Network Fault Resilient MPI for Multi-Rail Infiniband Clusters

A Thesis

Presented in Partial Fulfillment of the Requirements for the Degree Master of Science in the Graduate School of The Ohio State University

By

Siddhesh Pai Raikar, B. E.

Graduate Program in Computer Science and Engineering

The Ohio State University

2011

Master’s Examination Committee:

Dr. D. K. Panda, Advisor

Dr. P. Sadayappan
Abstract

In the past few years the number of cores on processors has increased at an alarming rate. To make an efficient use of these machines it is necessary to provide the required bandwidth. But this is hampered by the physical capacity of the medium which can allow data transfers only up to certain extents. To keep up with the multicore trend and bridge this gap between computation rate and the communication bottleneck, many supercomputers and clusters now-a-days are equipped with multiple network cards (rails) to provide a very high data transfer capability. The T2K cluster (Tsukuba) in Japan provides 4 network cards per node on their entire 648 node (10,368 cores) system. On the other hand there are many existing systems which use multiple rails per node but each on a different subnet to provide fault tolerance in case of network or other failures. The prime objective for these systems is to provide for switching over from one network to the other in case an error is encountered on the network.

This dissertation presents a design framework to achieve both these goals: Performance and Fault Tolerance, in a multi-rail scenario, but without compromising one over the other. In a general message passing scenario, whenever there is a network fault the entire operation aborts. This thesis presents an implementation, with which, it is possible in multi-rail clusters to continue operations even with network failures. In case of network component recovery it is also possible to re-establish connections
between the failed components and start using them once again. We show that our
implementation provides very little overhead and is able to deliver high performance.
Also the recovery is immediate and is associated with no additional overhead. We also
show sustenance of the design by running application benchmarks with permanent
failures. From the results we see that we are able to provide sustenance as well as
performance benefits along with fault resiliency with this design.
Dedicated to my family
Acknowledgments

I want to take this opportunity to thank my adviser Dr. D. K. Panda for the guidance and support he has given to me throughout my master degree. He has been a source of inspiration for me I know that his invaluable teachings will last with me for a long time.

I also want to thank Dr. P. Sadayappan for agreeing to be a committee member at my defense and make it work in spite of his tight schedule.

I would like to thank all my seniors Sreeram Potluri, Krishna Kandalla, Miao Luo, Xiangyong Ouyang and my co-workers and very good friends Jithin Jose, Raghunath R., Nishanth Dandapanthula, Vilobh Meshram, Devendar Bureddy who have always supported me and helped me get through difficult times. I want to thank Jerome Vienne for his good timely advice in both, curricular as well as personal matters. A special mention goes to our network administrator and friend Mark Arnold, who would at times go to lab on a Sunday just to reboot a node that I had taken down, and Jonathan Perkins, not only for his help but also for the long conversations we had in his office about everything under the sun. But most importantly I am grateful to my mentor Hari Subramoni who has brought me up to this point where I currently am. His guidance and encouragement helped me shape up my masters thesis and also helped me learn a lot.
In these last few years I made and bid farewell to a lot of friends here at Ohio State but some of them made lasting impressions and I will never forget them: Aarthi, Ajay, Amanda, Artika, Isha, Jennifer, Kim, Praneeth, Satyajeet, Srinivas, Sushma, Zoya; to name a few. I owe all of you one. I want to thank my best friends Divya, Fennex and Prachee who I have known for a very long time and have supported me and put up with me.

Lastly I want to thank my parents (Mrs. Sunita Pai Raikar and Mr. Prakash Pai Raikar), my sister (Ashwini), uncle (Ashok) and uncle (Satish Pai) without whom any of this would never have been possible. I am also very grateful to God, Dada Maharaj and Vasudev Mama for their blessings and grace throughout my journey and helping me reach this pedestal in my life.
Vita

November 26, 1985 ....................... Born - Mumbai, India

2007 ......................................... B.E., Computer Engineering,
                              University of Mumbai,
                              Mumbai, India.

2007-2009 ................................. Software Engineer,
                              Infosys Technologies Limited,
                              Bangalore, India

2010-2011 ................................. Graduate Research Associate,
                              The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering

# Table of Contents

Abstract .............................................................. ii

Dedication ............................................................. iv

Acknowledgments ......................................................... v

Vita ................................................................. vii

List of Tables ............................................................ x

List of Figures .......................................................... xi

1. Introduction ......................................................... 1
   1.1 InfiniBand - An Overview ...................................... 2
   1.2 MPI .............................................................. 3
   1.3 MVAPICH and MVAPICH2 ......................................... 3
   1.4 Motivation ........................................................ 4
   1.5 Problem Statement ............................................... 4
      1.5.1 Enhanced rail selection policies ........................... 5
      1.5.2 Failover design with Multi-rail ........................... 6
      1.5.3 Our Approach ............................................... 7
   1.6 Organization of Thesis .......................................... 8

2. Background and Related Work ...................................... 10
   2.1 Multi-rail Clusters ............................................ 10
   2.2 Scheduling policies ............................................ 11
   2.3 A note on Fault tolerance ...................................... 12
      2.3.1 Failover ...................................................... 12
   2.4 InfiniBand primitives ........................................... 13
2.5 Related Work ................................................. 14

3. Enhanced design for Multi-Rail Configurations .................. 15
   3.1 Basic Scheduling in Multi-rail networks ..................... 16
   3.2 Implementation details ..................................... 16
   3.3 Evaluation and Experiments ................................. 19
      3.3.1 Experimental setup .................................... 19
      3.3.2 Performance of Various Rail Scheduling policies .... 20

4. Design for implementing Failover technique ....................... 28
   4.1 Network Failures ........................................... 29
      4.1.1 Failure Notifications .................................. 30
   4.2 Failure Detection .......................................... 31
      4.2.1 Remote Failures ....................................... 32
      4.2.2 Local Failures .......................................... 33
      4.2.3 New rail selection and policies ....................... 34
   4.3 Recovery from failures ..................................... 35
      4.3.1 Notification of Recovery ............................... 35
      4.3.2 Corrective actions ..................................... 36
      4.3.3 Recovery Protocol ...................................... 37
   4.4 Experimental Results ....................................... 40
      4.4.1 Experimental Setup ..................................... 40
      4.4.2 Point to point operations .............................. 41
      4.4.3 Collective Operations ................................ 49
      4.4.4 Application benchmarks ................................. 49

5. Conclusion and Future Work .................................... 54
   5.1 Summary of contributions .................................. 54
      5.1.1 Enhanced multi-rail scheduling policies ............... 54
      5.1.2 Failover with multi-rail ............................... 55
   5.2 Future Work ................................................ 55

