemes perform comparably for both blocking and non-blocking version.

Figures 5 and 6 show the results with only 10 pre-posted buffers. We can observe that when the window size exceeds the number of pre-posted buffers, the user-level dynamic scheme achieves the best performance because it is able to adapt to the communication pattern and increase the number of buffers. On the other hand, user-level static scheme performs the worst because the communication is stalled when there are not enough credits. We also notice that for user-level schemes, blocking version of bandwidth tests achieve better performance. This is because in user-level schemes, when there is no credit available, Rendezvous protocol will be used even for small messages. In the blocking tests, the sender waits for the send operation to finish and therefore is able to get more credits through the handshaking procedure of Rendezvous protocol.

Figures 7 and 8 show the results with 10 pre-posted buffers for large messages (32K bytes). Since large messages always go through Rendezvous protocol, the communication pattern in these tests becomes more symmetric because of the handshaking procedure. As a result, all three schemes are able to perform well even with less number of buffers. The non-blocking version performs much better than the blocking version because it achieves better communication overlap.

6.3 NAS Parallel Benchmarks

To better understand the impact of different flow control schemes on application performance, we have conducted

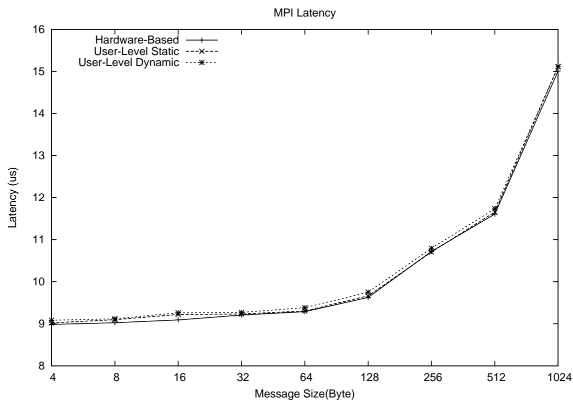

Figure 2. MPI Latency

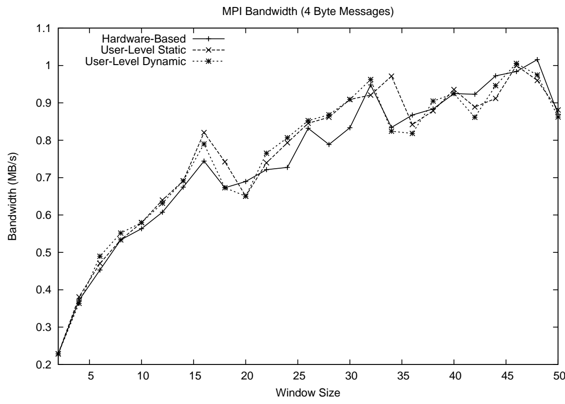

Figure 3. MPI Bandwidth (Pre-Post = 100, Blocking)

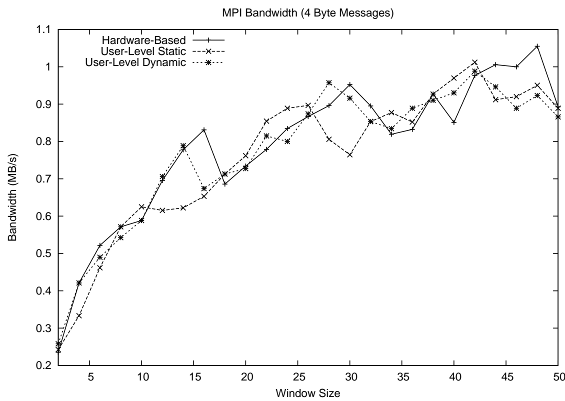

Figure 4. MPI Bandwidth (Pre-Post = 100, Non-Blocking)

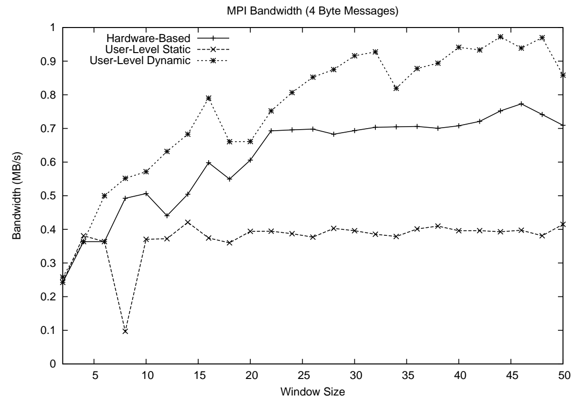

Figure 5. MPI Bandwidth (Pre-Post = 10, Blocking)

