Shared Receive Queue based Scalable MPI Design for InfiniBand Clusters

SAYANTAN SUR, LEI CHAI, HYUN-WOOK JIN, AND D. K. PANDA

Technical Report
OSU-CISRC-10/05-TR66

Shared Receive Queue based Scalable MPI Design for InfiniBand Clusters [*]

Sayantan Sur Lei Chai Hyun-Wook Jin
Dhabaleswar K. Panda

Network-Based Computing Laboratory
Department of Computer Science and Engineering
The Ohio State University
{surs, chail, jinhy, panda}@cse.ohio-state.edu

Abstract

InfiniBand is an emerging cluster interconnect that is being used in many modern clusters. InfiniBand clusters with several thousands of nodes have already appeared in the Top 500 list. The next-generation InfiniBand clusters are expected to be even larger with tens-of-thousands of nodes. Most of the scientific parallel applications running on these clusters are written using the Message Passing Interface (MPI). Thus, a high-performance scalable MPI design is crucial for these applications to unlock the potential of these large clusters. MVAPICH is a popular implementation of MPI over InfiniBand. It is based on a connection oriented InfiniBand interface. The requirement of this interface to make communication buffers available for each connection imposes a memory scalability problem. In order to mitigate this issue, the latest InfiniBand standard includes a new feature called Shared Receive Queue (SRQ) which allows sharing of communication buffers across multiple connections. In this paper, we propose a novel MPI design which efficiently utilizes SRQs and provides very good performance. Our analytical model reveals that our proposed designs will take only 1/10th the memory requirement as compared to the original design on a cluster sized at 16,000 nodes. Performance evaluation of our design on our 8-node PCI-Express shows that our new design was able to provide the same performance as the existing design utilizing only a fraction of the memory required by the existing design. In comparison to tuned existing designs our design showed a 20% and 5% improvement in execution time of NAS Benchmarks (Class A) LU and SP, respectively. The High Performance Linpack was able to execute a much larger problem size using our new design, whereas the existing design ran out of memory.

[*]This research is supported in part by Department of Energy's grant #DE-FC02-01ER25506, National Science Foundation's grants #CNS-0403342 and #CNS-0509452; grants from Intel, Mellanox, Cisco and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Apple, Microway, PathScale, Silverstorm and Sun Microsystems.

1 Introduction

Cluster computing has become quite popular during the past decade. The interconnect used in these clusters is very crucial for attaining the highest possible performance [1]. InfiniBand [7] is an emerging high-performance interconnect, offering low latency (1.5-3.0 microseconds) and high bandwidth (several GigaBytes/second). In addition to high-performance, InfiniBand also provides many advanced features like Remote Direct Memory Access (RDMA), atomic operations, multicast and QoS. As InfiniBand gains popularity, large scale clusters are being built using it [13]. Clusters of several tens-of-thousands of nodes have now appeared as the most powerful machines in the Top 500 list [18]. Accordingly, it is expected that the scale of the InfiniBand clusters to be deployed in the near future will be even larger. MPI [11] is the de-facto standard in writing parallel scientific applications. Hence, a scalable and high performance MPI design is very critical for end HPC applications which will run on these modern and next generation very large scale clusters.

MVAPICH [14] is a popular implementation of MPI over InfiniBand which is used by more than 270 organizations world-wide. It has enabled several InfiniBand clusters to achieve Top 500 ranks. MVAPICH is also distributed in an integrated manner with emerging OpenIB/Gen2 [15] software stack for Linux and Windows clusters. It implements the Abstract Device Interface (ADI) layer of MPICH [6]. MVAPICH was derived from MVICH [8].

MVAPICH uses a reliable connection oriented model provided by InfiniBand. This model provides superior performance on current generation InfiniBand stacks than the unreliable connectionless model as well as providing reliable transport. However, one of the restrictions is that messages can be received only in buffers which are already available to the Host Channel Adapter (HCA) or Network Interface Card (NIC). In order to achieve this, MVAPICH allocates and dedicates buffers for each connection (the number of connections increases as the number of processes). Although the amount of buffers allocated per connection can be tuned and MVAPICH has scaled quite well for contemporary clusters (up to 1000 nodes and beyond), the challenges imposed by the scale of next generation very large clusters (up to 10,000 nodes and beyond) is quite hard to meet with the current buffer management model.

The latest InfiniBand standard (Release 1.2) [7] has provided a new feature called *Shared Receive Queues* (SRQ) which aims at solving this scalability issue at the HCA level. This new feature removes the requirement that message buffers be available in a dedicated fashion for each connection. Using this feature, a process which intends to receive from multiple processes can in fact provide receive buffers in a single queue. Before utilizing this feature, however, we need to solve several design challenges.

In this paper, we carry out detailed analysis of the design alternatives and propose a high-performance MPI design using SRQ. We propose a novel flow control mechanism using a "watermark" based approach. In addition, we design a mechanism which can help users fine tune our designs on their specific platforms. Further, we come up with an analytical model which can predit memory usage by the MPI library on clusters of tens-of-thousands of nodes. Verification of our analytical model reveals that our model is accurate within 1%. Based on this model, our proposed designs will take only 1/10th the memory requirement as compared to the default MVAPICH distribution on a cluster sized at 16,000 nodes. Performance evaluation of our design on our 8-node PCI-Express shows that our new design was able to provide the same performance as the existing design utilizing only a fraction of the memory required by the existing design. In comparison to tuned existing designs our design showed a 20% and 5% improvement in execution time of NAS Benchmarks (Class A) LU and SP, respectively. The High Performance Linpack [4] was able to execute a much larger problem size using our new design, whereas the existing design ran out of memory.

