Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation

JIUXING LIU, ABHINAV VISHNU AND D. K. PANDA

Technical Report
OSU-CISRC-5/04-TR26

Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation*

Jiuxing Liu Abhinav Vishnu Dhabaleswar K. Panda
Computer Science and Engineering
The Ohio State University
Columbus, OH 43210
{liuj, vishnu, panda}@cis.ohio-state.edu

Abstract

InfiniBand is becoming increasingly popular in the area of cluster computing due to its open standard and high performance. However, even with InfiniBand, network bandwidth can become the performance bottleneck for some of today's most demanding applications.

In this paper, we study the problem of overcoming the bandwidth bottleneck by using multirail networks. We present different ways (multiple HCAs, multiple ports and virtual multirail configuration) of setting up multirail networks with InfiniBand and propose a unified MPI design that can support these approaches. We discuss various important design issues (out of order message handling, handling multiple HCAs) and provide an in-depth discussion of different policies for using multirail networks. We also propose an adaptive striping scheme that can dynamically change the striping parameters based on current system conditions.

We implement our design and evaluate it with microbenchmarks and applications. Our performance results show that multirail networks can significantly improve MPI communication performance. With a two rail InfiniBand cluster, we can achieve almost twice the bandwidth and half the latency for large messages compared to the original MPI implementation. The multirail MPI implementation can significantly reduce the communication time as well as the total execution time depending on the communication pattern at the application level. We also show that the adaptive striping scheme can achieve excellent performance without apriori knowledge of individual rail bandwidth.

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

1 Introduction

In the past couple of years, the computational power of commodity PCs has been doubling about every eighteen months. At the same time, network interconnects that provide low latency and high bandwidth are also emerging. This trend makes it very promising to build high performance computing environments by *Cluster Computing*. It combines the computational power of commodity PCs and the communication performance of high speed network interconnects. In this area, Message Passing Interface (MPI) [8] has become the *de facto* standard of writing parallel applications.

Recently, InfiniBand Architecture [9] has been proposed as the next generation interconnect for I/O and inter-process communication. Due to its open standard and high performance, InfiniBand is becoming increasingly popular for cluster computing. High performance MPI implementations over InfiniBand have also become available [17, 16]. One of the notable features of InfiniBand is its high bandwidth. Currently, InfiniBand 4x links support a peak bandwidth of 1GB/s in each direction. However, even with InfiniBand, network bandwidth can become the performance bottleneck for some of today's most demanding applications. This is especially the case for clusters built with SMP (2-16 way symmetric multiprocessor systems) machines, in which multiple processes may run on a single node and must share the node bandwidth.

One important way to overcome the bandwidth bottleneck is to use *multirail networks* [4]. The basic idea is to have multiple independent networks(rails) to connect nodes in a cluster. With multirail networks, communication traffic can be distributed to different rails. There are two ways of distributing communication traffic. In *multiplexing*[1], messages are sent through different rails in a round robin fashion. In *striping*, messages are divided into several chunks

[1] Also called *reverse multiplexing* in the networking community.

and sent out simultaneously using multiple rails. By using these techniques, the bandwidth bottleneck can be significantly alleviated.

In this paper, we present a detailed study in designing high performance multirail InfiniBand clusters. We discuss various ways of setting up multirail networks with InfiniBand and propose a unified MPI design that can support these approaches. Our design achieves low overhead by taking advantage of RDMA operations in InfiniBand and integrating the multirail design with MPI communication protocols. Our design also features a very flexible architecture that supports different policies of using multiple rails. We provide an in-depth discussion of different policies (*even* and *weighted striping*) and also propose an adaptive striping policy that can dynamically change the striping parameters based on the current available bandwidth for different rails.

We implement our design and evaluate it using microbenchmarks and applications on a 8-node InfiniBand testbed. Our performance results show that multirail networks can significantly improve MPI communication performance. With a two rail InfiniBand network, we can achieve almost twice the bandwidth and half the latency for large messages compared with the original MPI implementation. Depending on the communication pattern, multirail MPI implementation can significantly reduce communication time as well as execution time for certain applications. We also show that for rails with different bandwidth, the adaptive striping scheme can achieve excellent performance without *apriori* knowledge of the bandwidth of each rail. It can even outperform static schemes with *apriori* knowledge of rail bandwidth in certain cases.

The rest of the paper is organized as follows: In Section 2, we provide background information for Infiniband and MPI Protocols. We discuss different ways of setting up InfiniBand Multirail networks in Section 3. We describe the multirail MPI design in Section 4. In Section 5, we discuss the design issues like handling out of order messages and multiples HCAs. In Section 6, we present performance results of our multirail MPI implementation. We move to related work in Section 7. In Section 8, we conclude and discuss our future directions.

2 Background

In this section, we provide background information for our work. First, we provide a brief introduction of InfiniBand. Then, we discuss some of the internal communication protocols used by MPI and their implementation over InfiniBand.

2.1 Overview of InfiniBand

The InfiniBand Architecture (IBA) [9] defines a switched network fabric for interconnecting processing nodes and I/O nodes. It provides a 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). There are two kinds of channel adapters: *Host Channel Adapter* (HCA) and *Target Channel Adapter* (TCA). HCAs sit on processing nodes.

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

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

2.2 Overview of MPI Protocols

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. These protocols are handled by a component in the MPI implementation called *progress engine*. In Eager protocol, the message is pushed to the receiver side regardless of its state. In Rendezvous protocol, a handshake takes place 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.