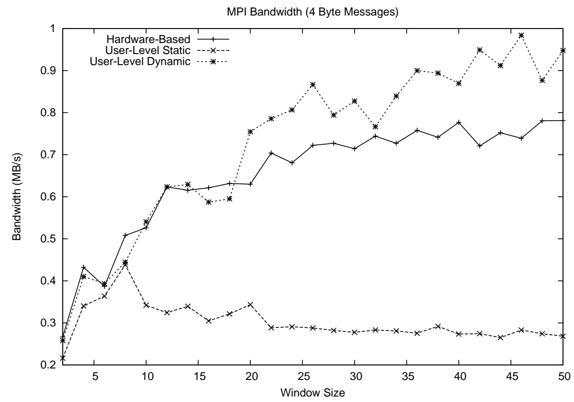

Figure 6. MPI Bandwidth (Pre-Post = 10, Non-Blocking)

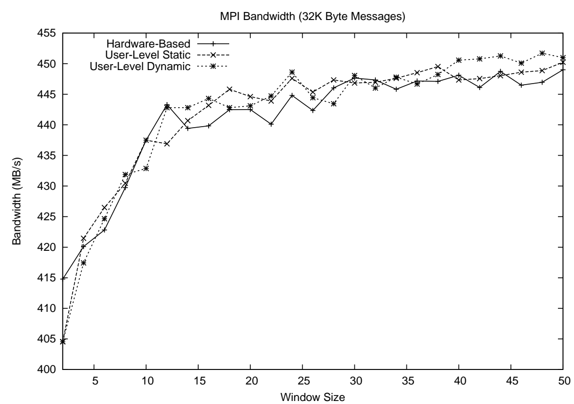

Figure 7. MPI Bandwidth (Pre-Post = 10, Blocking)

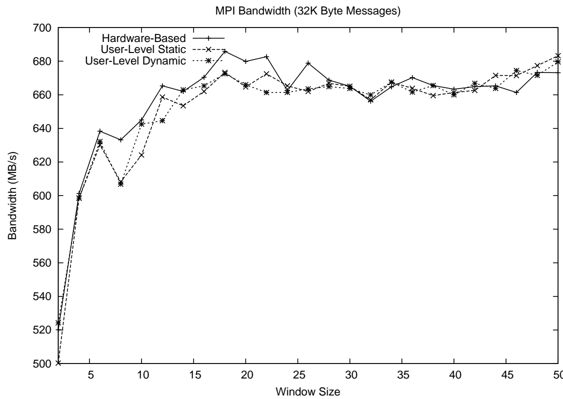

Figure 8. MPI Bandwidth (Pre-Post = 10, Non-Blocking)

experiments using the NAS Parallel Benchmarks (Class A). IS, FT, LU, CG and MG tests were carried out using 8 processes on 8 nodes. Since SP and BT tests require the number of processes to be a square number, they were conducted using 16 processes on 8 nodes.

6.3.1 Impact of Flow Control on Normal Communication

First, we study the impact of different flow control schemes on normal communication where there are always enough pre-posted buffers or credits. We carried out the experiments with 100 pre-posted buffers, which are more than any of the application will need. The results are shown in Figure 9. We notice that the three flow control schemes perform comparably for almost all the applications, with at most 2%–3% difference due to random fluctuation. One exception is the LU application. (There are also some discrepancies in the running time of BT. We are currently investigating this issue.) For LU, the hardware-based scheme is the best, which outperforms both user-level schemes by around 5%–6%. The reason why user-level schemes performs worse is that they use explicit credit messages. If the application communication pattern is very asymmetric, these messages have to be generated frequently in order to transfer credit information and the performance will be degraded. Table 1 shows the average number of explicit credit messages (ECM) for each connection at each process and the total number of messages (including data and control messages). We can see that for LU, explicit credit messages make up for a significant percentage of the total number of messages (18%). However, there are almost no explicit flow control messages for other applications. We should also note that the number of explicit credit messages depends on a threshold credit value, which suppresses any explicit credit messages if the number of credit to be transferred is below the threshold. Currently we use a relatively small threshold value of 5. Performance can be improved by increasing this value for LU.

6.3.2 Impact of Number of Pre-Posted Buffers

As we have discussed, the number of pre-posted buffers has significant impact on the scalability of applications. In this subsection, we consider an extreme case where there is only one pre-posted buffer for every connection at each process. Figure 10 shows the percentage of performance drop when we change the pre-post value from 100 to 1. This case can serve as an "upper bound" of the impact of changing the number of pre-posted buffers.

