HIGH PERFORMANCE AND NETWORK FAULT TOLERANT MPI WITH MULTI-PATHING OVER INFINIBAND

DISSERTATION

Presented in Partial Fulfillment of the Requirements for

the Degree Doctor of Philosophy in the

Graduate School of The Ohio State University

By

Abhinav Vishnu, B. Tech, M. S.

* * * * *

The Ohio State University

2007

Dissertation Committee:

Prof. D. K. Panda, Adviser

Prof. P. Sadayappan

Prof. S. Parthasarathy

Approved by

____

Adviser

Graduate Program in Computer Science and Engineering

© Copyright by

Abhinav Vishnu

2007

ABSTRACT

In the last decade or so, the high performance community is observing a paradigm shift with interconnection methodology for processing elements. Combining commercial off-the-shelf components to build supercomputers has provided users with an excellent price-to-performance ratio. At the same time, scientific applications ranging from molecular dynamics to ocean modeling are being designed with Message Passing Interface (MPI) being the *de facto* programming model. The insatiable computational requirements of the scientific applications has been continuously pushing the scale of these clusters. Increasing scale of these clusters has aggravated the occurrence of hot-spots in the network and reduced the mean time between failures of difference network components. In order to provide the best performance to the scientific applications, it is imperative that the MPI libraries are capable of avoiding network hot-spots and resilience to faults in the network.

At the same time, InfiniBand has emerged as a popular interconnect, providing a plethora of modern features with open standard and high performance. In this dissertation, we focus on designing a *communications and network fault tolerance layer* with InfiniBand, which leverages the presence of multiple paths in the network for avoidance of hot-spots in the network and network fault tolerance. Much of the dissertation has been integrated with an open source effort, MVAPICH, which is

a popular implementation of MPI over InfiniBand and is used by a large number of supercomputers in the world.

Dedicated to my wife, my parents and my brother

ACKNOWLEDGMENTS

I would like to thank my adviser, Prof. D. K. Panda for guiding me throughout the duration of my PhD study. I'm thankful for all the efforts he took for my dissertation. I would like to thank him for his friendship and counsel during the past years.

I would like to thank my committee members Prof. P. Sadayappan and Dr. S. Parthasarathy for their valuable guidance and suggestions.

I'm grateful for financial support by National Science Foundation (NSF), Department of Energy (DOE) and IBM for PhD Fellowship.

I'm thankful to Dr. Rama Govindaraju, Dr. Rajeev Sivaram, Dr. Hanhong Xue, Brad Benton and Chulho Kim for their support and guidance during my summer internships.

I would like to thank all my senior Nowlab members for their patience and guidance, Dr. Pavan Balaji, Dr. Jiuxing Liu, Dr. Jiesheng Wu, and Dr. Weikuan Yu. I would also like to thank all my colleagues Amith Mamidala, Sundeep Narravula, Karthik Vaidyanathan, Sayantan Sur, Gopal Santhanaraman, Savitha Krishnamoorthy, Weihang Jiang, Wei Huang, Qi Gao, Matt Koop, Lei Chai and Ranjit Noronha. I'm especially grateful to Amith and Sundeep and I'm lucky to have collaborated closely with them.

Finally, I would like to thank my family, Khushbu (my wife), Poonam (my mother), Vishnu (my father) and Abhishek (my brother). I would not have had made it this far without their love and support.

VITA

May 20, 1980	Born - Moradabad, India.
August 1998 - May 2002	B.Tech, Computer Science and Engineering, Institute of Technology, Banaras Hindu University, Varanasi, India.
August 2002 - December 2003	Graduate Teaching Associate, The Ohio State University.
January 2004 - June 2005	Graduate Research Associate, The Ohio State University.
June 2005 - September 2005	Summer Intern, IBM, Austin, TX.
September 2005 - June 2006	Graduate Research Associate, The Ohio State University.
June 2006 - September 2006	Summer Intern, IBM, Poughkeepsie, NY.
September 2006 - June 2007	IBM PhD Fellow, The Ohio State University.
June - December 2007	Graduate Research Associate, The Ohio State University.