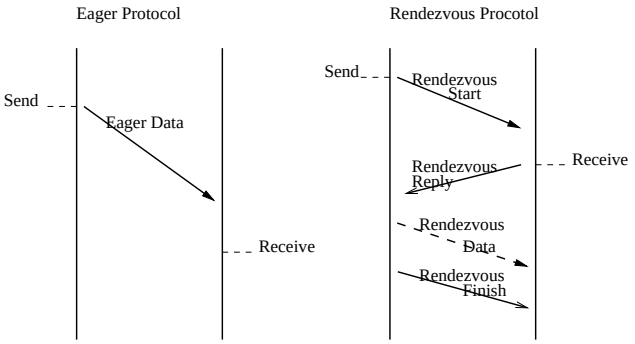

Figure 1. MPI Eager and Rendezvous Protocols

For the transfer of 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 [17, 10, 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 InfiniBand send/receive operations. However, send/receive operations have their disadvantages such as lower performance and higher overhead. Therefore, our recent work in [11] proposed a scheme that uses RDMA operations also for small data and control messages. This scheme improves both latency and bandwidth of small message transfers in MPI.

3 InfiniBand Multirail Network Configurations

InfiniBand multirail networks can be set up in different ways. In this section, we discuss three possible multirail network configurations and their respective benefits. In the first approach, multiple HCAs are used at each node. The second approach exploits multiple ports on a single HCA. Finally, we describe how to set up *virtual multirail networks* with a single port of each HCA by using the LID mask control (LMC) mechanism in InfiniBand.

3.1 Multiple HCAs

Although InfiniBand Architecture specifies 12x links, currently available InfiniBand HCAs support only up to 4x speed. A straightforward way to alleviate the bandwidth bottleneck is to use multiple HCAs at each node and connect them to the InfiniBand switch fabric. With the support of communication software, users can take advantage of the aggregated bandwidth of all HCAs in each node without modifying any user application. Another advantage of using multiple HCAs per node is that possible bandwidth bottlenecks in local I/O buses can also be avoided. For example, the PCI-X 133 MHz/64 bit bus (used by most 4x HCAs in currently available) can only support around 1 GB/s aggregated bandwidth. Although a 4x HCA has a peak aggregated bandwidth of 2 GB/s for both link directions, its performance is limited by the PCI-X bus. This problems can be alleviated by connecting multiple HCAs to different I/O buses in a system.

A multirail InfiniBand setup using multiple HCAs per node can connect each HCA on a node to a separate switch. If enough ports on the switch are available, all HCAs can be connected to the single physical network. By using appropriate switch configurations and routing algorithms, using a single network can be equivalent to a multirail setup.

3.2 Multiple Ports

Currently, many publically available InfiniBand HCAs have multiple ports. For example, InfiniHost HCAs [12] from Mellanox have two ports on each Channel Adapter. Therefore, multirail InfiniBand networks can also be constructed by taking advantage of multiple ports in a single HCA. This approach looks very attractive, because compared with using multiple HCAs, it only requires one HCA per node. Hence, the total cost of multirail networks can be significantly reduced.

However, as already stated, the local I/O bus can be the performance bottleneck in such a configuration because all ports of a HCA have to share the I/O bus. Hence, this approach will not achieve any performance benefit by using 4x HCAs with PCI-X buses. However, benefits may be achieved by using future HCAs that support PCI-X Double Data Rate (DDR) or Quad Data Rate (QDR) interfaces. Recently, PCI Express [19] has been introduced as the next generation local I/O interconnect. PCI Express uses a serial, point-to-point interface. It can deliver scalable bandwidth by using multiple lanes in each point-to-point link. For example, an 8x PCI Express link can achieve 2 GB/s bandwidth in each direction (4 GB/s total). Multiple port InfiniBand HCAs that support PCI Express are already available publically [13]. Therefore, this approach can be very useful in constructing multirail networks using systems that

have PCI Express interfaces. Although, with large (8-way, 16-way) SMP configurations and proposed 16X/32X PCI-Express systems, multiple HCAs will provide a very scalable solution.

3.3 Single Port with LMC

In this subsection, we discuss another approach of setting up multirail InfiniBand networks, which does not require multiple ports or HCAs for each node. The basic idea of this approach is to set up different paths between two ports on two nodes. By using appropriate routing algorithms, it is possible to make the paths independent of each other. Although a single network is used in this approach, we have multiple logical networks (or logical rails). If the logical networks are independent of each other, conceptually they are very similar to multirail networks. We call them *virtual multirail networks*.

In InfiniBand, each port has a *local identifier* (LID). Usually, a path is determined by the destination LID. Therefore, multiple LIDs need to be used in order to have different paths. To address this issue, InfiniBand provides a mechanism called *LID Mask Control* (LMC). LMC provides a mechanism to associate multiple logical LIDs with a single physical port. Hence, multiple paths can be constructed by using LMC.

It should be noted that in virtual multirail networks, a port is shared by all the logical rails. Hence, if the port link bandwidth or the local I/O bus is the performance bottleneck, this approach cannot bring any performance benefit. It can only be used for fault tolerance in this case. However, if the performance bottleneck is inside the network, virtual multirail networks can improve communication performance by utilizing multiple independent paths.

4 Multirail MPI Design

In this section, we present the high level design issues involved in supporting multirail networks in MPI over InfiniBand. We first present the basic architecture of our design. Next, we discuss how we can have a unified design to support multirail networks using multiple HCAs, multiple ports, multiple connections for a single port, or any combination of the above. Next, we describe how do we achieve low overhead by integrating our design with MPI and taking advantage of InfiniBand RDMA operations. One important component of our architecture is *Scheduling Policies*. In the last part of this section, we discuss several policies supported by our architecture and present an *adaptive striping scheme* that can dynamically adjust striping parameters based on current system conditions.

4.1 Basic Architecture

The basic architecture of our design to support multirail networks is shown in Figure 2. We focus on the architecture of the sender side. In the figure, we can see that besides MPI Protocol Layer and InfiniBand Layer, our design consists of three major components: *Communication Scheduler*, *Scheduling Policies*, and *Completion Filter*.