One surprising findings from Figure 10 is that most applications perform quite well even in this extreme condition. For IS, FT, SP and BT, the maximum performance degradation for all three schemes is only 2%. For the hardware-based scheme, performance drops significantly for LU and MG. This drop is due to the large number of time-out and re-transmission happening at the hardware level. For the user-level static scheme, the largest performance drops are for LU (13%) and CG (6%). Since the user-level dynamic scheme is able to adapt its behavior according to the application communication pattern, there is almost no performance degradation. In Table 2, we show the maximum number of posted buffers for every connection at every process in the user-level dynamic scheme after program execution. We can see that for all applications except LU, only a very small number of buffers (no more than 7) are needed for each connection. Therefore, the user-level dynamic flow control scheme can potentially achieve both performance and buffer efficiency. If this communication pattern remains unchanged for large number of processes, buffer space will not be the limitation of scalability. We plan to investigate this direction further in the future on large-scale clusters.

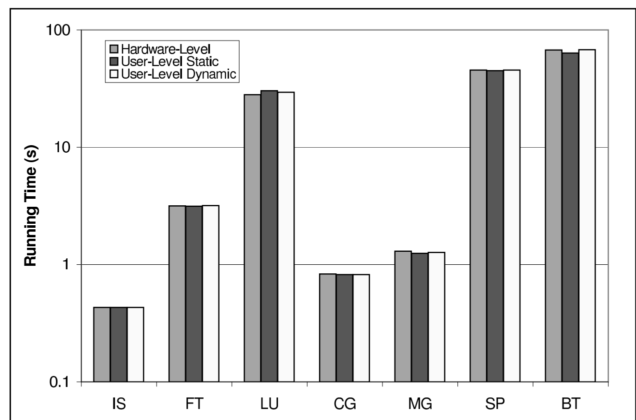

Figure 9. NAS Benchmarks (Pre-Post = 100)

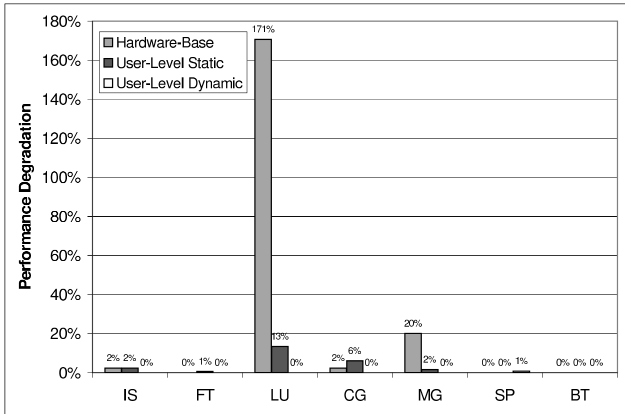

Figure 10. NAS Benchmarks (Performance Degradation from Pre-Post=100 to Pre-Post=1)

Table 1. Explicit Credit Messages for User-Level Static Scheme

App	# ECM Msg	# Total Msg
IS	0	383
FT	0	193
LU	9002	48805
CG	0	4202
MG	1	1595
BT	0	28913
SP	0	14531

Table 2. Maximum Number of Posted Buffers for User-Level Dynamic Scheme

App	# Buffer
IS	4
FT	4
LU	63
CG	3
MG	6
BT	7
SP	7

7 Related Work

Flow control is an important issue in cluster computing and has been studied in the literature. Similar to our schemes, Flow control in FM [20] is also credit-based. In [2], a reliable multicast scheme is implemented by exploiting link-level flow control in Myrinet. GM [16] is a messaging layer over Myrinet developed by Myricom. In GM, a sender can only send a message when it owns a *send token*. This is essentially a credit-based flow control scheme. Unlike the above, our work concentrates on flow control schemes in MPI over InfiniBand.

MVICH [12] is an MPI implementation over VIA [6]. It uses a user-level static flow control scheme. Our original MPI implementation over InfiniBand [14] was based on it and used a similar flow control scheme. In this paper, we carry out a detailed study and comparison of different flow control schemes. For static schemes, we implemented two more methods to avoid deadlock: optimistic approach and RDMA approach. Our performance evaluation also shows that instead of using static schemes, we can use dynamic schemes to achieve both performance and scalability. We have also evaluated the hardware-based flow control scheme.