The rest of the paper is organized as follows: in Section 2 we provide a background to our research. In Section 3 we present the motivation for designing MPI with SRQ. In Section 4 we propose our designs. In Section 5 we experimentally evaluate our designs. In Section 6 we discuss related work in this area. Finally in Section 7 we conclude the paper.

2 Background

In this section we provide a detailed background behind the work done in this paper. Broadly, there are two important topics which pertain to this work. First, we present an overview of the InfiniBand network and its transport models. Secondly, we describe the existing design of MPI over InfiniBand, particularly, MVAPICH [14].

2.1 InfiniBand Overview

The InfiniBand Architecture [7] (IBA) defines a switched network fabric for interconnecting processing and I/O nodes. In an InfiniBand network, hosts are connected to the fabric by Host Channel Adapters (HCAs). A queue based model is used in InfiniBand. A Queue Pair (QP) consists of a send and a receive queue. Communication operations are described in the Work Queue Requests (WQR), or descriptors, and submitted to the work queue. It is a requirement that all communication buffers be posted into receive work queues before any message can be placed into them. In addition, all communication buffers need to be registered (locked in physical memory) before any operations can be issued from there. This is to ensure that memory is present when HCA accesses the memory. Finally, the completion of WQRs is reported through Completion Queues (CQ).

IBA provides several types of transport services: Reliable Connection (RC), Unreliable Connection (UC), Reliable Datagram (RD) and Unreliable Datagram (RD). RC and UC are connection-oriented and require one QP to be connected to exactly one other QP. On the other hand, RD and UD are connectionless and one QP can be used to communicate with many remote QPs. To the best of our knowledge, Reliable Datagram (RD) transport has not been implemented by any InfiniBand vendor yet.

On top of these transport services, IBA provides software services. However, all software services are not defined for all transport types. Figure 1 depicts which software service is defined for which transport, as of IBA specification release 1.2. As shown in the figure, the send/receive operations are defined for all classes of transport. For connection-oriented transport, a new type of software service called Shared Receive Queue (SRQ) has been introduced. This allows the association of many QPs to one receive queue even for connection oriented transport. Thus, any remote process which is connected by a QP can send a message which is received in buffers specified in the SRQ.

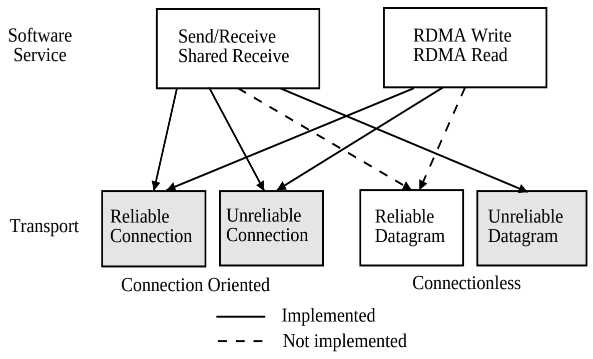


Figure 1. IBA Transport and Software Services

Apart from the basic send/receive operations, IBA also defines Remote Direct Memory (RDMA) operations. Using this service, applications can directly access memory locations of remote processes. In order to utilize RDMA, the requesting process is required to know the virtual address and a memory access key of the remote process. RDMA is supported on all reliable transports. The only exception being that RDMA Read is not supported on UC.

In addition to these features, IBA provides a host of other exciting features like hardware multicast, QoS, Atomic operations. These features are not described here because they are not related to the research direction discussed in the paper. Additional details on these features can be obtained from IBA specification [7].

2.2 MVAPICH Design Overview

MVAPICH [14] is a popular implementation of MPI over InfiniBand. It uses both Send/Receive and RDMA operations to achieve high-performance message passing. It has two modes of data transfer namely Eager and Rendezvous. Eager mode is used to transmit small messages over either the RDMA or send/receive. It is based on copy-in (sender side) and copy-out (receiver side), thus utilizing the pre-registered (locks in physical memory) MPI internal buffers as "communication buffers". The Rendezvous mode of communication is utilized for transferring large messages. The cost of copying large messages into MPI internal buffers is prohibitive. Instead, the Rendezvous mode directly registers the application buffer and directly sends the memory contents to the receiving process's memory location. Thus, the Rendezvous mode can achieve zero-copy data transfer.

Eager mode communication is divided into RDMA and send/receive channels. The RDMA channel [10] is the highest performing one out of them. This channel dedicates a certain number of buffers per connection. These buffers are used in a cyclic manner. These buffers are allocated during `MPI_Init` and the virtual addresses and memory keys are exchanged. Each process maintains a window of RDMA buffers with every other remote process for flow control. These buffers will be called "RDMA Buffers" in the rest of the paper.