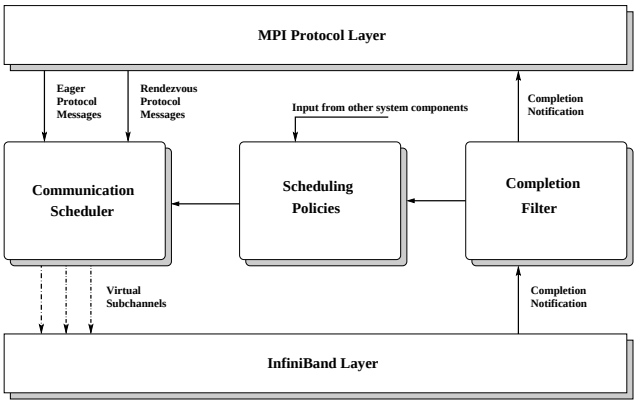

Figure 2. Basic Architecture

The Communication Scheduler is the central part of our design. It accepts protocol messages from the MPI Protocol Layer, and stripes (or multiplexes) them across multiple *virtual subchannels*. (Details of virtual subchannels will be described later.) In order to decide how to stripe or multiplex, the Communication Scheduler uses information provided by the Scheduling Policies component. Scheduling Policies can be static that are determined at initialization time. They can also be dynamic that adjust themselves based on input from other components of the system.

Since a single message may be striped and sent as multiple messages through the InfiniBand Layer, we use the Completion Filter to process completion notifications and to inform the MPI Protocol Layer about completions only when necessary. The Completion Filter can also gather information based on the completion notifications and use it as an input to adjust dynamic scheduling policies.

4.2 Virtual Subchannel Abstraction

Multirail networks can be built by using multiple HCAs on a single node, or by using multiple ports in a single HCA. We have also seen that even with a single port, it is possible to achieve performance benefits by allowing multiple paths to be set up between two end-points. Therefore, it is desirable to have a single implementation to handle all these cases instead of dealing with them separately.

In MPI applications, every two processes can communicate with each other. This is implemented in many MPI

designs by a data structure called *virtual channel* (or *virtual connection*). A virtual channel can be regarded as an abstract communication channel between two processes. It need not necessarily correspond to a physical connection of the underlying communication layer.

In this paper, we use an enhanced virtual channel abstraction to provide a unified solution to support multiple HCAs, multiple ports, and multiple paths in a single port. In our design, a virtual channel can consist of multiple *virtual subchannels* (referred as subchannels from here onwards). Since our MPI implementation mainly takes advantage of the InfiniBand Reliable Connection (RC) service, each subchannel corresponds to a reliable connection at the InfiniBand Layer. At the virtual channel level, we maintain various data structures to coordinate all the subchannels.

It is easy to see how this enhanced abstraction can deal with the all the multirail configurations we have discussed. In the case of each node having multiple HCAs, subchannels for a virtual channel correspond to connections that go through different HCAs. If we would like to use multiple ports of the HCAs, we can set up subchannels so that there is one connection for each port. Similarly, different subchannels/connections can be set up in a single port that follow different paths. Once all the connections are initialized, the same subchannel abstraction is used for communication in all cases. Therefore, there is essentially no difference for all the configurations except for the initialization phase. The subchannel abstraction can also easily deal with cases in which we have a combination of multiple HCAs, multiple ports, and multiple paths in a single port. This idea is further illustrated in Figure 3.

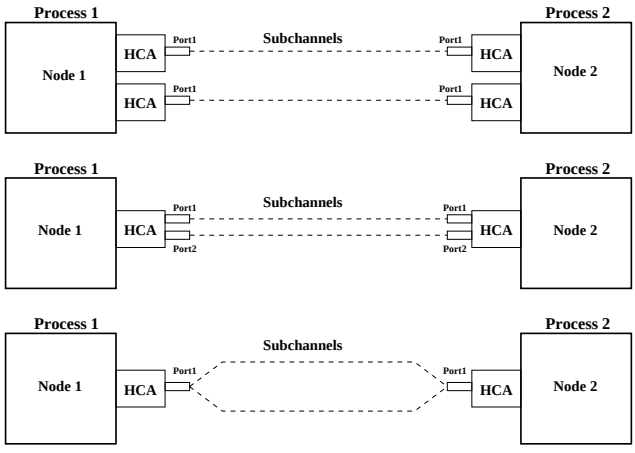

Figure 3. Virtual Subchannel Abstraction

4.3 Integration with MPI protocols

In some MPI implementations, functionalities such as striping messages across multiple network interfaces are part a of the messaging layer. This messaging layer provides an interface to upper layer software such as VMI [21]. One advantage of this approach is portability, as other upper layer software can also benefit from multirail networks. Our design is different because we have chosen to integrate these functionalities more tightly with the MPI communication protocols. Instead of focusing on portability, we aim to achieve high efficiency, performance and flexibility in implementation. Since multirail support is integrated with MPI protocols, we can specifically tailor its design to MPI to reduce overhead. This tightly coupled structure give us more flexibility in controlling how messages are striped or multiplexed in different MPI protocols.

One key design decision we have made is to allow message striping only for large messages, although all messages, including RDMA and send/receive, can use multiplexing. This is not a serious restriction for MPI because MPI implementations over InfiniBand usually only use RDMA operations to transfer large messages. Send/receive operations are often used only for transferring small messages.

By using striping with RDMA, there is almost no overhead to reassemble messages because data gets directly placed into the destination buffer. Zero-copy protocols in MPI, which usually take advantage of RDMA, can be supported in a straightforward manner.

As an example, let's take a look at the Eager and the Rendezvous protocols shown in Figure 1. In the Eager protocol, the data message can be sent using either RDMA or send/receive operations. However, since this message is small, striping will incur extra overhead and hence only multiplexing is used. In the Rendezvous protocol, control messages are not striped. However, the data message can be striped since it is usually large.

4.4 Scheduling Policies

Different scheduling policies can be used by the Communication Scheduler to decide which subchannels to use for transferring each message. We categorize different policies into two classes: *static schemes* and *dynamic schemes*.