Recently we have developed a new MPI implementation over InfiniBand by exploiting RDMA write operations for small and control messages [13]. In this paper, our original implementation [14] is used as the basis for studying different flow control schemes. However, the results in this paper are directly applicable to the RDMA-based MPI implementation. The difference is that in the RDMA-based implementation, the user-level dynamic scheme is more complicated because cooperation between both the sender and the receiver is necessary in order to increase the number of posted buffers.

In order to improve the scalability of MPI implementations, an on-demand connection set-up scheme was proposed in [23]. In this scheme, connections are set up between two processes when they communicate with each other for the first time. If there is no communication between them, no connection will be set up and therefore no buffer space will be used. Our proposed dynamic flow control scheme can be combined with on-demand connection setup to further improve the scalability of MPI implementations.

8 Conclusions and Future Work

In this paper, we present a detailed study of the flow control issue in implementing MPI over the InfiniBand Architecture with Reliable Connection service. We categorize flow control schemes into three classes: hardware-based, user-level static and user-level dynamic. These schemes

differ in their run-time overhead and how their decide the number of pre-posted buffers for each connection. The hardware-based scheme has the least overhead under normal conditions. However, in user-level schemes, MPI implementation can have more control over the system communication behavior when the receiver is overloaded. In particular, the user-level dynamic scheme is able to adjust the number of pre-posted buffers according to the application communication pattern. Therefore, it can potentially achieve both good performance and high scalability in terms of buffer usage.

We have implemented all three schemes in our MPI implementation over InfiniBand and conducted performance evaluation on our 8-node InfiniBand cluster. We use both micro-benchmarks and the NAS Parallel Benchmarks for the evaluation. We have shown that the overheads of user-level schemes are very small. Our results have also shown that the user-level dynamic scheme can achieve both performance and buffer efficiency by adapting to the communication pattern. Another finding in our performance evaluation is that for most NAS applications, only a small number of pre-posted buffers are required to achieve good performance.

In future, we plan to extend our study to large-scale clusters and carry out more in-depth analysis of application scalability. We also plan to include more applications in our study. Although in this paper we concentrate on flow control issues in MPI, we believe that many of our results are applicable to the design of other middleware layers over InfiniBand, such as Distributed Shared Memory (DSM) systems and parallel file systems. We will investigate similar problems in these systems. Another direction we intend to pursue is to study flow control issues in using other InfiniBand transport services such as Reliable Datagram.

Acknowledgments

We would like to thank Jiesheng Wu and Sushmitha Kini for the helpful discussions.

References

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

[2] Bhoedjang, Ruhl, and Bal. Effi cient multicast on myrinet using link-level fbw control. In *ICPP: 27th International Conference on Parallel Processing*, 1998.

[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, R. Riesen, B. Lawry, and A. B. Maccabe. Portals 3.0: Protocol Building Blocks for Low Overhead Communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters (CAC)*, April 2002.

[5] D. Buntinas, D. K. Panda, and R. Brightwell. Application-bypass broadcast in mpich over gm. In *International Symposium on Cluster Computing and the Grid (CCGRID '03)*, May 2003.

[6] Compaq, Intel, and Microsoft. VI Architecture Specifi cation V1.0, December 1997.

[7] R. Dimitrov and A. Skjellum. An Effi cient 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] Infi niBand Trade Association. Infi niBand Architecture Specifi cation, Release 1.0, October 24 2000.

[12] Lawrence Livermore National Laboratory. MVICH: MPI for Virtual Interface Architecture, August 2001.

[13] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-Based MPI Implementation over Infi niBand. In *17th Annual ACM International Conference on Supercomputing (ICS '03)*, June 2003.

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

[15] Mellanox Technologies. Mellanox Infi niBand Infi niHost Adapters, July 2002.

[16] Myricom. GM Messaging Software. http://www.myri.com/scs/index.html.

[17] NASA. NAS Parallel Benchmarks.

[18] NCSA. MPICH over VMI2 Interface. http://vmi.ncsa.uiuc.edu/.

[19] Network-Based Computing Laboratory. MVAPICH: MPI for Infi niBand on VAPI Layer. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/index.html, January 2003.

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

[21] W. Rehm, R. Grabner, F. Mietke, T. Mehlan, and C. Siebert. Development of an MPICH2-CH3 Device for Infi niBand. http://www.tu-chemnitz.de/informatik/RA/cocgrid-/Infi niband/pmwiki.php/Infi niBandProject/ProjectPage.

[22] 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.

[23] 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.