PUBLICATIONS

S. Narravula, A. Mamidala, A. Vishnu and D.K. Panda, "High Performance MPI over iWARP: Early Experiences", International Conference on Parallel Processing (ICPP'07), September 2007, Xian, China.

A. Vishnu, M. Koop, A. Moody, A. Mamidala, S. Narravula and D.K. Panda, "Hot-Spot Avoidance with Multi-Pathing Over InfiniBand: An MPI Perspective",

International Conference on Cluster Computing and Grid (CCGrid'07), May 2007, Brazil.

S. Narravula, A. Mamidala, A. Vishnu and D.K. Panda, "High Performance Distributed Lock Management Services using Network-based Remote Atomic Operations ", International Conference on Cluster Computing and Grid (CCGrid'07), May 2007, Brazil.

A. Vishnu, B. Benton and D.K. Panda, "High Performance MPI on IBM 12x InfiniBand Architecture", International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS '07 (HIPS'07), March 2007 , LongBeach, CA.

A. Vishnu, A. Mamidala, S. Narravula and D.K. Panda, "Automatic Path Migration over InfiniBand: Early Experiences", Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS '07 (SMTPS'07), March 2007, LongBeach, CA.

A. Mamidala, S. Narravula, A. Vishnu and D.K. Panda, "Connection-Less Transport for Performance and Scalability of Collective and One-sided Operations: Trade-offs and Impact" , International Conference on Principles and Practices of Parallel Programming, March 2007, San Jose, CA.

A. Vishnu, P. Gupta, A. Mamidala and D.K. Panda, "A Software Based Approach for Providing Network Fault Tolerance in Clusters with uDAPL interface: MPI Level Design and Performance Evaluation", SuperComputing (SC'06), November 2006, Tampa, FL.

A. Mamidala, A. Vishnu and D.K. Panda, "Efficient Shared Memory and RDMA based design for MPI Allgather over InfiniBand", EuroPVM/MPI 2006, September 2006, Germany.

M. Koop, W. Huang, A. Vishnu and D.K. Panda, "Memory Scalability Evaluation of the Next Generation Intel Bensley Platform for InfiniBand", HotInterconnects (HotI) 2006, August 2006, Stanford, CA.

A. Vishnu, G. Santhanaraman, W. Huang, H- W. Jin and D.K. Panda, "Supporting MPI-2 One Sided Communication on Multi-Rail InfiniBand Clusters: Design Challenges and Performance Benefits", International Conference on High Performance Computing 2005, December 2005, Goa, India.

S. Sur, A. Vishnu, W. Huang and D.K. Panda, "Can Memory-Less Network Adapters Benefit Next-Generation InfiniBand Systems? ", HotInterconnects 2005 (HotI'05), August 2005, Stanford, CA.

A. Vishnu, A. Mamidala, H- W. Jin and D.K. Panda, "Performance Modeling of Subnet Management on Fat Tree InfiniBand networks using OpenSM", International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS '05 (SMTPS'05), April 2005, Denver, CO.

J. Liu, A. Mamidala, A. Vishnu and D.K. Panda, "Evaluation InfiniBand Performance with PCI-Express", IEEE Micro, February 2005.

J. Liu, A. Vishnu and D.K. Panda, "Building Multi-Rail InfiniBand Clusters: MPI-level Design and Performance Evaluation", SuperComputing 2004 (SC'04), November 2004, Pittsburgh, PA.

J. Liu, A. Mamidala, A. Vishnu and D.K. Panda, "Performance Evaluation of InfiniBand with PCI-Express", HotInterconnects 2004 (HotI'04), August 2004, Stanford, CA.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

Computer Architecture	Prof. D. K. Panda
Computer Networks	Prof. D. Xuan
Software Systems	Prof. P. Sadayappan

TABLE OF CONTENTS

Page

Abstract ii

Dedication iv

Acknowledgments v

Vita vii

List of Figures xv

Chapters:

1. Introduction 1

2. Background 6

    2.1 Introduction to Message Passing Interface 6
        2.1.1 MPI Point-to-Point Communication 7
        2.1.2 MPI One-Sided Communication 9
    2.2 Overview of InfiniBand Architecture (IBA) 10
        2.2.1 Transport Services 11
        2.2.2 Management Infrastructure 12
        2.2.3 Overview of Communication State Transition 13
        2.2.4 Overview of Automatic Path Migration 15
        2.2.5 Quality-of-Service Support 17
        2.2.6 Hardware Support for Congestion Notification 17
    2.3 Overview of InfiniBand Adapters and Access Layers 17
        2.3.1 Overview of InfiniBand Adapters 18
        2.3.2 Overview of Access Layers 20