- In static schemes, the policy and its parameters are determined at initialization time and stay unchanged during the execution of MPI applications.
- In dynamic schemes, we can switch between different policies or change parameters during the program execution.

In our design, scheduling policies can also be classified into multiplexing and striping schemes. Multiplexing schemes are used for send/receive operations and RDMA operations with small data, in which messages are not striped. Striping schemes are used for large RDMA messages.

For multiplexing schemes, a simple solution is *binding*, in which only one subchannel is used for all messages. This scheme has the least overhead. And it can take advantage of multiple subchannels if there are multiple processes on a single node. In the case of utilizing multiple subchannels for a single process, schemes similar to Weighted Fair Queuing (WFQ) and Generalized Processor Scheduling (GPS) have been proposed in the networking area [1]. These schemes take into consideration the length of a message. In InfiniBand, the per operation cost usually dominates for small messages. Therefore, we choose to ignore the message size for small messages. As a result, simple *round robin* or *weighted round robin* schemes can be used for multiplexing. In some cases, different subchannels may have different latencies. This will result in many out-of-order messages for round robin schemes. To alleviate this problem, a variation of round robin called *window based round robin* can be used. In this scheme, a window size W is given and a subchannel is used to send W messages before the Communication Scheduler switches to another subchannel. Since W consecutive messages travel the same subchannel, the number of out-of-order messages can be significantly reduced for subchannels with different latencies.

For striping schemes, the most important factor we need to consider is the bandwidth of each subchannel. It should be noted that we should consider *path bandwidth* instead of *link bandwidth*, although they can sometimes be the same depending on the switch configuration and the communication pattern. *Even striping* can be used for subchannels with equal bandwidth, while *weighted striping* can be used for subchannels with different bandwidths. Similar to multiplexing, *binding* can be used when there are multiple processes on a single node.

4.5 Adaptive Striping

As we have discussed in the previous subsection, it is important to take into consideration path bandwidth for striping schemes. A simple solution is to use *weighted striping* and set the weights of different subchannels to their respective link bandwidths. However, this method fails to address the following problems:

- Firstly, sometimes information such as link bandwidth is not directly available to the MPI implementation .
- Secondly, in some cases, bottleneck in the network or switches may reduce the path bandwidth in comparison to the link bandwidth.
- Finally, path bandwidth can also be affected by other ongoing communication at the same node. Therefore, it may change over time.

A partial solution to is to carry out a small test during the initialization phase of MPI applications to determine the path bandwidth. However, in addition to its high overhead (tests need to be done for every subchannel between every pair of nodes), it fails to solve the last problem.

In order to solve the last problem, we propose a dynamic scheme for striping large messages. Our scheme, called *adaptive striping scheme*, is based on the weighted striping . However, instead of using a set of fixed weights set at initialization time, we periodically monitor the progress of different stripes in each subchannel and exploit feedback information from the InfiniBand Layer to adjust the weights to their near optimal values.

In designing the adaptive striping scheme, we assume the latencies of all subchannels are about the same and focus on their bandwidth. In order to achieve optimal performance for striping, a key insight is that the message must be striped in such a way that transmission of each stripe finishes at about the same time. This results in perfect load balancing and minimum message delivery time. Our scheme periodically monitors the time each stripe spends in each subchannel and uses this information to adjust the link weights so that the striping distribution becomes more balanced and eventually attains near optimality. This feedback based control mechanism is illustrated in Figure 4.

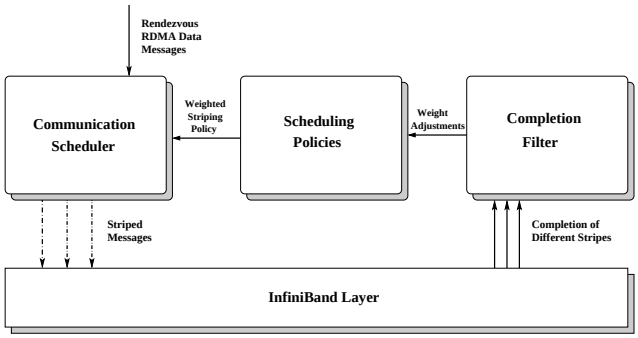

Figure 4. Feedback Loop in Adaptive Striping

In InfiniBand, a completion notification is generated after message delivery to the destination and the corresponding acknowledgment receipt. The Completion Filter of our implementation, helps the progress engine to poll and check for new completion notifications and takes appropriate action. To calculate the delivery time of each stripe, we record the start time for a stripe, when it is handed over to the InfiniBand Layer for transmission. On the finish of the delivery, a completion notification is generated by the InfiniBand Layer. The Completion Filter component then records the finish time and calculates the delivery time . After collecting the delivery time for each message stripe, weights are recalculated and sent to the Scheduling Policies component to adjust the policy. Later, the Communication Scheduler

uses the new policy for striping.

Next we discuss the details of weight adjustment. Our main idea is to have a fixed number of total weights and redistribute them based on feedback information obtained from different stripes of a single message. Suppose the total weight is W_{total}, the current weight of subchannel i is W_i, the path bandwidth of subchannel i is BW_i, the message size is S, and the stripe delivering time for subchannel i is t_i, we then have the following:

$$BW_i = \frac{S \cdot \frac{W_i}{W_{total}}}{t_i} = \frac{S \cdot W_i}{t_i \cdot W_{total}} \quad (1)$$

Since W_{total} and S are the same for all subchannels, we have the following:

$$BW_i \propto \frac{W_i}{t_i} \quad (2)$$

Therefore, new weight distributions can be done based on Equation 2. Suppose W_i' is the new weight for subchannel i, the following can be used to calculate W_i':

$$W_i' = W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in subchannels} \frac{W_k}{t_k}} \quad (3)$$