Bibliography ...................................................... 57
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Evaluation with NAS BT benchmark</td>
</tr>
<tr>
<td>4.2</td>
<td>Evaluation with NAS LU benchmark</td>
</tr>
</tbody>
</table>
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Multirail Scheduling</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Multi-rail Scheduling policies</td>
<td>18</td>
</tr>
<tr>
<td>3.3 osu_bw - Small Messages</td>
<td>20</td>
</tr>
<tr>
<td>3.4 osu_bw - Large Messages</td>
<td>21</td>
</tr>
<tr>
<td>3.5 osu_latency - Small Messages</td>
<td>22</td>
</tr>
<tr>
<td>3.6 osu_latency - Large Messages</td>
<td>23</td>
</tr>
<tr>
<td>3.7 osu_mbw_mr - Bandwidth for Small Messages</td>
<td>24</td>
</tr>
<tr>
<td>3.8 osu_mbw_mr - Bandwidth for Large Messages</td>
<td>25</td>
</tr>
<tr>
<td>3.9 osu_mbw_mr - Message Rate for Small Messages</td>
<td>26</td>
</tr>
<tr>
<td>3.10 osu_mbw_mr - Message Rate for Large Messages</td>
<td>27</td>
</tr>
<tr>
<td>4.1 Rail State Diagram</td>
<td>35</td>
</tr>
<tr>
<td>4.2 VC State Diagram</td>
<td>37</td>
</tr>
<tr>
<td>4.3 Failover recovery timeline</td>
<td>38</td>
</tr>
<tr>
<td>4.4 osu_mbw_mr at 64 processes with Failover</td>
<td>42</td>
</tr>
<tr>
<td>4.5 osu_bw at message size 4MB with Failover</td>
<td>44</td>
</tr>
</tbody>
</table>
4.6  osu_mbw_mr at message size 64K with Failover . . . . . . . . . . . . . . . . . . . . . 45
4.7  osu_mbw_mr Messages/s at size 64K with Failover . . . . . . . . . . . . . . . . . . . . 46
4.8  osu_mbw_mr at message size 1M with Failover . . . . . . . . . . . . . . . . . . . . . 47
4.9  osu_mbw_mr Messages/s at size 1M with Failover . . . . . . . . . . . . . . . . . . . . 48
4.10 mpiBench - Bcast at message size 128K with Failover . . . . . . . . . . . . . . . . 50
4.11 mpiBench - Reduce at size 256K with Failover . . . . . . . . . . . . . . . . . . . . . 51
4.12 mpiBench - All-to-all at size 8K with Failover . . . . . . . . . . . . . . . . . . . . . 52
Chapter 1: INTRODUCTION

In this era of Exascale Computing, High Performance Computing (HPC); which at some point may have been designed for, and to an extent kept restricted only to complex scientific and mathematical applications, has become an integral component of almost every aspect of today’s life. Be it Stock Markets, Astronomy, Defense, Banks, Meteorology or Shopping, High Performance Computing has become a commonplace term. HPC integrates Computer Networks, Parallel Programming, Algorithms, Security and many more into one discipline whose primary goal is to provide high availability and reliability along with enormous computing power. These systems employ thousands of computing units connected over a very high-speed interconnect making maximum use of the computing power at hand to provide Peta/Exa flops \((10^{15}/10^{18} \text{ calculations per second})\) of performance. However, to extract the maximum available computational ability it is very much desirable that these machines also communicate with each other at a rapid rate. This is provided by the interconnect over which these machines are connected. Interconnection networks have also rapidly evolved to offer low latencies and high bandwidths to meet the communication requirements of distributed computing applications. InfiniBand, an interconnect specially designed to cater to such HPC clusters and grids provides very high throughput and at the same time, maintaining very low latency in data transfer operations. According to the
Top500 [17] ratings of supercomputers done in Nov’11, 41.80% of the top 500 most powerful supercomputers in the world are based on the InfiniBand interconnects.

1.1 InfiniBand - An Overview

InfiniBand Architecture (IBA) [8] defines a switched network fabric for inter-connecting processing nodes and I/O nodes, using a queue-based model. The I/O devices and compute nodes are connected to the switch fabric using one or more channel adapters (CA). Every fabric may consist of one or more subnets which are interconnected by routers. Each subnet is managed in an autonomous way. InfiniBand specifications do not enforce the usage of any specific network topology or routing algorithm. IB has multiple transport services including Reliable Connection (RC) and Unreliable Datagram (UD), and supports two types of communication semantics: Channel Semantics (Send-Receive communication) over RC and UD, and Memory Semantics (Remote Direct Memory Access - RDMA communication) over RC. Both semantics can perform zero-copy data transfers, i.e, the data can directly be transferred from the application source buffers to the destination buffers without additional host level memory copies. IB also proposes link layer Virtual Lanes (VL) that allows the physical link to be split into several virtual links, each with their specific buffers and flow control mechanisms. This possibility allows the creation of virtual networks over the physical topology. However, current generation InfiniBand interfaces do not offer performance counters for different virtual lanes.
1.2 MPI

The Message Passing Interface (MPI) [10] has been the de-facto parallel programming model for the past two decades. MPI has been widely adopted across various scientific domains and is very successful in implementing regular, iterative parallel algorithms with well defined communication patterns. The current MPI-2 version supports a lot of advanced features such as one-sided communication, dynamic process management and advanced datatype processing. There is a modernization effort on-going for MPI version 3.0.

1.3 MVAPICH and MVAPICH2

The MVAPICH (for MPI-1) and MVAPICH2 (for MPI-2) open-source libraries [1] have been designed over the last ten years to take advantage of modern InfiniBand, 10GigE/iWARP and RDMA over Converged Enhanced Ethernet (RoCE) networking technologies/protocols. These libraries currently deliver high performance and scalability for large-scale (100K+ cores) clusters. Currently, these libraries are being used by more than 1,820 organizations worldwide (in 65 countries). During the last ten years, there have been more than 90,000 downloads of these libraries from the OSU web site alone. These libraries are also available from the software stack of many server and networking vendors, including the widely used OpenFabrics software stack [2]. These libraries are empowering many InfiniBand clusters in the TOP500 list (including #5, #7 and #25 in the latest Nov ’11 TOP500 ranking). These libraries are also used in many large production environment clusters including the TACC Ranger system (62,976 cores).
1.4 Motivation

Network faults that can occur at any point in an interconnect, are at most times transient or easily recoverable. Failover is primarily defined as automatic switching to a redundant network or system. It is always preferable if we can use all the available resources for maximum performance. The redundant networks used in failover are not used for regular performance purposes but only serve as a back up in an event of failure.