3. Problem Statement and Research Approaches 22

3.1 Dissertation Overview 30

4. Efficient MPI-1 Design for Multi-Rail InfiniBand Clusters 34

4.1 Basic MPI Design for Multi-Rail Networks 35
4.1.1 Virtual Subchannel Abstraction 36
4.1.2 Scheduling Policies 38
4.2 Implementation Details 39
4.2.1 Handling Multiple Adapters 39
4.2.2 Out-of-Order Message Processing 40
4.2.3 Multiple RDMA Completion Notifications 41
4.3 Performance Benefits with Multi-Rail Design on MPI-1 Benchmarks 42
4.3.1 Experimental Testbed 42
4.3.2 Performance Evaluation with Point-to-Point and Collective Communication 45
4.3.3 Performance Evaluation with MPI Applications 46
4.4 Summary 49

5. Supporting One-Sided Communication With Multi-Rail InfiniBand Clusters 50

5.1 Communications Layer for One-sided Communication with Multi-Rail Clusters 51
5.2 Detailed Design Issues 52
5.2.1 Multiple Synchronization Messages 52
5.2.2 Scheduling of RDMA Read and RDMA Write Operations 53
5.2.3 Scheduling Policies Classification based on Message Size . 54
5.2.4 Absence of Rendezvous Protocol 55
5.2.5 Ordering Relaxation 55
5.3 Performance Benefits of Multi-Rail Design for One-Sided Communication 56
5.3.1 Experimental TestBed 56
5.3.2 Micro-benchmark Evaluation for MPI_Put Operation . . . 57
5.3.3 Micro-Benchmark Performance Evaluation for MPI Get Operation 60
5.3.4 MPI-2 One-Sided Interleaving Test 61
5.4 Summary 64

6. Improving Performance with IBM 12x InfiniBand Architecture 65

6.1 Overall MPI Design for 12x InfiniBand Architecture 66
6.2 Discussion of Scheduling Policies for different Communication Patterns 67
6.2.1 Point-to-Point Communication 67
6.2.2 Collective Communication 68
6.2.3 Communication Marker 69
6.3 Performance Evaluation with MPI over IBM 12x InfiniBand Architecture 70
6.3.1 Experimental Testbed 70
6.3.2 Performance Evaluation with Micro-Benchmarks 71
6.3.3 Performance Evaluation with NAS Parallel Benchmarks . 75
6.4 Summary 76

7. Hot-Spot Avoidance With Multi-Pathing Using LMC Mechanism . . . 78

7.1 Limitations of Existing Designs 79
7.1.1 Leveraging Multiple Paths Using LMC 81
7.1.2 Adaptive Striping 82
7.1.3 Selecting Number of Paths 85
7.1.4 Scalability Aspects of HSAM 87
7.2 Performance Evaluation 88
7.2.1 Experimental Testbed: 88
7.2.2 Performance Benefits of HSAM with Collective Communication: 89
7.2.3 Performance Benefits at Application Level 92
7.3 Summary 93

8. Enhanced Design for Avoiding Hot-Spots with Better Network Paths Utilization 94

8.1 Adaptive Approaches for Hot-Spot Avoidance 95
8.1.1 Automatic Path Migration Based Hot-Spot Avoidance . . 95
8.1.2 Current Software Based Approaches and Limitations . . . 96
8.2 BSS Policy Based MPI Design 97
8.2.1 The BSS Scheduling Policy 97
8.2.2 BSS Policy Based Communications Layer 99
8.3 Detailed Design Issues 100
8.3.1 Selecting Number of Paths: 100
8.3.2 Scalability Issues: 101

8.3.3 Integration with MPI Rendezvous Protocol: 101
8.3.4 Discussion 102
8.4 Performance Benefits with the BSS Policy Based MPI Design . . 103
8.4.1 Experimental TestBed 104
8.4.2 Performance Evaluation with Collective Communication . 104
8.4.3 Performance Evaluation with MPI Applications 109
8.5 Summary 111