In Equation 3, weights are completely redistributed based on the feedback information. To make our scheme more robust to fluctuations in the system, we preserve part of the historical information. Suppose α is a constant between 0 and 1, we have the following equation:

$$W_i' = (1-\alpha) \cdot W_i + \alpha \cdot W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in subchannels} \frac{W_k}{t_k}} \quad (4)$$

In our implementation, the start times of all stripes are almost the same and can be accurately measured. However, completion notification is generated by the InfiniBand Layer asynchronously. Hence, we can record the finish time of a stripe only as soon as its completion notification is found. Since MPI progress engine processing can be delayed due to application's computation, we can only obtain an upper bound on the actual finish time and the resulting delivery time t_i is also an upper bound. Therefore, the question arises, how accurately can we estimate the delivery time t_i for each subchannel. To address this question, we consider three cases:

1. Progress engine is not delayed. In this case, accurate delivery time can be obtained.

2. Progress engine is delayed and some of the delivery times are overestimated. Based on Equation 4, in this case, weight redistribution will not be optimal, but it will still improve performance compared to the original weight distribution.

3. Progress engine is delayed for a long time and we find all completion notifications at about the same time. Based on Equation 4, this will essentially result in no change in the weight distribution.

We can see that in no case will the redistribution result in worse performance than the original distribution. In practice, case 1 is the most common and accurate estimation can be expected most of the time.

5 Detailed Design Issues

Our multirail MPI implementation is based on MVAPICH [17, 11], our MPI implementation over InfiniBand. MVAPICH is derived from MPICH [8], which was developed at Argonne National Laboratory and is currently one of the most popular MPI implementations. MVAPICH is also derived from MVICH [10], which is an ADI2 implementation for VIA [5].

In this section, we discuss the detailed design issues in our multirail MPI design. These issues include handling multiple HCAs, out-of-order messages, and RDMA completion notification.

5.1 Handling Multiple HCAs

In Section 4, we described how we can provide a unified design for multiple HCAs, multiple ports, and multiple connections in a single port. The key idea is to use the subchannel abstraction. Once subchannels are established, there is essentially no difference in dealing with all the different cases.

However, due to some restrictions in InfiniBand, there are two situations that must be handled differently for multiple HCAs:

- completion queue (CQ) polling
- buffer registration.

Our MPI implementation uses mostly RDMA to transfer messages and we have designed special mechanisms at the receiver to detect incoming messages [11]. However, CQs are still used at the sender side for completion notification. Although multiple connections can be associated with a single CQ, InfiniBand requires all these connections to be physically linked to a single HCA. Hence, we need to use multiple CQs for multipls HCAs. This results in slightly higher overhead due to the extra polling of CQs.

Buffer registration also needs different handling for multiple HCAs. In InfiniBand, buffer registration serves two purposes. Firstly, it ensures the buffer will be pinned down in physical memory so that it can be safely accessed by InfiniBand hardware using DMA. Second, it provides the InfiniBand HCA with address translation information so that

buffers can be accessed through virtual addresses. Hence, if a buffer is to be sent through multiple HCAs, it must be registered with each one of them. Currently, we have used a simple approach of registering the complete buffer with all HCAs. Although this approach increases the registration overhead, this overhead can be largely avoided by using a registration cache. In future, we plan to investigate schemes that only register a part of the buffer with each HCA.

5.2 Out-of-Order Message Processing

In order to maintain correctness, applications require messages to be processed in a sequential order . Since we use Reliable Connection (RC) transport service provided by InfiniBand for each subchannel, messages are not lost and delivered in order for a single subchannel. However, there is no ordering guaranteed for multiple physical subchannels of the same virtual channel. To address this problem, we introduce a *Packet Sequence Number* (PSN) variable for each virtual channel.This variable is shared by all virtual subchannels of a virtual channel. Every message sent through this virtual channel will carry current PSN and increment it. Each receiver also maintains an *Expected Sequence Number* (ESN) for every virtual channel. When an out-of-order message arrives, it is enqueued on a *out-of-order queue* associated with this virtual channel and its processing is deferred. This queue is checked at proper times when a message in the queue may be the next expected packet.

The basic operations on the out-of-order queue are *enqueue*, *dequeue*, and *search*. To improve performance, it is desirable to optimize these operations. In practice we have found that when appropriate communication scheduling policies are used, out-of-order messages are very rare. As a result, very little overhead is spent in out-of-order message handling.

5.3 RDMA Completion Notification

In our design, large messages which use the Rendezvous protocol are striped into multiple small messages. Hence, multiple completion notifications are generated for each striped message at the sender side. The Completion Filter component in our design notifies the MPI Protocol Layer only after it has collected all the notifications.

At the receiver, the MPI protocol Layer also needs to know when the data message has been placed into the destination buffer. In our original design, this is achieved by using an *Rendezvous finish* control message. This message is received after the RDMA data messages are received, since ordering is guaranteed for a single physical subchannel. However, this scheme is not enough for multiple subchannels. In this case, we have to use multiple Rendezvous finish messages – one per each physical subchannel used for RDMA data transfer. The receiver will notify the MPI Protocol Layer only after it has received all the RDMA finish messages. It should be noted that these Rendezvous finish messages are sent in parallel and their transfer times are overlapped. Therefore, in general they have very small extra overhead.

6 Performance Evaluation

In this section, we evaluate the performance of our multirail MPI design over InfiniBand. Our evaluation consists of two parts. In the first part, we show the performance benefit we can achieve compared to the original MPI implementation. In the second part, we provide an evaluation of our adaptive striping scheme. Due to the limitation of our testbed, we focus only on multirail networks with multiple HCAs in the section.