The Send/Receive channel is used as a backup, in case the RDMA channel is unavailable for some reason (e.g., filled up with unexpected messages). This channel requires pre-posting a certain number of buffers for each QP connection during `MPI_Init`. When a certain threshold of messages is exceeded on any connection, a larger set of buffers are posted for that connection. The buffers used for Send/Receive are pre-allocated and registered during `MPI_Init`. These buffers will be called the "Send/Receive buffers" for the rest of the paper. To facilitate the easy management and to avoid runtime registration of these buffers, they are organized in a large pool (which is shared between all the connections) of typically thousands of small buffers. These buffers will be referred to as "Buffer Pool" in the rest of the paper.

3 SRQ Based MPI design: Is it Beneficial?

3.1 Limitations in Current Design

In Section 2.2 we described the design of MVAPICH. MVAPICH is based on the connection-oriented reliable transport of InfiniBand utilizing both RDMA and Send/Receive channels. In order to communicate using these channels, it has to allocate and dedicate buffers to each remote process. This means that the memory consumption grows linearly with the number of processes. Although the number of buffers per process can be tuned (at runtime), and MVAPICH has scaled well for contemporary InfiniBand clusters, the next-generation InfiniBand clusters are in the order of tens-of-thousands of nodes. In order for MVAPICH to scale well for these clusters, the linear growth of memory requirement with number of processes has to be removed.

Adaptive buffer management is a mechanism by which the MPI can control the amount of buffers available for each connection during runtime based on message patterns. However, there are several problems with this mechanism when implemented on top of the Send/Receive and RDMA channels:

- **Send/Receive Channel:** This channel allows us to choose how many buffers are posted on it dynamically. However, buffers once posted on a receive queue cannot be recalled. Hence, posted buffers on idle connections lead to wasted memory. This problem exacerbates memory consumption issues in large scale applications that run for a very long time. In addition, if MVAPICH is very aggressively tuned to run with low number of buffers per Send/Receive channel, this will lead to performance degradation. This is because the Send/Receive channel is based on window-based flow control mechanism [9]. Reducing the window in order to reduce memory consumption hampers the message passing performance.

- **RDMA Channel:** This channel allows very low-latency message passing. However, the allocation of buffers for every connection is very rigid. The cyclic window of buffers (Section 2.2) needs to be contiguous memory. If not, then another round of address and memory key exchange (extra overhead) is required. Recalling of RDMA buffers is possible from any connection, but there is an additional overhead of informing remote nodes about the reduced memory they have with the receiving process. This process can lead to some race conditions which have to be eliminated using further expensive atomic operations, thus, leading to high overheads.

Thus, in order to improve the buffer usage scalability of MPI while preserving high-performance we need to explore a different communication channel.

3.2 Benefits of SRQ

Since we aim to remove the dependence of number of communication buffers with the number of MPI processes, we need to look at connectionless models. As described in Section 2.1, InfiniBand provides two kinds of connectionless transport. One is Reliable Datagram (RD) and the other is Unreliable Datagram (UD). Unfortunately, Reliable Datagram is not implemented in any InfiniBand stack (to the best of our knowledge), so that rules out this option. UD can provide the scalable features, but the MPI design would now have to provide reliability. This will add to the overall cost of message transfers, and may result in loss of high-performance. In addition, UD does not support RDMA features, which are needed for zero-copy message transfer, thus further degrading performance.

Shared Receive Queues (SRQ) provides a model to efficiently share receive buffers across connections whilst maintaining the good performance and reliability of a connection oriented transport. Thus, the SRQ is a good candidate for achieving scalable buffer management.

Figure 2 shows the difference between the buffer organization schemes for MVAPICH and the new proposed design based on SRQ.

3.3 Reduced Memory Consumption with SRQ

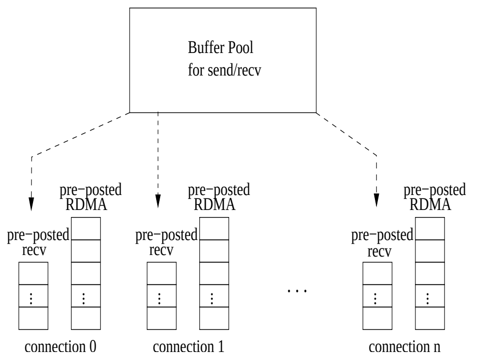
In order to fully understand the impact of the memory usage model of our proposed SRQ based design, we construct an analytical model of the memory consumption by MPI internal buffers.

There are several components of the memory consumed during startup. The major components are memory consumed by the Buffer Pool, RDMA channel, Send/Receive channel and the memory consumed by the InfiniBand RC connections themselves.

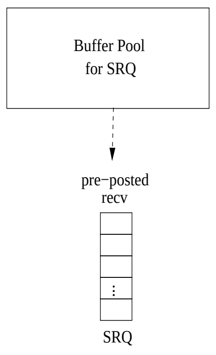
The size of the Buffer Pool is given by the product of the number of buffers in the pool and the size of each buffer.

$$M_{bp} = N_{pool} * S_{buf} \tag{1}$$

Where, M_{bp} is the amount of memory consumed by the Buffer Pool, N_{pool} is the number



(a) Existing MVAPICH Buffer Organization



(b) Proposed SRQ Based Buffer Organization

Figure 2. Comparison of Buffer Management Models

of buffers in the pool and S_{buf} is the size of each buffer.