Also currently in any communication scenario if any network fault is encountered, the entire operation comes to a halt and completely aborts. If the configuration consists of a multi-rail cluster we might be able use the other rails that are active and try to take a job to completion.

The ultimate goal of this work is to try to provide an efficient failover mechanism for multi-rail clusters which makes use of the existing network configuration and in some manner eliminates the necessity for a redundant network, which is meant only for fault tolerance purposes, as far as transient network faults are concerned. In case of permanent failures we still want the job to reach completion instead of a total abort.

1.5 Problem Statement

Along with high performance, InfiniBand provides specifications for fault tolerance using the failover mechanism. However, we need a unified solution high speed communication and network fault tolerance which semantically abides the MPI layer with the InfiniBand features. Hence the question we address in this dissertations is:
How can we design a high-performance and multi-rail fault resilient system that can tolerate transient or permanent failures and recover from them as and when possible and take the job to completion?

The implementation should provide an efficient utilization of multi-rail configurations in the absence of network faults. In case of the occurrence of network faults, this implementation should provide error detection and recovery from them. This leads us to the following challenges, which we will address in this dissertation.

- Can we utilize the maximum available bandwidth and try to eliminate the need for redundant subnets?
- Can we recover quickly from faults and continue operations without aborting?
- How can we provide a recovery mechanism for transient/reparable issues?
- To be precise, how can we efficiently design a failover module which: Efficiently utilizes network bandwidth, Unifies the concept of High Performance and Fault Tolerance, Easily resumes normal operations without overhead, Takes a job to completion despite permanent failures and also provides optimum performance on failures.

1.5.1 Enhanced rail selection policies

There are several existing implementations of various rail scheduling policies. The approaches are sometimes aimed towards high availability where every process is given the capability of using as many rails that are available to it. With the availability of a higher number of rails it is also implied that every process will spend a higher amount of time in deciding which rail the data can be sent on. Although this may
be desirable to provide higher bandwidth, it is associated with a cost. For every send operation performed by a process it needs to select from a pool of available rails. The cost associated with a single select operation may be small but as the number of messages sent go on increasing, the cost is likely to become higher. Sometimes, it may be desirable that a process is allowed to send only on a particular rail. This may depend on the application in use, the behavior of the algorithm, complexity of other available rail selection techniques or even the topology of the network.

Our Approach

As a part of this thesis, we want to implement a rail binding strategy for rail selection on multi-rail clusters where every process is allowed to send data only on a pre-assigned rail. In other words a process should be 'bound' to a particular rail. We perform some experimental evaluations, on a few benchmarks and justify our approach. The rail binding scheme sometimes is expected to provide a better performance than the rail-sharing schemes because the library will not spend time in selecting the rail each time for every send-receive operation. Also it is desirable that the user should be allowed the flexibility to select particular rails for particular processes. This is likely to help where the user understands the system, its limitations and the benchmarks well. Thus, along with providing default binding schemes we also want to provision for a user input string.

1.5.2 Failover design with Multi-rail

Several attempts have been made at providing fault tolerance to multi-rail clusters. Sun’s HPC environment provides some support for failure on their MPI implementation. The Supercomputers developed by Appro have a mechanism to send data along
other channels in case of failures. Sometimes it is likely that these failures are inter-
mittent and the failed hardware might respond after we have decommissioned a rail. It is desirable that there exists some mechanism which will detect these recoveries and take the appropriate steps to resume normal operations and thus provide high bandwidth.

The number of nodes in a cluster is increasing at a very rapid rate and thus the Mean-Time-Between-Failures (MTBF) is also on the increase. The bigger the cluster the greater the number of nodes, the more more network components and hence more failures.

Also, there could a large job running on a cluster and the problem could be as small as a cable fault. But the entire operation will abort and the cluster remains unusable till the fault is corrected along with a wastage of time with valuable resources and the added overhead of re-running the job again. It will be desirable that such failures are seamlessly handled by software, and continue normal operation and not just abort completely.

1.5.3 Our Approach

In this thesis, we try to come up with a protocol for network fault-tolerance which is not very sensitive to hardware failures and will provide performance till the last working rail is good to be used. We also want to ensure that if a particular rail fails, it is not selected for any further operation until it is restored. Also in case of recovery, the new rail ideally should be brought back into the pool of available rails and should now be used to send data. In case the failure is not transient, then the library should
proceed to completion. However it is important that after recovery, we re-establish all the connections on that rail before we start using it to send actual data.

The user may also have employed different rail scheduling policies for different message sizes and thus it is also desired that the same policy is applied even in the absence of the failed rail. We try to develop a protocol which re-establishes connections using a series of exchange messages.

Finally all of this should be done by not letting performance degrade more than what the performance would be if only the active working rails are put to use. We also want to verify this with experimental results.

1.6 Organization of Thesis

The remainder of this thesis is organized as follows. In chapter 2, We look at the different multi-rail design configurations and study their benefits. Then we look at some examples. Then we look at different rail scheduling policies. Further we take a brief look at the various fault-tolerance techniques and understand the concept of failover. Also we touch upon a few InfiniBand primitives that will help us achieve our goal. We finally discuss the other work from a few other authors that pertain to our area of fault tolerance on multi-rail clusters. Chapter 3 focuses on our enhanced design for rail-scheduling policies on multi-rail clusters. We implement a rail-binding scheme and discuss its performance characteristics. This new scheme provides more flexibility to the user in allocating rails to processes and thereby improve on performance. In Chapter 4 we present the design for a failover mechanism on multi-rail clusters. We first talk about failures in the multi-rail and high performance computing context and then propose a design that provides a good tolerance to failures and does not affect
performance. We evaluate and analyze the working of our design in various scenarios by simulating failures on network components findings. Finally we summarize our conclusions and possible future work in Chapter 5.
Chapter 2: BACKGROUND AND RELATED WORK

In this chapter we provide a brief background related to our work. We begin with a brief introduction to Multi-rail clusters and look at some examples. Then we discuss some scheduling policies for multi-rail configurations. This is followed by a brief discussion on fault tolerance and the different flavors it comes in. We briefly touch failover in a generic sense and then describe how it fits into our context. Later we discuss a few InfiniBand primitives that help us in our approach. Finally we take a look at the past and ongoing related work in our area.

2.1 Multi-rail Clusters

Multi-rail clusters help bridge the computation and communication bottleneck gap by providing multiple HCAs (rails) on clusters. With multiple rails, it becomes possible for multiple processes to send out data simultaneously thereby providing a significant improvement in bandwidth. Many clusters now-a-days are equipped with multiple rails and we take a look at some of them.