9. Network Fault Tolerance Using Automatic Path Migration 113

9.1 Overall Design 114
9.2 Design of Network Fault Tolerance Modules 115
9.2.1 Alternate Path Specification Module 116
9.2.2 Path Loading Request Module 117
9.2.3 Path Migration Module 118
9.3 Interaction of Main Execution Thread and the Asynchronous Thread With Network Fault Tolerance Modules 118
9.3.1 Integration of Network Fault Tolerance Modules at Verbs and MPI Layer 119
9.4 Performance Evaluation with the APM Based Network Fault Tolerance Modules 120
9.4.1 Experimental Testbed 120
9.4.2 Evaluation of the Network Fault Tolerance Modules at the Verbs Layer 121
9.4.3 Impact of PSM and LRM on QP Transitions 123
9.4.4 Impact of Network Fault Tolerance Modules on Latency . 123
9.4.5 Impact of Network Fault Tolerance Modules on Computation 125
9.4.6 Evaluation of the Network Fault Tolerance Modules at the MPI Layer 127
9.5 Summary 130

10. Software Based Network Fault Tolerance on Clusters with uDAPL Interface 131

10.1 Overall Design for uDAPL Based Network Fault Tolerant MPI . . 132
10.2 Basic Infrastructure For Network Fault Tolerance Design 133
10.2.1 Multi-Network Abstraction Layer 135
10.2.2 Implementing Abstraction Layer over uDAPL 135
10.2.3 Communication Methodology for Multiple Interconnects . 136
10.3 Design of Communications and Network Fault Tolerance Layer . 137
10.3.1 Completion Filter and Error Detection Module 137
10.3.2 Message (Re)Transmission Module 139

10.3.3 Path Recovery and Network Partition Handling Module . 140
10.4 Performance Evaluation with uDAPL Based MPI 141
10.4.1 Experimental Testbed 142
10.4.2 Performance Evaluation on Configuration A 143
10.4.3 Performance Evaluation on Configuration B 148
10.4.4 Performance Evaluation with Network Faults 148
10.5 Summary 152

11. Open Source Software Release and Adoption 155

12. Conclusions and Future Research Directions 157

12.1 Summary of Research Contributions 157
12.1.1 Efficient MPI-1 Design for Multi-Rail InfiniBand Clusters 158
12.1.2 Supporting MPI One-Sided Communication with Multi-Rail InfiniBand Clusters 158
12.1.3 Improving Performance with IBM 12x InfiniBand Architecture 159
12.1.4 Hot-Spot Avoidance with Multi-Pathing Using LMC Mechanism 159
12.1.5 Enhanced Design for Avoiding Hot-Spots with Better Network Paths Utilization 160
12.1.6 Network Fault Tolerance Using Automatic Path Migration 161
12.1.7 Software Based Network Fault Tolerance on Clusters with uDAPL Interface 162
12.2 Future Research Directions 163