6.1 Experimental Testbed

Our testbed cluster comprises of 8 SuperMicro SUPER X5DL8-GG nodes with ServerWorks GC LE chipsets. Each node has dual Intel Xeon 3.0 GHz processors, 512 KB L2 cache, and PCI-X 64-bit 133 MHz bus. We have used InfiniHost MT23108 DualPort 4x HCAs from Mellanox. If both ports of an HCA are used, we can potentially achieve one way peak bandwidth of 2 GB/s. However, the PCI-X bus can only support around 1 GB/s maximum bandwidth. Therefore, for each node we have used two HCAs and only one port of each HCA is connected to the switch. The ServerWorks GC LE chipsets have two separate I/O bridges and three PCI-X 64-bit 133 MHz bus slots. To reduce the impact of I/O bus, the two HCAs are connected to seperate PCI-X buses connected to different I/O bridges. All nodes are connected to a single Mellanox InfiniScale 24 port switch MTS 2400, which supports all 24 ports running at full 4x speed. Therefore, our configuration is equivalent to a two-rail InfiniBand network built from multiple HCAs. The kernel version we used is Linux 2.4.22smp. The InfiniHost SDK version is 3.0.1 and HCA firmware version is 3.0.1. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 1 GB of PC2100 DDR-SDRAM.

6.2 Performance Benefits of Multirail Design

To evaluate the performance benefit of using multirail networks, we compare our multirail MPI implementation with the original MPI implementation. In the multirail MPI design, unless otherwise stated, even striping is used for large messages and round robin scheme is used for small messages. We first present performance comparisons using micro-benchmarks, including latency, band-

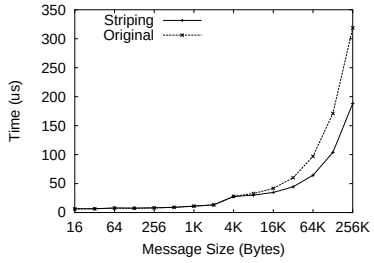

Figure 5. MPI Latency (UP mode)

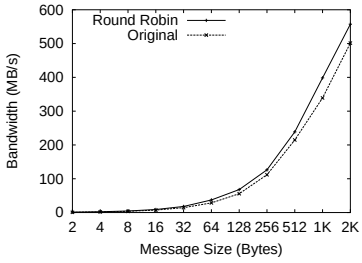

Figure 6. MPI Bandwidth (Small Messages, UP mode)

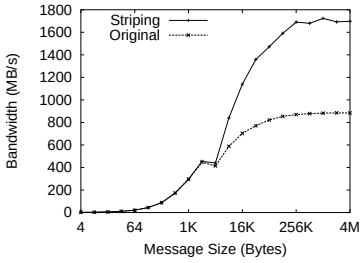

Figure 7. MPI Bandwidth (UP mode)

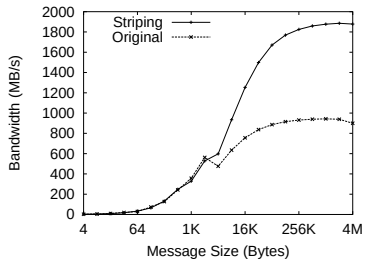

Figure 8. MPI Bidirectional Bandwidth (UP mode)

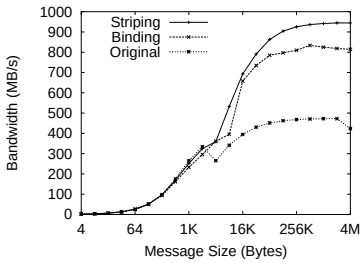

Figure 9. MPI Bandwidth (SMP mode)

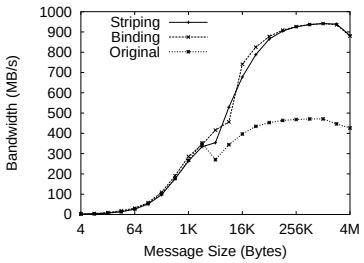

Figure 10. MPI Bidirectional Bandwidth (SMP mode)

width and bi-directional bandwidth. We then present results for some collective communication by using Pallas MPI benchmarks [18]. Finally, we carry out application level evaluation by using some of the NAS Parallel Benchmarks [15] and a visualization application. In many of the experiments, we have considered two cases: UP mode (each node running one process) and SMP mode (each node running two processes).

In Figures 5, 7 and 8, we show the latency, bandwidth and bidirectional bandwidth results in UP mode. We also show bandwidth for small messages in Figure 6. From Figure 5, we can see that for small messages, the original design and the multirail design perform comparably. The smallest latency is around 6 μs for both. However, as message size increases, the multirail design outperforms the original design. For large messages, it achieves about half the latency of the original design. In Figure 7, we can observe that multirail design can achieve significantly higher bandwidth. The peak bandwidth for the original design is around 884 MB/s. With the multirail design, we can achieve around 1723 MB/s bandwidth, which is almost twice the bandwidth obtained with the original design. Bidirectional bandwidth results in Figure 8 show a similar trend. The peak bidirectional bandwidth is around 943 MB/s for the original design and 1877 MB/s for the multirail design. In Figure 6, we can see that the round robin scheme can slightly improve bandwidth for small messages when compared with the original scheme.

For Figures 9 and 10, we have used two processes on each node, each one of them sending and receiving data from a process on the same node. It should be noted in the bidirectional bandwidth test, the two senders are on the same node. For the multirail design, we have shown results using both even striping policy and binding policy for large messages. Figure 9 shows that both striping and binding perform significantly better than the original design. We can also see that striping does slightly better than binding. The reason is that striping can utilize both HCAs in both directions while binding only uses one direction in each HCA. When conducting the bidirectional bandwidth test in SMP mode, both HCAs are utilized in both directions. Hence, striping and binding perform comparably, as can be seen from Figure 10.

In Figures 11, 12, 13 and 14, we show results for MPI_Bcast and MPI_Alltoall for 8 processes (UP mode) and 16 processes (SMP mode) using Pallas Benchmarks. The trend is very similar to what we have observed in previous tests. With multirail design, we can achieve significant performance improvement for large messages compared with the original design.