• T2K: The T2k cluster is Japan is a 10,368 cores (648 nodes) cluster from Appro International [16]. Every node is equipped with InfiniBand 4xDDR cards making it a Quad-rail high performance SAN. It provides a peak performance of 95.4 Tflops/s [18].
• Gordon: The Gordon cluster at SDSC is a joint venture by Appro and SDSC consisting of 1,024 compute nodes and 64 I/O nodes. Each compute node contains two 8-core 2.6 GHz Intel EM64T Xeon E5 (Sandy Bridge) processors and 64 GB of DDR3-1333 memory. The network topology is a 4x4x4 3D torus with adjacent switches connected by three 4x QDR InfiniBand links (120 Gbit/s). Compute nodes (16 per switch) and I/O nodes (1 per switch) are connected to the switches by 4x QDR (40 Gbit/s). The theoretical peak performance of Gordon is 341 TFlop/s [6].

2.2 Scheduling policies

In multi-rail InfiniBand clusters, the rail selection for sending out data is done via a scheduler. To achieve maximum performance it is critical to employ various scheduling policies which will select the optimal rail for a particular data transfer. The choice of the best available rail and thus the scheduling policy depends on a lot of factors such as the number of processes, message sizes, hardware constraints, communication patterns, etc. There is also a possibility of poor latency performance if the logic for scheduling is complex. Hence scheduling policies play an important role in multi-rail message passing.

There are numerous scheduling policies in practice and each one performs differently for different systems. E.g. For utilizing multiple sub-channels for a single process, schemes similar to Weighted Fair Queuing (WFQ) and Generalized Processor Scheduling (GPS) have been proposed in the networking area by [3]. For small message sizes, simple round robin or weighted round robin schemes can be used for
multiplexing. For large message sizes, Even Striping - for sub-channels with equal bandwidth or Weighted Striping - for sub-channels with different bandwidths.

We will discuss some of these scheduling policies in detail in Chapter 3 and show results with implementing various process binding policies.

2.3 A note on Fault tolerance

With the increase in size of clusters and thereby an increase in the components comprising the cluster, the possibility of an application being interrupted by a failure during its execution becomes so high that fault-tolerance has become a necessity. There are various approaches to Fault Tolerance that have been proposed by many scientists. Checkpoint/Restart (C/R) has been a common practice to guarantee fault-tolerance for large scale applications [14] [11] [15] and also on parallel file systems [4]. Job/process migration, a pro-active fault tolerance mechanism has also gained popularity in recent times.

Fault tolerance via enhanced RC semantics in the form of Network Fault Resiliency [9] is also a well known means for providing fault tolerance. There are various MPI Implementations that specialize in fault tolerance as well such as FT-MPI which can survive the crash of n-1 processes in a n-process job, and, if required, can re-spawn/restart them [5].

Failover is a technique for fault tolerance which uses redundant network component to recover for failures and has been described in detail in 2.3.1.

2.3.1 Failover

Failover, in the generic sense, refers to automatic switching to standby servers, system or networks whenever an abnormal termination occurs of previously active
components. A failover mechanism is generally very essential for systems requiring a high-reliability and also a high-availability. In terms of network failover, many multi-rail systems come equipped with a separate subnet only meant for fault tolerance via redundant network components. It is up to the network designer to employ a system with failover which can be automatic or may require human intervention. An alternative to failover features in recent times is sometimes given by virtualization software which reduces the failover reliance on physical hardware.

2.4 InfiniBand primitives

The Infiniband verbs [8] interface provides several primitives at the user-level to access kernel modules and retrieve information from them. We will discuss some primitives that will help us realize our design and aid in gathering information from the network.

- \textit{ibv.poll\_cq}: This primitive polls the Completion Queue (CQ) for work completions. The completion entry contains all the information about the two communicating nodes. Every entry has a status flag associated with it. The status flag is 0 on a successful completion else it returns with an error code which specifies the kind of failure that occurred.

- \textit{ibv.create\_qp}: \texttt{ibv\_create\_qp} is used to create a Queue Pair between two rails of two communicating processes. In our recovery protocol we use this primitive to create new queue pairs.

- \textit{ibv.modify\_qp}: \texttt{ibv\_modify\_qp} is used to change the state of the qp to a different state e.g. READY-TO-SEND, READY-TO-RECEIVE, etc.
• *ibv_query_port*: This primitive returns the port attributes of a device (HCA). It contains a flag *state* which shows the state of the port as DOWN or ACTIVE.

### 2.5 Related Work

Fault tolerance has been an important topic in HPC research and a lot of work has gone into providing numerous solutions to the problem of failures. Koop et. al. [9] provide a good end-to-end reliability semantics with a message queueing mechanism guaranteeing end-to-end delivery. OpenMPI [12] provides for numerous fault tolerance features such as resending unacknowledged messages periodically, checksum calculation and CR. IBM’s Low Latency Messaging [7] provides support for failover features by providing notification events about failures in the network. Sun’s HPC [13] allows for resending failed messages on other rails, but does not talk about the library’s behaviour when the failed components resume operation.
Chapter 3: ENHANCED DESIGN FOR MULTI-RAIL CONFIGURATIONS

The primary reason for the popularity of InfiniBand is its ability to provide very high bandwidth and thus very high performance. However with ever increasing number of cores, the need for increasing the number of rails also has increased. We have observed that even with the increase in the number of rails it is also important to allow the user to efficiently use these rails depending on the application that it is being used for. Thus having an efficient rail scheduling policy also is of prime importance. If the behavior of the application is properly understood we can achieve significantly better results by using an efficient rail selection policy.

There are various scheduling policies in existence such as the USE_FIRST policy which always selects a default rail. This is useful for time critical applications since it provides the least latency. Another policy: ROUND_ROBIN, allows for the rails to be selected in a round-robin manner each time a message needs to be sent. STRIPING: which is used to split large messages and evenly or unevenly distribute them over multiple rails to provide a better bandwidth is very widely used.

In this chapter we discuss an enhanced design; for rail selection policies, that provides the user more flexibility in scheduling data over multiple rails. We show that we can deliver a better performance if an effective rail selection policy is employed.
The chapter is organized as follows: In section 3.1, we explain a basic design for multi-rail scheduling. We then discuss our enhanced schemes in section 3.2 for a much more effective utilization of rails. Section 3.3 presents performance results depicting the usage of the multi-rail policies. Finally in section 3.4, we summarize our observations.

3.1 Basic Scheduling in Multi-rail networks

The basic scheduling mechanism with multi-rail networks is shown in Figure 3.1. This primarily represents the working on the sender’s side. For sending messages over multiple rails a scheduler is used to select a particular rail which will send out the data. The data is fed to the scheduler by the library and it is up to the scheduler to decide which rail the given data will be sent on. The scheduler decides which rail to send on by taking into account many factors such as the message size, RDMA capabilities and most importantly the scheduling policy specified by the user. The multi-rail network may be built using multiple adapters on a single node, or by using multiple ports in a single HCA or multiple virtual sub-channels over a single port. The scheduler will also take care of all these implementations.