Bibliography 165

LIST OF FIGURES

Figure **Page**

2.1 MPI Communication Semantics 7

2.2 MPI Protocols 8

2.3 InfiniBand Architecture [26] 11

2.4 QP Communication State Diagram 14

2.5 QP Path Migration State Diagram (Adaptation From InfiniBand Specification [26]) 16

2.6 IBM 12x InfiniBand HCA Block Diagram 19

2.7 Access Layers for InfiniBand and Other RDMA-enabled Interconnects 20

3.1 Overall Interaction of Communications and Network Fault Tolerance Layer 23

3.2 Proposed Mapping of Multi-Pathing MPI Design Components with InfiniBand Features 25

3.3 144-port InfiniBand Switch Block Diagram 27

3.4 Communication Steps in Displaced Ring Communication 27

3.5 Link Usage with Displaced Ring Communication 28

4.1 Basic Architecture 36

4.2 Virtual Subchannel Abstraction 37

4.3 MPI Latency (UP mode) 43
4.4 MPI Bandwidth (Small Messages, UP mode) 43
4.5 MPI Bandwidth (UP mode) 44
4.6 MPI Bidirectional Bandwidth (UP mode) 44
4.7 MPI Bandwidth (SMP mode) 44
4.8 MPI Bidirectional Bandwidth (SMP mode) 44
4.9 MPI_Bcast Latency (UP mode) 47
4.10 MPI_Alltoall Latency (UP mode) 47
4.11 MPI_Bcast Latency (SMP mode) 47
4.12 MPI_Alltoall Latency (SMP mode) 47
4.13 Application Results (8 processes, UP mode) 48
4.14 Application Results (16 processes, SMP mode) 48
5.1 Basic Architecture 51
5.2 Multiple Synchronization Messages 53
5.3 Scheduling Policies at different layers for one-sided communication . 54
5.4 MPI_Put Bandwidth on the IA32 Cluster 58
5.5 MPI_Put Bandwidth on the EM64T Cluster 58
5.6 MPI_Put Bidirectional Bandwidth on the IA32 Cluster 59
5.7 MPI_Put Bidirectional Bandwidth on the EM64T Cluster 59
5.8 MPI_Put Latency on the IA32 Cluster 59