The memory consumed by MVAPICH-SR (tuned version of MVAPICH using only Send/Receive channel) is composed of three parts, the memory consumed by the Buffer Pool and the memory consumed by each connection and the Send/Receive buffers.

$$M_{sr} = M_{bp} + (M_{rc} + N_{sr} * S_{buf}) * N_{conn} \quad (2)$$

Where, M_{sr} is the amount of memory consumed by MVAPICH-SR, M_{rc} is the memory needed for each InfiniBand connection by HCA driver, N_{sr} is the number of Send/Receive buffers for each connection and N_{conn} is the total number of connections.

MVAPICH-RDMA (default version of MVAPICH using both RDMA and Send/Receive channels) consumes all the memory as MVAPICH-SR and in addition, allocates RDMA buffers for each connection. The RDMA channel also needs to keep dedicated send buffers per connection [10]. Hence, the amount of dedicated buffers per connection doubles.

$$M_{rdma} = M_{sr} + 2 * N_{rdma} * S_{buf} * N_{conn} \quad (3)$$

Where, M_{rdma} is the amount of memory consumed by MVAPICH-RDMA and N_{rdma} is the number of RDMA buffers per connection.

Finally, the MVAPICH-SRQ (our proposed SRQ based design) only needs to allocate the Buffer Pool and a fixed number of buffers for posting to the SRQ.

$$M_{srq} = M_{bp} + M_{rc} * N_{conn} + N_{srq} * S_{buf} \quad (4)$$

Where, M_{srq} is the memory consumed by MVAPICH-SRQ and N_{srq} is the number of SRQ buffers.

Analyzing Equations 2, 3 and 4, we observe that the memory requirement by MVAPICH-SRQ is much lesser if the number of connections is very large.

4 Proposed SRQ Based MPI Design

In this section we present the design challenges associated with SRQ based MPI design. The SRQ mechanism achieves good buffer scalability by exposing the same set of receive WQEs to all remote processes on a first come first serve (FCFS) basis. However, in this mechanism, the sending process lacks a critical piece of information: number of available receive buffers at the receiver. In the absence of this information, the sender can overrun the available buffers in the SRQ. To achieve optimal message passing performance, it is critical that this situation is avoided. In the following sections, we propose our novel design which enables the benefits provided by SRQ, while avoiding senders from over-running receive buffers.

4.1 Proposed SRQ Refilling Mechanism

A high-performance MPI design often requires the MPI progress engine to be polling to achieve the lowest possible point-to-point latency. MVAPICH is thus based on a polling progress engine. Ideally, we would like to maintain the polling nature of the MPI for the SRQ based design. However, in this polling based design, MPI can only discover incoming messages from the network when explicit MPI calls are made. This increases the time intervals in which MPI can check the state of the SRQ. Moreover, if the MPI application is busy performing computation or involved in I/O, there can be prolonged periods in which the state of SRQ is not observed by the MPI. In the meantime, the SRQ might have become full. In order to efficiently utilize SRQ feature, we must avoid this situation. Broadly, three design alternatives can be utilized: Explicit acknowledgement from receiver, Interrupt based progress and Selective interrupt based progress.

4.1.1 Explicit Acknowledgement from Receiver

In this approach, the sending processes can be instructed to refrain from sending messages to a particular receiver unless they receive an explicit `OK_TO_SEND` message after every k messages. Arrival of the `OK_TO_SEND` message means that the receiver has reserved k buffers in a dedicated manner for this sender and allows the sender to send k more messages. Where k is a threshold of messages that can be tuned or selected at runtime. This scheme can avoid the scenario in which the sender completely fills up the receiver queue with messages. This scheme is illustrated in Figure 3. However, this scheme suffers from a couple of critical deficiencies:

1. **Waste of Receive Buffer:** Since in this design alternative we reserve k buffers for a specific sender if the sender does not have more messages to send the memory resource for the reserved buffers can be wasted. To prevent this problem, if we reduce the value of k, we cannot achieve high bandwidth because the sender should wait the `OK_TO_SEND` message for every few messages.

2. **Early throttling of senders:** Even though not all senders may be transmitting at the same time, a particular sender may send k messages and then be throttled until the receiver sends the `OK_TO_SEND` message. If the receiver is busy because of a computation, the sender blocks until the receiver operates the progress engine and sends the `OK_TO_SEND` message.

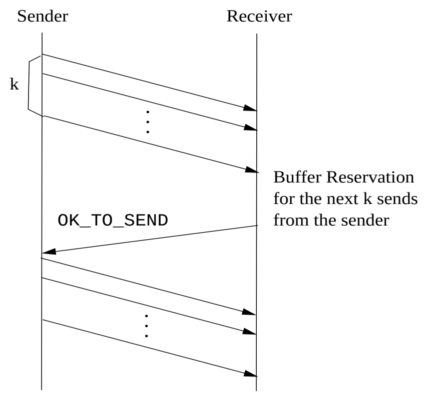


Figure 3. Explicit ACK mechanism

4.1.2 Interrupt Based Progress

As mentioned earlier in this section, if the MPI application is busy performing computation or I/O, it cannot observe the state of the SRQ. In this design approach, the progress engine of the

MPI is modified so as to explicitly request an interrupt before returning execution control to the MPI application. If there is an arrival of a new message, the interrupt handler thread becomes active and processes the message along with refilling the SRQ. The Figure 4 illustrates this design alternative.