3.2 Implementation details

The different rail scheduling policies that are implemented in our MPI library are shown below in figure 3.2. The scheduling policies are broadly classified into two types. Rail Sharing and Rail Binding.

Rail Sharing policies are those in which the scheduler shares the rails amongst all the processes by assigning any rail to a process from a pool of available rails. All the processes have all the rails available to them and hence the scheduler can select
any one of these rails and assign it to the process to send the data on for a given send-receive operation. It can also decide to distribute the data over multiple rails and send it across in parallel.

The enhanced design now features a Rail Binding scheme; where a particular rail is permanently bound to a particular process for all the send operations conducted by that process. In other words the process is unaware of an existence of other rails and will always invariably select only the assigned rail. This approach is effective when there are only small messages that are required to be sent throughout the operation as the process does not waste time selecting the rail since it has only one rail available to it.
By default the processes are bound to the rails in a round-robin manner depending on the number of rails. However we also allow the user to decide which rail is be to mapped to which process. This provides an added sense of flexibility to the user. This could be particularly effective in some scenarios where a user is aware of specific communication patterns in the application and wants to evenly distribute the rail usage amongst the processes to avoid bottlenecks. It could also be useful if each of the rail provides a different bandwidth and thus the user can unevenly distribute the processes to maximize performance. A few examples can be given with numbers. The user defined policies are of the type: BUNCH, SCATTER and a CUSTOM string. They can be set by giving appropriate values to a parameter by the user at run-time. We briefly explain the policies below with some examples.

BUNCH: The HCAs are assigned in a block manner. e.g. For 4 rails and 16 processes on one node the mapping will be of the form 0:0:0:0:1:1:1:1:2:2:2:2:3:3:3:3 where Rail 0 is assigned to the process with local-ranks 0, 1, 2 and 3; Rail 1 is assigned
to processes with local-ranks 4, 5, 6 and 7; and so on. SCATTER: Here the HCAs are assigned in a cyclic manner. e.g. For 4 rails and 16 processes on one node the mapping will be 0:1:2:3:0:1:2:3:0:1:2:3 where Rail 0 is assigned to the process with local-ranks 0, 4, 8 and 12; Rail 1 is assigned to processes with local-ranks 1, 5, 9 and 13; and so on. CUSTOM: The third available option is a custom list in which the rails can be passed as a string to the library separated by colons(;) as shown in the BUNCH and SCATTER policies described above.

3.3 Evaluation and Experiments

3.3.1 Experimental setup

The experimental setup is a cluster of 64 cores (16 cores per node) which are all which are all Quad-Core AMD Opteron Processor 8350 with a 2000.082 MHz CPU with 512MB cache size and is connected to an InfiniBand Switch. Each machine has 2 independent InfiniBand Mellanox mlx4 HCAs connected to it thus making it a pure multi-rail design. Hence we do not need to use virtual channels to simulate a multi-rail scenario. We use a few MPI benchmarks for evaluation. We first show the uni-directional bandwidth benchmark osu_bw and the latency osu_latency to show the effects of the various policies on point-to-point operations. Then we move over to a multi-bandwidth benchmark osu_mbw_mr which gives good insight on the various policies that we talk about with regards to the bandwidth and the number of messages transferred per second.
3.3.2 Performance of Various Rail Scheduling policies

In Figure 3.3 and Figure 3.4 we can see that for simple point to point operations, the various rail sharing policies do not make any significant impact on the performance. All the policies seem to provide the same performance. This is the expected behavior because we have only a single process and with most of the default policies the process will always get mapped to the same rail.

![Figure 3.3: osu_bw - Small Messages](image-url)
Figure 3.4: osu_bw - Large Messages
Similarly we show an evaluation for 2 processes using the *osu_latency* benchmark in Figure 3.5 and Figure 3.6. Even here we do not see any significant overheads with the different rail scheduling policies.

![Graph showing latency vs message size for different scheduling policies](image)

**Figure 3.5: osu_latency - Small Messages**

However in our next experiment we show a performance evaluation with 64 processes using the *osu_mbw_mr* benchmark. In this benchmark all the processes are actively getting assigned rails by the scheduler and hence we see varying performances for different message sizes.
Figure 3.6: osu_latency - Large Messages
Figure 3.7 shows bandwidth numbers for small messages. We can see that the new rail-binding policy does significantly better than the ROUND_ROBIN or the USE_FIRST policies. Since USE_FIRST uses only a single rail a very low performance is expected.

![Graph showing bandwidth for small messages](image)

Figure 3.7: osu_mbw_mr - Bandwidth for Small Messages

Figure 3.8 shows similar numbers for large messages over 64 processes. We see that all the policies come together toward a common point. This happens so because the ROUND_ROBIN and USE_FIRST policies start striping after they have reached the large message striping threshold. However, the Binding policy still gives a similar
performance because all the processes are sending a very high amount of data on all the rails and have an equal distribution of load.

The next two figures, Figure 3.9 and Figure 3.10, provide an insight on the number of messages sent per second. For smaller messages we see that the Rail Binding policies provide a much better performance compared to ROUND ROBIN and USE_FIRST. Further for large messages we see that all the policies show a similar performance.
Figure 3.9: osu_mbw_mr - Message Rate for Small Messages
Figure 3.10: osu mbw_mr - Message Rate for Large Messages
Chapter 4: DESIGN FOR IMPLEMENTING FAILOVER TECHNIQUE

As we mentioned earlier, although InfiniBand networks are designed to deliver high performance in a closely coupled environment, the entire system is subject to many failures. These could be at the software level, node level or network level. In our design we concentrate on the faults that occur at the network level. For our design purposes we define a link between two nodes as an end-to-end connection between two nodes. This includes the HCAs of both the communicating nodes, the cables connecting them and the intermediate switches to which these nodes are connected depending on the topology. The number of switches that fall between the link’s end-points (HCAs) can vary based on the topology. For example in a fat tree topology, the communicating nodes could be connected to the same switch or they could be very far apart from each other, each being on the left and right sub trees of the root switch.

Also each HCA may have multiple ports and any of these may be connected to different components in the network. Hence our level of granularity in terms of a link is assumed to be an individual switch port which is connected via network components to another port on an HCA of the other node. Also, as we have mentioned earlier
in Chapter 3, a rail can refer to an HCA, a port on an HCA or one of may virtual sub-channels on a single HCA port.