5.9 MPI_Put Latency on the EM64T Cluster 59

5.10 MPI_Get Bandwidth on the IA32 Cluster 60

5.11 MPI_Get Bandwidth on the EM64T Cluster 60

5.12 MPI_Get Latency on the IA32 Cluster 61

5.13 MPI_Get Latency on the EM64T Cluster 61

5.14 Ordered issue of one-sided operations 62

5.15 Re-ordered issue of one-sided operations 62

5.16 Interleaved throughput on the IA32 Cluster 63

5.17 Interleaved throughput on the EM64T Cluster 63

6.1 Overall MPI Design for IBM 12x InfiniBand Architecture 66

6.2 MPI Latency For Small Messages 72

6.3 MPI Latency For Large Messages 72

6.4 Impact of Scheduling Policies on Small Message Uni-directional Bandwidth 73

6.5 Impact of Scheduling Policies on Small Message Bi-directional Bandwidth 73

6.6 Large Message Uni-directional Bandwidth 74

6.7 Large Message Bi-directional Bandwidth 74

6.8 Alltoall, Pallas Benchmark Suite, 2x4 Configuration 74

6.9 Integer Sort, NAS Parallel Benchmarks, Class A 75

6.10 Integer Sort, NAS Parallel Benchmarks, Class B 75

6.11 Fourier Transform, NAS Parallel Benchmarks, Class A 76

6.12 Fourier Transform, NAS Parallel Benchmarks, Class B 76

7.1 Feedback Loop in Adaptive Striping 83

7.2 MPI Latency 88

7.3 MPI Alltoall Personalized (48x1) 88

7.4 MPI AllReduce (48x1) 89

7.5 MPI Reduce Scatter (48x1) 89

7.6 Displaced Ring Communication, 24x1, HSAM, 4 Paths, adaptive . . 91

7.7 Displaced Ring Communication, 24x1, Original, 1 Path 91

7.8 NAS Parallel Benchmarks, FT, Class B 91

7.9 NAS Parallel Benchmarks, FT, Class C 91

7.10 Performance Evaluation with PSTSWM 92

8.1 APM Based Hot-Spot Avoidance Approach 95

8.2 BSS Based MPI Design 99

8.3 BSS Integration with Rendezvous Protocol 99

8.4 144-port Switch Block Diagram 100

8.5 MPI_AlltoAll (48x1), Sequential Mapping 105

8.6 MPI_AlltoAll (64x1), Sequential Mapping 105

8.7 MPI_AlltoAll (32x1), Default Mapping 106

8.8 MPI_AlltoAll (64x1), Default Mapping 106

8.9 MPI_Allgather(48x1), Sequential Mapping 107

8.10 MPI_Allgather (64x1), Sequential Mapping 107

8.11 MPI_Allgather (32x1), Default Mapping 108

8.12 MPI_Allgather (64x1), Default Mapping 108

8.13 MPI_AllReduce (48x1), Sequential Mapping 108

8.14 MPI_AllReduce (64x1), Sequential Mapping 108

8.15 MPI_AllReduce (32x1), Default Mapping 109

8.16 MPI_AllReduce (64x1), Default Mapping 109

8.17 NAS Parallel Benchmarks, FT, Class B 110

8.18 NAS Parallel Benchmarks, FT, Class C 110

8.19 Other NAS Parallel Benchmarks, Class C, 64 Processes 110

9.1 Overall Design of Network Fault Tolerance Modules and Interaction with User Applications 114

9.2 An Example of Primary and Alternate Path of Communication used by PSM 117

9.3 Interaction of Network Fault Tolerance Modules with Main Execution Thread and Asynchronous Thread 119

9.4 Transition From INIT-RTS, Small Number of QPs 121

9.5 Transition From INIT-RTS, Large Number of QPs 121

9.6 Timings for different transition states in APM, Small Number of QPs 122

9.7 Timings for different transition states in APM, Large Number of QPs 122

9.8 Impact on Latency for 128 Byte Message with Increasing Number of QPs 124

9.9 Impact on Latency for Small Messages using 512 QPs with Increasing Message Size 124

9.10 Impact on Computation, Migrated-Armed Requested During Computation 126

9.11 Impact on Computation, Armed-Migrated Requested During Computation 126

9.12 Performance Evaluation on IS, Class A, 4x2 Configuration 128

9.13 Performance Evaluation on IS, Class B, 4x2 Configuration 128

9.14 Performance Evaluation on FT, Class B, 4x2 Configuration 129

9.15 Performance Evaluation on LU, Class B, 4x2 Configuration 129

10.1 Overall Design of Network Fault Tolerant MPI with a node comprising of multiple networks 134

10.2 A Node with only IBA Network 134

10.3 A Node with only Amasso Network 134

10.4 Communications and Network Fault Tolerance Layer 138

10.5 Communication Protocol For Recovery from Network Partitions and Previously Failed Paths 138

10.7 Latency Overhead for uDAPL and VAPI based MPI 144

10.8 Large Message Latency 144

10.9 Uni-directional Bandwidth Comparison 145

10.10 Bi-directional Bandwidth Comparison 145

10.11 Performance Evaluation of IS and CG NAS Parallel Benchmarks, Class A 146

10.12 Performance Evaluation of FT and MG NAS Parallel Benchmarks, Class A 146

10.13 Uni-directional Bandwidth Comparison for Fault Tolerant Schemes, IBA Path Fails 147

10.14 Bidirectional Bandwidth Comparison for Fault Tolerant Schemes, IBA Path Fails 147

10.15 Uni-directional Bandwidth Comparison for Fault Tolerant Schemes, GigE Path Fails 149

10.16 Bidirectional Bandwidth Comparison for Fault Tolerant Schemes, GigE Path Fails 149

10.17 Performance Comparison on NAS Benchmarks, When the IBA Path Fails on First Message Transmission 150

10.18 Performance Comparison on NAS Benchmarks, When the IBA Path Fails on First Message Transmission 150

10.19 Uni-directional Bandwidth Comparison for Fault Tolerant Schemes with Network Partition, IBA Path Recovers First 153

10.20 Uni-directional Bandwidth Comparison for Fault Tolerant Schemes with Network Partition, GigE Path Recovers First 153

CHAPTER 1

INTRODUCTION

The computational needs of today's scientific applications has led to the augmentation of high performance computing. Applications varying from molecular dynamics [11], ocean modeling [4] to car crash simulations require precision in accuracy of predictions and minimal execution time. Combining the commercial off-the-shelf (COTS) processors to solve today's challenging applications has led to cluster computing [47], a very effective methodology for excellent cost-performance ratio. Networks which provide very low latency and excellent bandwidth have been emerging in the past decade [43, 48, 53]. At the same time, MPI [38, 39] has become the de-facto programming model to write the above applications. However, the traditional communication protocols such as TCP/IP limit the applications from realizing the peak potential of the clusters due to their high protocol overhead, heavy kernel involvement and extra data copies in the communication critical path [28]. Communication systems have been proposed in the last decade or so to overcome these limitations [60, 10, 46]. The motivation of these communication systems is to minimize the interaction with the operating system kernel in the path of critical execution, and reduction in data copies. The result is the realization

of higher communication performance to the application layer. Efforts to combine these communication systems have also been proposed recently [17, 15, 9].