This approach can effectively avoid the situation where the SRQ is left without any receive WQEs. However, this approach also has a limitation. There is now an interrupt on arrival of any new message when the application is busy computing. This can cause increased overhead and lead to non-optimal performance. In addition, we note that the arrival of the next message as such is not a critical event. There may be several WQEs still available in the SRQ. Hence, most of the interrupts caused by this mechanism will be unnecessary.

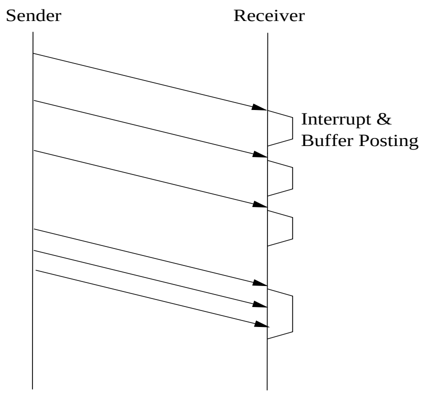


Figure 4. Interrupt Based Progress

4.1.3 Selective Interrupt Based Progress

In this design alternative, we try to minimize the number of interrupts to the bare minimum. InfiniBand provides an asynchronous event associated with a SRQ called `SRQ_LIMIT_REACHED`. This asynchronous event is fired when a low "watermark" threshold (preset by the application) is reached. This event allows the application to act accordingly. In our case, we can utilize this event to trigger a thread to post more WQEs in the SRQ. The Figure 5 demonstrates the sequence of operations. In step 1, the remote processes send messages to the receiver. In step 2, the arrival of a new message causes the SRQ WQE count to drop below the limit (as shown by the grayed out region of the SRQ). In step 3, the thread designated to handle this asynchronous event (called LIMIT thread from now on) becomes active. In step 4, the LIMIT thread posts more WQEs to the SRQ. It should be noted that as soon as a SRQ WQE is consumed it is directly moved to the completion queue (CQ) by the HCA driver.

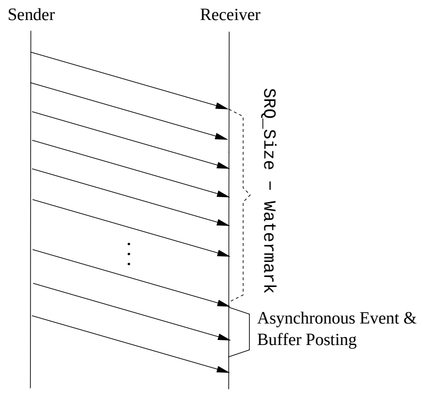


Figure 5. SRQ Limit Event Based Design

This design alternative meets our design criteria and causes minimum interference with the MPI application. Hence, we choose this design alternative for our SRQ based MPI design.

4.2 Proposed Design of SRQ Limit Threshold

As mentioned in Section 4.1.3, we utilize the `SRQ_LIMIT_REACHED` asynchronous event provided by InfiniBand. This event is fired when a preset limit is reached on the SRQ. In order to achieve an optimal design, we need to make sure that the event is: a) not fired too often and b) has enough time to post buffers so that SRQ is not left empty.

In order to calculate a reasonable low watermark limit, we need to find out the rate at which the HCA can fill up receive buffers. We can find out this information in a dynamic manner by querying the HCA. In addition to that, we need to find out the time taken by the LIMIT thread to become active. For finding out this value, we design an experiment, as illustrated in Figure 6. In this experiment, we measure the round-trip time using SRQ (marked as t_1). The subsequent message triggers the SRQ LIMIT thread which replies back with a special message. We mark this time as t_2. The LIMIT thread wakeup latency is given by: $(t_2 - t_1)$. On our platform, this is around 12μs.

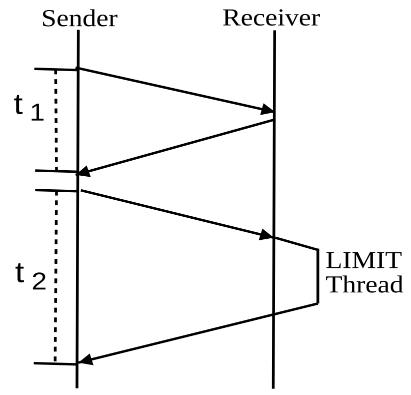


Figure 6. LIMIT Thread Wakeup Latency

Thus, we can calculate the minimum low watermark limit as:

$$Watermark = \frac{BW * 10^3}{MinPacketSize} * t_{wakeup} \quad (5)$$

Where, BW is the maximum bandwidth supported by the HCA is Gb/s, $MinPacketSize$ is the minimum packet size of MPI messages in bits and t_{wakeup} is the time taken by the LIMIT thread to wake up in microseconds. For our experimental platform (described in Section 5), the $Watermark$ value is 300. In addition to the MPI library, another utility will be distributed which can automatically calculate the value of the $Watermark$ value on the MPI library user's platform. The user can then simply plug in this value in the MPI application's environment, from where it will be picked up by the MPI library.

5 Experimental Evaluation

In this section we evaluate the memory usage and performance of our MPI with SRQ design over InfiniBand. We first introduce the experimental environment, and then compare our design with MVAPICH in terms of memory usage and application performance. We also show the importance of flow control in using SRQ.