The chapter is organized as follows: In 4.1, we first explain the failure types and the associated errors. In 4.2, we then talk about how these failures are detected and how we react to them. We also explain how the design interacts with the different rail selection policies for different message sizes. In 4.3, we speak about recovery from the failures and explain our recovery protocol in detail. In 4.4, we show performance evaluation of our design on a multi-rail cluster. Section 4.5 presents a summary of our design and findings.

4.1 Network Failures

In this section, we describe the various network failures that we need to deal with in our design implementation.

Even in a tightly coupled InfiniBand network we see many failures occurring at the network level. This includes, but is not restricted to, HCA failures, cable faults, switch port failures and switch failures. For our design purposes we broadly classify failures occurring between two communicating nodes in two categories.

1. Local failures

2. Remote failures

Local failures are those which can be detected by the hardware and software on one of the nodes as opposed to Remote Failures which cannot be detected by the hardware but only via software on one or both the nodes. Failures such as an HCA port becoming inactive, cable fault on the connection between the HCA and the switch
at the first hop or a failure occurring at the switch/switch-port directly connected to an HCA are termed as local failures where as all the other failures occurring past the first switch are termed as remote failures. These could also be software failures. This classification is more important in terms of recovery because the host will get notified of such a recovery event in case of a local failure whereas in case of recovery in a remote failure the node needs to be explicitly made aware of it to resume any communication activity.

Generally, with no implementation for any recovery or error handling, whenever any failure on a rail is encountered, the entire operation cannot proceed further and the job aborts. One of the reasons for this is: the messages that were lost in the failure are never recovered. Hence even if the scheduler does not use the failed rail for new messages, the operation still will be incomplete. Also the at the scheduler level there is no means for checking the state of the rails at the hardware level. Software failures may be identified and taken care of but if any network component fails the higher layers are unaware of it. The message may at best be retransmitted a couple of times to ensure there are no transient issues.

4.1.1 Failure Notifications

In case of local failures as mentioned above, the associated port of the corresponding HCA moves to the DOWN state and stays in that state until all of the failed components become alive again. In remote failures however, even though the software may be made aware of such a failure occurring on the remote side, the HCA port remains in the ACTIVE state. In the failover design, we can decide on the type
of the failure at every node based on state of the HCA, of which a rail is a partial or a complete abstraction.

Each communicating node is associated with a send queue, receive queue and a completion queue. A completion queue (CQ) is dedicated to a particular (rail) (may be) shared by all processes using that rail for communication. As a send or a receive completes the corresponding completion entry is posted on the CQ and there is a status flag associated with it. These completion queue entries are used to notify the software of a completion event and take necessary action such as freeing the memory associated with the send-receive operation. InfiniBand provides a primitive ibv_poll_cq() for this purpose. Once a CQ entry is polled from the CQ it is permanently lost and cannot be recovered again. In case of successful completions, the CQEs are returned with an IBV_WC_SUCCESS code. In case of a failure the status is an error code which specifies the type of the failure.

4.2 Failure Detection

Whenever a failure occurs at the sender’s side, the completion entry returns with a retry-exceeded error (IBV_WC_RETRY_EXC_ERR) after having waited for a timeout. This time is specified in the QP attributes and is calculated as 4.096*(2 power (qp_attr.timeout)) * (qp_attr.retry_cnt) usec. Thus this value can be tuned by the user while setting up connections. After the first ‘retry-exceeded’ error is received, all the other completion entries associated with this QP return with an IBV_WC_WR_FLUSH_ERR.

During a normal poll CQ operation if we get an IBV_WC_RETRY_EXC_ERR we initiate the failover procedure. We immediately check for the status of the port of
the HCA on which we received the failure event. Here we can determine whether it was a remote failure or a local failure. If this error was received in a send operation we primarily need to recover the lost data and re-send it to the receiver. In case of a receive operation the sender side would also have received the same error and the sender will appropriately resend the lost data. In a very unlikely scenario that neither of the sides are able to detect a lost data packet, it will be taken care of by the Reliable Connection (RC) semantics of the layer above.

After determining the failure type we take the following actions at the sender’s side, depending on the failure type.

4.2.1 Remote Failures

For remote failures, the following steps are needed to be taken

- Recover all the lost Work Queue Entries (WQEs) that arrived in failure on that port.
- Recover other information such as the receiver’s rank, the port number, rail number, etc.
- Mark this particular failed rail INACTIVE, so that no new send operations use this rail.
- Also mark the connection as PARTIALLY-ACTIVE since it can still use the other active rails for sending data.
- Resend all the recovered WQEs to the same receiver using a different working rail.
• For any future send operations ensure that we do not choose the failed rail and use only those rails that are in the ACTIVE state.

4.2.2 Local Failures

In case of local failures, similar to remote failures, the following steps are needed to be taken

• Recover all the lost Work Queue Entries (WQEs) that arrived in failure on that port.

• Recover other information such as the receiver’s rank, the port number, rail number, etc.

• Mark this particular failed rail INACTIVE, so that no new send operations use this rail.

• Also mark the connection as PARTIALLY-ACTIVE since it can still use the other active rails for sending data.

• Resend all the recovered WQEs to the same receiver using a different working rail.

• For any future send operations ensure that we do not choose the failed rail and use only those rails that are in the ACTIVE state.

In addition to all the steps mentioned above, there is one more step that we perform which can improve the performance and provide a much more efficient solution.

• Since the CQ is shared by all the process using a particular rail, we can recover all the entries that were received in error on all the processes by repeatedly
polling the CQ for completion events and resend all the messages to their respective destinations. We can do this because it is ensured that if we have a local failure then all the messages that were sent out on that particular rail would have returned in error.

4.2.3 New rail selection and policies

After a failure is encountered the scheduler needs to be notified of such a failure. If this is not done the scheduler will continue to select the failed rail. At this point, i.e. immediately after a failure has occurred and the recovered messages are sent out, the scheduler is not aware of it. To aid the scheduler in dynamically deciding which rails to choose and which not to, we associate each rail with a state. Healthy rails are in the ACTIVE state where as in case of a hardware failure, the rail will be moved to the INACTIVE state. The scheduler can always check for the state of a rail before it assigns it to a particular process. Generating only trivial active and inactive states is not enough when it comes to recovery purposes. A rail may have recovered from a failure but it may not be safe to send on it as the connection may have been broken. In Figure 4.1 we show the different states that the rail may be in, throughout an MPI operation.

With regards to rail selection policies we also make sure that for a new send operation, the newly selected rail conforms to the rail selection policy that is being employed by the user. For example in a round-robin policy the next best rail will be selected in a round robin manner where as in a rail-binding policy a particular process will be bound to the next best available rail. Also in striping we make sure
that we stripe the data across all the rails except for the one that is known to have failed.