In Figures 15 and 16, we show application results. We have chosen the IS and FT applications (Class A and Class

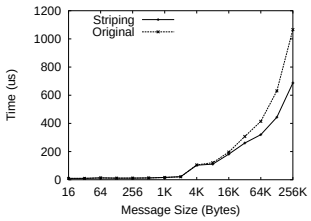

Figure 11. MPI_Bcast Latency (UP mode)

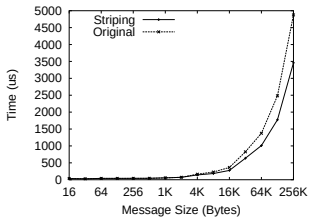

Figure 12. MPI_Alltoall Latency (UP mode)

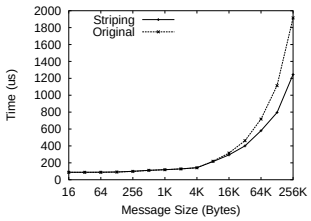

Figure 13. MPI_Bcast Latency (SMP mode)

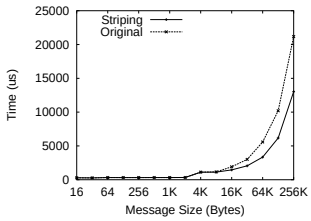

Figure 14. MPI_Alltoall Latency (SMP mode)

B) in the NAS Parallel Benchmarks because compared with other applications, they are more bandwidth-bound. We have also used a visualization application. This application is a modified version of the program described in [6]. We show performance results for both UP and SMP modes. However, due to the large data set size in the visualization application, we can only run it in UP mode. From the figures we can see that multirail design results in significant reduction in communication time for all the applications in both UP and SMP modes. For FT, the communication time is reduced almost by half. For IS, the communication time is reduce by up to 38%, which results in up to 22% reduction in application running time. For the visualization application, the communication time is reduced by 43% and the application running time is reduced by 16%.

Overall, we can conclude that the multirail design can bring significant performance improvement to bandwidth-bound applications.

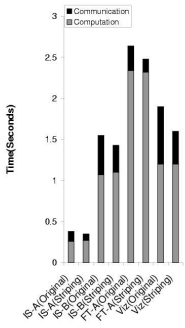

Figure 15. Application Results (8 processes, UP mode)

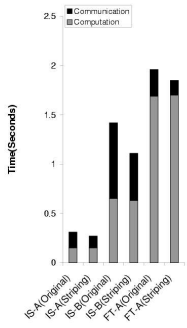

Figure 16. Application Results (16 processes, SMP mode)

6.3 Evaluating the Adaptive Striping Scheme

In this subsection, we show how our proposed adaptive striping scheme provides good performance in case each rail has different bandwidth. This difference can arise due to congestion, link failure or presence of hotspots in the fabric. To simulate this environment, for most of our experiments, we have forced the second HCA on each node to run at 1x speed with a peak bandwidth of 250 MB/s. The first HCA on each node still operates at the normal 4x speed (1 GB/s peak bandwidth). Without *apriori* knowledge of this environment, our multirail MPI implementation will use even striping. With this knowledge, it will use weighted striping and set the weights to 4 and 1 respectively for each subchannel. We compare both of them with the adaptive striping scheme, which assigns equal weights to both subchannels initially. We focus on microbenchmarks and UP mode in this subsection.

Figures 17 and 18 show the latency and bandwidth results. We can see that the adaptive striping scheme significantly outperforms even striping and achieves comparable performance with weighted striping. In Figure 19, we show bidirectional bandwidth results for the three schemes. A surprising finding is that our adaptive scheme can significantly outperform weighted striping in this case. This is because in the test, the communication traffic is assigned to the two subchannels as 4:1 based on the link speed (4x vs. 1x). With bidirectional traffic, the aggregate link speeds would be 8x and 2x respectively for each subchannel. However, the PCI-X bus can only sustain a peak bandwidth of 1 GB/s, which is equivalent to 4x speed. Therefore, if we take into account the I/O bus bottleneck, the speed should be 4x and 2x for the two subchannels, respectively. Hence, the optimal weighted scheme should use 2:1 instead of 4:1. This also shows that even with *apriori* knowledge of link speed, static schemes may fail to achieve optimal performance because of impact from other system components and pattern of communication traffic. In contrast, the adaptive striping scheme can easily adjust the policy to achieve optimal striping.

In the following bandwidth test, we let both HCAs op-

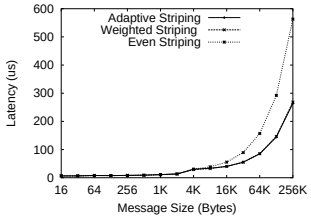

Figure 17. MPI Latency with Adaptive Striping (UP mode)

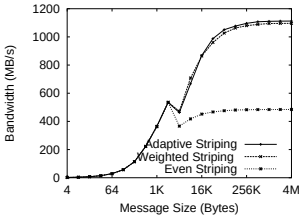

Figure 18. MPI Bandwidth with Adaptive Striping (UP mode)

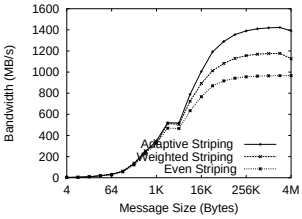

Figure 19. MPI Bidirectional Bandwidth with Adaptive Striping (UP mode)

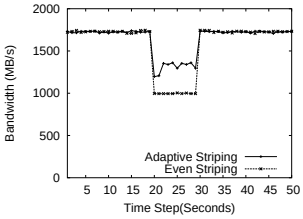

Figure 20. MPI Bandwidth with Adaptive Striping

erate at 4x speed. A bandwidth program runs on the two nodes and prints out the peak bandwidth results every one second. During the execution of the bandwidth program, we start another program on the two nodes which use the second HCA to transfer large messages. This program runs for around 10 seconds. We compare the adaptive striping scheme and even striping in Figure 20. We can see that at the beginning both schemes perform comparably. However, when the second program starts, one of the HCAs has to be shared by both programs. Hence, even striping is no longer optimal. As we can see, the adaptive scheme can achieve better performance by adjusting the weight of each subchannel accordingly. After the second program finishes, the adaptive striping scheme can again readjust the weights to achieve peak performance.