The default configuration of MVAPICH is to use a set of pre-registered RDMA buffers for small and control messages as described in section 2. In our performance graphs we call this configuration "MVAPICH-RDMA". MVAPICH can also be configured to use "Send/Receive" buffers for small and control messages. We also compared with this configuration, and it is called "MVAPICH-SR" in the graphs. We have incorporated our design into MVAPICH, and it is called "MVAPICH-SRQ".

5.1 Experimental Environment

Our testbed cluster consists of 8 dual Intel Xeon 3.2GHz EM64T systems. Each node is equipped with 512MB of DDR memory and PCI-Express interface. These nodes have MT25128 Mellanox HCAs with firmware version 5.1.0. The nodes are connected by an 8-port Mellanox InfiniBand switch. The Linux kernel used here is version 2.6.13.1. Open-IB Gen2 [15] is installed on all nodes.

5.2 Startup Memory Utilization

In this section we analyze the startup memory utilization of our proposed designs as compared to MVAPICH-RDMA and MVAPICH-SR. In our experiment, the MPI program starts up and goes to sleep after `MPI_Init`. Then we use the UNIX utility *pmap* to record the total memory usage of any one process. The same process is repeated for MVAPICH-RDMA, MVAPICH-SR and MVAPICH-SRQ. The results are shown in Figure 7.

Observing Figure 7, we can see that MVAPICH-RDMA scheme consumes the most memory. Since the RDMA buffers are dedicated to each and every connection, the

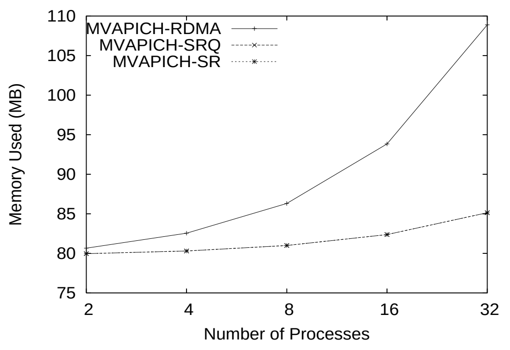


Figure 7. Memory Utilization Experiment

memory requirement grows linearly as number of processes. On the other hand, the MVAPICH-SR, as described earlier in this section is a highly-tuned version of MVAPICH using Send/Receive channel. This uses the same amount of memory as MVAPICH-SRQ. This is because the number of actual processes running is not enough for the per connection buffer posting to empty the *Buffer Pool*. If the number of processes is increased to a few hundred, then MVAPICH-SR will consume more memory than MVAPICH-SRQ. MVAPICH-SRQ just requires the same Buffer Pool and a *fixed* number of buffers which are posted on the SRQ. This number does not grow with the number of processes.

In Section 3.3, we have developed an analytical model for predicting the memory consumption by MVAPICH-RDMA, MVAPICH-SR and MVAPICH-SRQ on very large scale systems. In this section, we will first validate our analytical model and then use this to extrapolate memory consumption numbers on much larger scale systems.

On our experimental platform and MVAPICH configuration, the values of these parameters are: N_{rdma} = 32, N_{sr} = 10, N_{pool} = 5000, S_{buf} = 12KB, M_{rc} = 88KB. In addition, we have measured a constant overhead of 20MB which is contributed by various other libraries required by MVAPICH. It is to be noted that in the experiment, N_{sr} is simply taken from the Buffer Pool, so this factor does not show up. In Figure 8 we show the error margin of our analytical model with the measured data. We observe that our analytical model is indeed quite accurate.

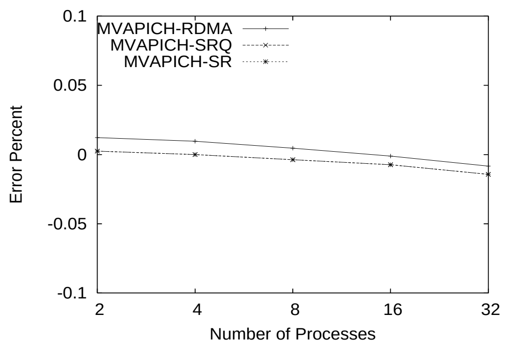


Figure 8. Error Margin of Analytical Model

Now we use this model to predict the memory consumption on much larger scale clusters. By increasing the number of connections and using the above mentioned parameter values, we extrapolate the memory consumption for each of the three schemes. The results are shown in Figure 9.

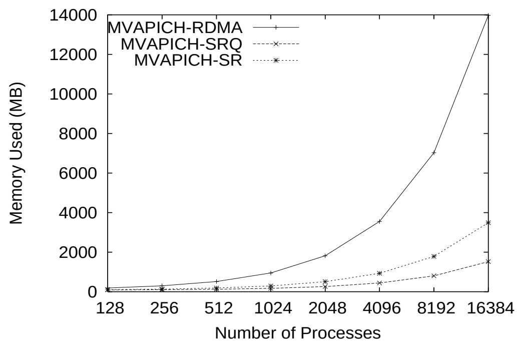


Figure 9. Estimation of memory consumption on very large clusters

5.3 Flow Control

In this section we present the importance of having flow control in using SRQ. We designed a micro-benchmark to illustrate it. The benchmark includes two nodes. The receiver first posts non-blocking receives (`MPI_Irecv`), and then starts computing. While the receiver is busy computing, the sender sends a "burst size" number of messages to the receiver. After the receiver finishes computing, it calls `MPI_Waitall` to finally get all the messages. We record the time the receiver spends in `MPI_Waitall` as an indication of how well the receiver can handle the incoming messages while it is computing.