4.3 Recovery from failures

In this section we describe our recovery protocol in detail.

4.3.1 Notification of Recovery

When a component that had failed recovers, it generates an asynchronous event and lets the library know of its active state. This happens only on the node which had reported a local failure. The library on the local node constantly runs a thread
which gets notified of such interrupts and can take appropriate actions. The peer
which had recorded a remote failure at this point does not know of any such recovery.

The local node cannot directly start sending on this recovered rail, as the connec-
tions on that rail may have broken or the earlier QP might be corrupted. Also the
remote node will not initiate a send on that particular rail since it is not yet aware
that the other side has recovered.

4.3.2 Corrective actions

For resuming normal operations on the newly recovered rail, we have to ensure
that it is safe to send and receive on that channel before initiating any send-receive
operations from either side. Since only the local side is aware of such a recovery it
needs to initiate a handshake of messages with the remote side. Also we have to
ensure that we newly establish all the connections on that channel and recreate all
the necessary data structures. At the same time we also have to make sure that the
other send-receive operations along all the other channels are not affected in any way.
It is desirable to continue message transfers along all the other channels to maximize
the performance. The most important container to establish new connections and
keep track of existing ones is the VIRTUAL_CHANNEL (VC). Since the VC contains
all the data pertaining to the rails it is also important that the VC is also aware at
all times of the failures and recoveries. Thus, similar to the rails, the VC also needs
to be associated with special state for keeping track of failover activities.

Figure 4.2 describes the different VC states and their transitions based on various
events.

The VC states are referred to in the recovery protocol below.
For message recovery we have developed a connection management protocol which only reestablishes connections on the failed channel using UD messages and is described below.

4.3.3 Recovery Protocol

As we mentioned above, once a given communication pair is broken, the RC connection between them cannot be guaranteed to be active and we cannot trivially use the same connection to send the data even though the rail has recovered. All the connections need to be newly established and all the queue pairs need to be newly created and it also needs to be ensured that both the sides are in a ready-to-send state. Since the reliable connection is assumed to be broken and we cannot guarantee
a response from the remote side, we channelize the connection re-establishment via a series of UD exchange messages.

Figure 4.3 below explains the role of the client and server in the message exchange and shows the exchange of messages and the actions taken on both the sides.

Figure 4.3: Failover recovery timeline

The recovery protocol performs the following sequence of message transfers:

- When a local node is notified of a recovery, it searches amongst all of its processes, and isolates those processes which were using that rail to communicate with others. One process may have communicated with multiple processes using that rail and hence we need to establish connection with all of the processes. Thus it is not enough to only know of processes using that rail but also we need to know of every other peer that this process communicated with.
• We move this communication channel to a NEED-FOR-REPAIR state and also mark the rail as PARTIALLY-ACTIVE for that channel. We cannot at this point directly mark the rail as ACTIVE even on the local node because that process is actively selecting other working rails continuously sending out data on them. If we mark our newly recovered rail as active it might get chosen for the next data transfer but fail because we have not re-initiated and new connection on that rail.

• The local process in the NEED-FOR-REPAIR state now sends out a UD FAILOVER-RECONNECT-REQUEST message to each of its communicating peers. This local node now takes the role of a client and expects the remote side to assume the role of a server. If the remote side is unable to receive these messages the client will try to resend them on the UD channel periodically till it receives a reconnect reply message or time-out eventually. It will create new QPs at it side but they will still be in the inactive state.

• When the remote side receives such a reconnect request on its UD channel it assumes the role of a server. The newly formed server now moves its communication channel to the NEED-FOR-REPAIR state and also marks the rail as PARTIALLY-ACTIVE for that channel. Then it responds with a FAILOVER-RECONNECT-REPLY message to the client. At this point the server knows that the other side has created QPs and it is safe to create new ones. It can also move the QP to the READY-TO-RECEIVE (RTR) state.

• The client on getting this reply message knows that the server is also ready to send and receive messages. At this point we can also move the connection as
well as the rail to the ACTIVE state since we know that the server is in the RTR state. The client now moves to the READY-TO-SEND state and sends an acknowledgment to the server.

- The server now receives the ACK message and moves to the RTS state. It can now move its connection to the ACTIVE state and also marks the partially active state as active. Then it sends one more ACK to the client to notify that it is ready to send data.

- Since both the peers have their rails in the active state, during normal operation, the process can now select this rail since the connection is in the active state and start sending and receiving data.

4.4 Experimental Results

In this section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results.

4.4.1 Experimental Setup

The experimental setup is a cluster of 64 cores (16 cores per node) which are all Quad-Core AMD Opteron Processor 8350 with a 2 GHz CPU with 512MB cache size and is connected to an InfiniBand Switch which has an internal topology of a Fat Tree. Each machine has two independent Mellanox InfiniBand ConnectX HCAs making it a pure multi-rail configuration. Thus we do not need to use and virtual channels to simulate a multi-rail scenario. We use a set of MPI benchmarks for evaluation. We first show a typical failover case using a multiple-bandwidth benchmark osu_mbw_mr running over 64 processes. We then use the uni-directional bandwidth benchmark
osu\_bw and the osu\_mbw\_mr to show the performance evaluation of our design. Then we move over to collective operations. We are making use of the mpiBench benchmark which runs collective operations over a number of iterations. We show results with the Bcast, Reduce and Alltoall Collective algorithms.

4.4.2 Point to point operations

To test the immediate scalability and functionality we subjected the failover design to a 64 process multi-bandwidth benchmark. The set-up works in the following manner. We initiate a benchmark on our cluster and let it run without failure for a few message sizes. This is so that we catch errors in between operations which is the most likely scenario in a large cluster setup. Then we introduce a few errors in the network by bringing down a few ports from the switch. The switch port is a good choice because it allows us to visualize an actual run time scenario where the failure can occur at any component and not just at the HCA or the node level. Then after some time has elapsed, we bring the ports back up randomly. A thing to note here is that when one connection on an HCA goes down, 16 process (in our 64 process scenario) will be directly affected with the ROUNDROBIN scheduling policy for large message sizes.

In the Figure 4.4 below, we can see that the failure occurred after message size 1024. The performance at message size 2048 drops drastically due to the default retry time out as we had mentioned above. However we instantly notice that the performance then moves upto single-rail performance and continues throughout the duration that the rail is marked as down.
Figure 4.4: osu\_mbw\_mr at 64 processes with Failover
At message size 16K the rail comes up and as explained in the design section the failed processes call upon failover routine and re-establish connections. Now that all the rails are active we reach dual rail performance. Again at message size, 128K another rail goes down and we see a drastic drop in performance for the same reasons as before. Also the rail immediately recovers and reaches dual-rail performance for the next message size. We also show the number of messages transferred for each message size and we see a similar trend.