7 Related Work

Using multirail networks to build high performance clusters is proposed in [4]. The paper proposed different allocation schemes in order to eliminate conflicts at end points or I/O buses. However, the main interconnect focused in the paper was Quadrics [20] and the performance evaluation was done using simulation. In this paper, we focus on software support at the end points to build InfiniBand multirail networks and present experimental performance data.

VMI2 [21] is a messaging layer developed by researchers at NCSA. An MPI implementation over VMI2 is also available [16]. VMI2 runs over multiple interconnects. [21] briefly mentions VMI2's ability to stripe large messages across different network interconnects. Instead of using a separate messaging layer, our design has integrated the multirail support with MPI protocols.

LA-MPI [7] is an MPI implementation developed at Los Alamos National Labs. LA-MPI was designed with the ability to stripe message across several network paths. LA-MPI design includes a *path scheduler*, which determines which path a message will use for transfer. This design bears some similarity with our approach. However, in this paper we focus on InfiniBand architecture and discuss different design issues and policies. We have also proposed an adaptive striping scheme .

SGI's Message Passing Toolkit [22] and Sun's MPI implementation over its SUN Fire Link [23] both support the ability of striping message across multiple links. However, details about how striping is implemented are not available in the literature. Recently, Myricom announced its message passing layer called *Myrinet Express* [14]. This software can stripe messages across two different ports on a single Myrinet NIC and overcome the limitation of Myrinet link bandwidth. However, Myrinet Express has not been released and its internal design are not yet available.

Striping in the network systems has been used for many years. [3] provides a survey of how striping is used at different layers in the network subsystems. Work done in [1] proposes an architecture to stripe packets across multiple links in order to achieve fair load sharing. Striping across multiple TCP/IP connections has also been studied in the literature. One example is PSockets [24]. PSockets presents to the application layer the same socket interface as that used in TCP/IP. It transparently stripes message across multiple TCP/IP connections.

8 Conclusions and Future Work

In this paper, we presented an in-depth study of designing high performance multirail InfiniBand clusters. We discussed various ways of setting up multirail networks with InfiniBand and proposed a unified MPI design that can support all these approaches. By taking advantage of RDMA operations in InfiniBand and integrating the multirail design with MPI communication protocols, our design supported multirail networks with very low overhead.The design also supported different policies of using multiple rails. Another contribution of this paper is an adaptive striping scheme that can dynamically change the striping parameters based on the current available bandwidth of different rails.

We also implemented our design and carried out detailed performance evaluation. Our performance results have shown that the multirail MPI can significantly improve MPI communication performance. With a two rail InfiniBand network, we can achieve almost twice the bandwidth and half the latency for large messages in comparison with the original MPI implementation. The multirail MPI design can also significantly reduce communication time as well as running time for bandwidth-bound applications. We have also shown that the adaptive striping scheme can achieve excellent performance without *apriori* knowledge of the bandwidth of each rail.

In future, we plan to carry out experiments on large scale testbeds and study the impact of our design and different policies on applications. We would also like to evaluate our design using PCI Express systems by setting up multirail networks using multiple HCA ports. we also plan to work on optimizing collective communication by taking advantage of multirail networks. An important usage of multirail networks is to achieve network fault tolerance. We also plan to investigate this direction in the near future.

References

[1] H. Adiseshu, G. M. Parulkar, and G. Varghese. A reliable and scalable striping protocol. In *SIGCOMM*, pages 131–141, 1996.

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

[3] C. Brendan, S. Traw, and J. M. Smith. Striping within the network subsystem. *IEEE Network*, 9(4):22, 1995.

[4] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits. Using Multirail Networks in High-Performance Clusters. *Concurrency and Computation: Practice and Experience*, 15(7-8):625, 2003.

[5] Compaq, Intel, and Microsoft. VI Architecture Specification V1.0, December 1997.

[6] J. Gao and H.-W. Shen. Parallel View-Dependent Isosurface Extraction Using Multi-Pass Occlusion Culling. In *Proceedings of 2001 IEEE Symposium in Parallel and Large Data Visualization and Graphics*, pages 67–74, October 2001.

[7] R. L. Graham, S.-E. Choi, D. J. Daniel, N. N. Desai, R. G. M. C. E. Rasmussen, L. D. Risinger, and M. W. Sukalski. A network failure tolerant message passing system for terascale clusters. In *16th Annual ACM International Conference on Supercomputing (ICS '02)*, June 2002.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.1. http://www.infinibandta.org, November 2002.

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

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

[12] Mellanox Technologies. Mellanox InfiniBand InfiniHost MT23108 Adapters. http://www.mellanox.com, July 2002.

[13] Mellanox Technologies. Mellanox InfiniBand InfiniHost III Ex MT25208 Adapters. http://www.mellanox.com, February 2004.

[14] Myricom. Myrinet. http://www.myri.com/.

[15] NASA. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Software/NPB/.

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

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

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

[19] PCI-SIG. PCI Express Architecture. http://www.pcisig.com.

[20] F. Petrini, W. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network: High-Performance Clustering Technology. *IEEE Micro*, 22(1):46–57, 2002.

[21] S. Pakin and A. Pant. VMI 2.0: A Dynamically Reconfigurable Messaging Layer for Availability, Usability, and Management. In *SAN-1 Workshop (in conjunction with HPCA)*, Febuary 2002.

[22] SGI. SGI Message Passing Toolkit. http://www.sgi.com/software/mpt/overview.html.

[23] 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.

[24] H. Sivakumar, S. Bailey, and R. L. Grossman. PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks. In *Supercomputing*, 2000.