Figure 10 shows the experimental results. We used the selective interrupt based approach for flow control as described in section 4.1.3. We can easily see from the graph that MVAPICH-SRQ without flow control can handle messages as well as MVAPICH-SRQ with flow control up to burst size around 250. After that, the line of MVAPICH-SRQ without flow control goes up steeply, which means the performance becomes very bad. As we discussed in section 4.1, without flow control the receiver can only update (refill) the SRQ when it calls the progress engine. In this benchmark, since the receiver is busy computing, it has no means to detect the SRQ is full, so the incoming messages get silently dropped. Only after computation, the receiver can resume to receive messages, but it has already lost computation/communication overlap and the network traffic becomes messy because of the sender retries. MVAPICH-SRQ with flow control, however, can handle a large "burst size" number of messages, and it doesn't add much overhead. In later sections MVAPICH-SRQ refers to MVAPICH-SRQ with selective interrupt based flow control.

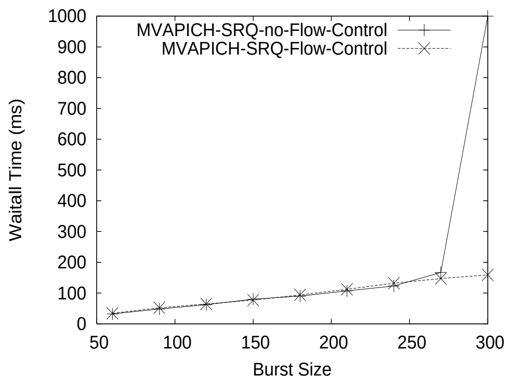


Figure 10. MPI_Waitall Time comparison

5.4 NAS Benchmarks

In this section we present the performance of MVAPICH-SRQ by using NAS Parallel Benchmarks [2], Class A. We conducted experiments on 16 processes. Figures 11 and 12 show the total execution time of MVAPICH-RDMA, MVAPICH-SR, and MVAPICH-SRQ.

From these two graphs we can see that for all benchmarks MVAPICH-SRQ performs almost exactly the same as MVAPICH-RDMA, which means using MVAPICH-SRQ we can dramatically reduce memory usage while not sacrificing performance at all. Looking at MVAPICH-SR, however, we can see that for LU, it performs 20% worse than MVAPICH-SRQ. This is because LU uses a lot of small messages, and in MVAPICH-SR, the sender will be blocked if it doesn't have enough credits from the receiver, as described in section 3.1. This is not a problem in MVAPICH-SRQ, because the sender can always send without any limitations. Similarly we can see a 5% performance difference between MVAPICH-SR and MVAPICH-SRQ for SP.

Comparing the performance of MVAPICH-SRQ and MVAPICH-SR, we find that although MVAPICH-SR can also reduce memory usage compared with MVAPICH-RDMA, it leads to performance degradation, so MVAPICH-SRQ is a better solution.

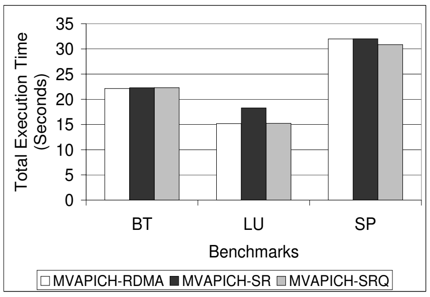


Figure 11. NAS Benchmarks Class A Total Execution Time (BT, LU, SP)

5.5 High Performance Linpack

In this section we carry out experiments using the standard High Performance Linpack (HPL)

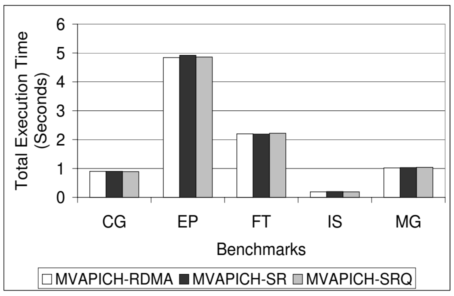


Figure 12. NAS Benchmarks Class A Total Execution Time (CG, EP, FT, IS, MG)

benchmark [4]. HPL stresses various components of a system including memory usage. For every system there is a limit for problem size based on the total amount of physical memory. The benchmark cannot run if the problem size goes beyond the limit. Figure 13 shows the performances of MVAPICH-RDMA, MVAPICH-SR, and MVAPICH-SRQ, in terms of Gflops.

From this graph we can see that MVAPICH-SR and MVAPICH-SRQ perform comparably with MVAPICH-RDMA for problem size from 10000 to 15000. For some problem sizes, such as 11000, 12000, and 13000, MVAPICH-SR and MVAPICH-SRQ perform even 10% better than MVAPICH-RDMA. This is because MVAPICH-RDMA needs to poll RDMA buffers of each connection when it makes communication progress. This polling wastes CPU cycles and pollutes cache content.