For the next set of results, we modified the osu_bandwidth benchmark to run for a constant message size. The benchmark is run over a period and we capture the bandwidth at regular intervals of time. In Figure 4.5 we observe that as after the rail failure occurs at Time Step 3 the performance does not go below the single-rail mark and again reaches dual rail during time steps 6 and 7. Similarly at the next failure we see a very predictable performance where the rail hits single rail performance and subsequently dual rail.

Similarly we show the performance of osu_mbw_mr for different message sizes. Figure 4.6 shows the uniform bandwidth experiment from above for a message size of 64K. The number of messages transferred per second is shown in Figure 4.7. In Figure 4.8 for the bandwidth, we observe slight variation from the trend that we have established above, and similarly in Figure 4.9 for the message rate. We observe that the performance does not drop below the single rail mark but instead immediately switches to the single rail bandwidth performance. This was an experiment where the rail failure was introduced before the start of the bandwidth calculation. This has been done to prove that the degradation in performance in the previous cases is a result of the stoppage time to confirm a failure.
Figure 4.5: osu_bw at message size 4MB with Failover
Figure 4.6: osu_mbw_mr at message size 64K with Failover
Figure 4.7: osu_mbw_mr Messages/s at size 64K with Failover
Figure 4.8: osu_mbw_mr at message size 1M with Failover
Figure 4.9: osu_mbw_mr Messages/s at size 1M with Failover
4.4.3 Collective Operations

Collective algorithms perform intense data transfer operations and are based on algorithms that are used to improve their performance. We have selected the mpiBench suite for our purposes and we show the performance figures with a few collectives viz. Bcast, Reduce and Alltoall.

We chose Bcast and Reduce since they perform somewhat complementary functions. In Bcast a single process (root) would send messages to all the other processes in the job. On the contrary, in Reduce, messages from many processes are cumulatively gathered on one root node. Hence it was made sure that for all the experiments, the root process always failed amongst other failures.

Figure 4.10 and Figure 4.11 show the behavior of the system with the failover mechanism. Here again we fixed the sizes of the system to 128K and 256K for Bcast and Reduce, respectively. From the graphs we observe that, similar to the point to point experiments the time required for a time step gets higher and reaches the single rail performance. On recovery it resumes the dual rail operation and thus we get higher performance.

Finally we show results with the All-to-all Collective operation. This algorithm consumes a lot of bandwidth and thus shows a significant improvement over single rail performance. In Figure 4.12 we can notice the difference between single and dual rails as compared to the Reduce and Bcast operations we have shown above.

4.4.4 Application benchmarks

As the final step in our performance evaluation we show performance numbers with the NAS suite of benchmarks. We use the Class B and Class C BT and LU
Figure 4.10: mpiBench - Bcast at message size 128K with Failover
Figure 4.11: mpiBench - Reduce at size 256K with Failover
Figure 4.12: mpiBench - All-to-all at size 8K with Failover
benchmarks for different systems sizes. We have observed that dual rail performance
does not provide any special benefits here, hence we show only the dual rail numbers.
However we also see that failover takes longer times to complete the working. As we
have mentioned earlier, the timeout associated with the \textit{ibv\_poll\_cq} function plays the
major role in the higher times for failover figures. However we can be assured that
the job goes to completion without aborting in between which would have been the
case without failover.

Table 4.1 and Table 4.2 show our performance comparisons for different message
sizes

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Job Size} & \textbf{Class B} & \textbf{Class C} \\
\hline
16 & 172.11 & 198.09 & 681.14 & 701.2 \\
36 & 76.02 & 109.74 & 309.59 & 339.54 \\
64 & 44.46 & 78.25 & 179.23 & 197.53 \\
\hline
\end{tabular}
\caption{Evaluation with NAS BT benchmark}
\end{table}

\begin{table}
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\textbf{Job Size} & \textbf{Class B} & \textbf{Class C} \\
\hline
16 & 164.48 & 179.61 & 728.78 & 736.09 \\
36 & 77.83 & 120.67 & 316.08 & 336.91 \\
64 & 47.53 & 68.84 & 179.54 & 187.95 \\
\hline
\end{tabular}
\caption{Evaluation with NAS LU benchmark}
\end{table}
Chapter 5: CONCLUSION AND FUTURE WORK

5.1 Summary of contributions

The research in this dissertation aims towards providing high performance and network fault tolerant MPI over InfiniBand. These designs are likely to benefit the existing and upcoming ultra-scale InfiniBand clusters by providing a very efficient fault tolerance solution. Following sections provide a more detailed summary of the research contributions:

5.1.1 Enhanced multi-rail scheduling policies

In Chapter 3, we presented an enhanced design for rail scheduling policies by introducing a rail-binding scheme which restricts the scope of access to the rails by processes. We saw in the performance evaluations that this design benefits certain benchmarks for some message sizes. We believe that, if the behavior of an application is well known before hand then the rail-binding policies can certainly improve the performance. Also there is an opportunity to better manage network hardware constraints by evenly distributing the traffic over different rails.
5.1.2 Failover with multi-rail

In Chapter 4, we discussed the various network failures that can occur in high performance networks and provided a design to tolerate and handle such failures meanwhile continuing to operate over the other available and healthy rails till the failed components are recovered. For recovery we proposed a new protocol which uses a handshake mechanism to exchange messages thereby helping to re-establish broken connections and help resume normal operation to provide the maximum available bandwidth.

Also, we showed that in case the failed components are not able to recover from the failures we can still continue to operate with the existing hardware and take the job to completion.

We presented performance numbers which depicted the failure and recovery of rails and established that the performance does not suffer in any way and provides almost equal bandwidth as much as the other healthy rails would have provided in isolation.

5.2 Future Work

We plan to take this work further to work with other network failures which the local nodes cannot be notified of. We briefly take a look at other related areas of research that our work can be improvised on.

1. Dynamic rail scheduling policies: It would be desirable if the communication scheduler can dynamically detect network congestions or traffic and select rails which are providing a better performance.
2. Failover for congestion control: A similar approach with failover can be used even to remove congestion in networks. If our multi-rail configuration spans over multiple subnets and we are notified of congestions, then it could be possible to move over to the other subnet and avoid congested routes.

3. Failure Notifications to other processes and end user: Mechanisms could be provided to notify the end user of the failures in networks thereby resulting in faster recovery times and higher performance. Also in collective operations, the failover messages can be propagated to other rails and the system as a whole can be made aware of the existence of failure. This could lead to lesser resending of data over the network and higher throughput.
Bibliography


57