It is to be noted that for problem size 16000, the result for MVAPICH-RDMA is missing. This is because the memory usage of MVAPICH-RDMA itself is so large that the benchmark doesn't have enough memory to run. In other words, the problem size limit for MVAPICH-RDMA is around 15000. MVAPICH-SR and MVAPICH-SRQ, however, continue to give good performance as the problem size increases. Our system size is not large enough to show that MVAPICH-SRQ scales better than MVAPICH-SR. On a much larger cluster we will also be able to show that MVAPICH-SR has a smaller problem size limit than MVAPICH-SRQ.

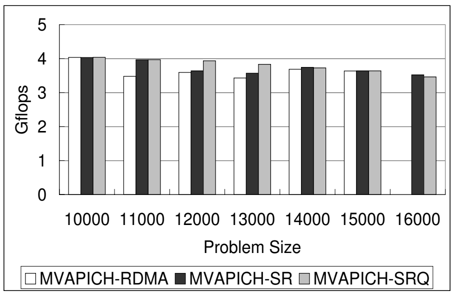


Figure 13. High Performance Linpack

6 Related Work

To reduce resource usage, P. Gilfeather and A. Maccabe proposed connection-less TCP [5]. They discussed issues in dynamically activating and deactivating TCP connections based on the need. But their paper mainly focused on connection management while this paper focuses on designing a Shared Receive Queue based buffer management.

Scalability limitations of VIA-based technologies in supporting MPI are discussed in [3]. In that paper the authors analyzed various issues that might prevent the system to scale. In this paper we propose to use Shared Receive Queue as a solution to improve memory usage scalability.

Many messaging libraries provide connection-less services to minimize memory resource allocation, such as GM [12], AM [17], and VMI [16]. But to the best of our knowledge they don't provide Shared Receive Queue.

7 Conclusion and Future Work

In this paper, we have proposed a novel Shared Receive Queue based Scalable MPI design. Our designs have been incorporated into MVAPICH which is a widely used MPI library over InfiniBand. Our design uses selective interrupts to achieve efficient flow control and utilizes the memory available to the fullest extent, thus dramatically improving the system scalability. In addition, we also proposed an analytical model to predict the memory requirement by the MPI library on very large clusters (to the tune of tens-of-thousands of nodes).

Verification of our analytical model reveals that our model is accurate within 1%. Based on this model, our proposed designs will take 1/10th the memory requirement as compared to the default MVAPICH distribution on a cluster sized at 16,000 nodes. Performance evaluation of our design on our 8-node PCI-Express shows that our new design was able to provide the same performance as the existing design utilizing only a fraction of the memory required by the existing design. In comparison to tuned existing designs our design showed a 20% and 5% improvement in execution time of NAS Benchmarks (Class A) LU and SP, respectively. The High Performance Linpack [4] was able to execute a much larger problem size using our new design, whereas the existing design ran out of memory.

We will continue working in this research area. We want to evaluate our design on a larger InfiniBand cluster. In addition to that we want to explore reliable data transfer mechanisms over Unreliable Datagram to achieve scalability for ultra-scale clusters (hundreds-of-thousands of nodes).

References

[1] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing (Second Edition)*. Benjamin/Cummings Publishing, 1994.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. volume 5, pages 63–73, Fall 1991.

[3] R. Brightwell and A. B. Maccabe. Scalability Limitations of VIA-Based Technologies in Supporting MPI. In *Fourth MPI Developer's and User's Conference*, 2000.

[4] J. Dongarra. Performance of Various Computers Using Standard Linear Equations Software. Technical Report CS-89-85, University of Tennessee, 1989.

[5] P. Gilfeather and A. B. Maccabe. Connection-less TCP. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 9*, 2005.

[6] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI, Message Passing Interface Standard. Technical report, Argonne National Laboratory and Mississippi State University.

[7] InfiniBand Trade Association. InfiniBand Trade Association. http://www.infinibandta.com.

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

[9] J. Liu and D. K. Panda. Implementing Efficient and Scalable Flow Control Schemes in MPI over InfiniBand. In *Workshop on Communication Architecture for Clusters (CAC) held in conjunction with IPDPS*, 2004.

[10] J. Liu, J. Wu, , and D. K. Panda. High performance RDMA-based MPI implementation over InfiniBand. *Int'l Journal of Parallel Programming*, In Press, 2004.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, Mar 1994.

[12] Myricom Inc. http://www.myrinet.com/.

[13] NAS Project: Columbia. Columbia Supercomputer. http://www.nas.nasa.gov/About/Projects/Columbia/columbia.html.

[14] Network-Based Computing Laboratory. MPI over InfiniBand Project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.

[15] Open InfiniBand Alliance. Open Source InfiniBand Software Stack. http://www.openib.org/.

[16] A. Pant, S. Krishnamurthy, R. Pennington, M. Showerman, and Q. Liu. VMI: An Efficient Messaging Library for Heterogeneous Cluster Communication. http://www.ncsa.uiuc.edu/Divisions/CC/ntcluster/VMI/hpdc.pdf.

[17] The Berkeley NOW Project. Active Messages. http://now.cs.berkeley.edu/AM/active_messages.html.

[18] The Top 500 Project. The Top 500. http://www.top500.org/.