More recently, the need for open standard and high performance led to the proposition of InfiniBand [26]. The InfiniBand architecture has been proposed as the next generation interconnect for I/O and inter-process communication. The current generation of InfiniBand products provide latency as low as 2us and bandwidth reaching beyond 30Gb/s. InfiniBand provides a plethora of features which can be used for designing high performance communication substrate. Features like Remote Direct Memory Access (RDMA) can be used for data transfer between end nodes with Kernel bypass. InfiniBand Automatic Path Migration provides automatic recovery from network failures and completion queue mechanism can be used for estimation of path bandwidth and reliable delivery of data to the remote node. Hardware multi-cast can be used for designing efficient collective communication primitives, and service level-virtual lane mechanism can be used for providing quality of service to MPI applications. Similarly, the LID Mask Count (LMC) mechanism can be used to provide multiple paths in the network. As a result, InfiniBand clusters are becoming increasingly popular, reflected by the TOP500 [6] supercomputer rankings.

However, increasing scale of these clusters has led to challenges beyond memory scalability. Hot-spots may occur at the I/O bus, with the increasing multi-way SMP systems (and the recent multi-core architectures). To make matters worse, Hot-spots in the network may become prevalent due to the sharing of network links by multiple communication instances, different jobs and various communication patterns. In the worst case, reduced mean time between failures (MTBF) of the

network component may break the existing path(s) of communication, due to the failure of cables, switches and adapters. In order to overcome the above issues, multiple paths in the network may be provided. Presence of multiple interconnects at the end node can alleviate the hot-spots at the I/O bus. Presence of multiple paths in the network (as an example, Fat Tree, 3-D Torus etc.) with the help of multiple links between the switches may help the hot-spot avoidance. The presence of multiple paths and multiple interconnects allows network failover in the presence of network faults.

In this dissertation, we design a communication sub-system; *communications and network fault tolerance layer*, to leverage the presence of multiple paths in the network for communication performance, avoidance of hot-spots and network fault tolerance. In designing our communication sub-system, we aim to achieve the following goals:

1. *High Performance With Multi-Pathing:* Our design should provide high performance in the presence and absence of hot-spots at the I/O bus. It should provide efficient scheduling policies for different multi-pathing configurations (multiple adapters, multiple ports and combinations) for different communication semantics and heterogeneous networks.

2. *Hot-spot Avoidance With Bandwidth Estimation:* Our design should provide hot-spot avoidance in the network using the multi-pathing mechanism provided by InfiniBand. It should leverage InfiniBand mechanisms for bandwidth estimation and efficient usage of multiple paths to avoid hot-spots in the network.

3. *Network Fault Tolerance:* Our design should provide efficient recovery from network faults with negligible overhead in the absence of faults. It should provide application transparent recovery from network faults using InfiniBand mechanisms. It should also handle network partitions and recover from them without application-restart, in addition to efficiently utilizing the recovered paths from network faults.

In the dissertation, we will investigate how to leverage InfiniBand mechanisms for achieving the above goals. Design of efficient scheduling policies, path bandwidth estimation, network fault detection; recovery, for different communication semantics will constitute the design.

The rest of the dissertation is organized as follows. In chapter 2, we present the background of our work including MPI communication semantics, InfiniBand and access layers. In chapter 3, we present the problem statement of our dissertation and research approaches we have used for the design of *communications and network fault tolerance layer*. In chapter 4, we present the design for supporting multi-rail InfiniBand clusters for MPI-1 communication semantics. The design for supporting MPI-2 one-sided communication semantics is discussed in chapter 5. In chapter 6, we discuss the design challenges for efficient utilization of multiple send/receive engines with 12x InfiniBand architecture. In chapter 7, we present the design for avoiding hot-spots in the network using InfiniBand multi-pathing mechanism. In chapter 8, we present a novel approach to maximize the utilization of independent paths in the network using a batch based striping and sorting methodology. In chapter 9, we present the design for network fault tolerance layer using InfiniBand Automatic Path Migration (APM). In chapter 10, we use the

reliable connection semantics for network fault detection and present the design for recovery from network faults and network partitions. In chapter 11, we present the overview of MVAPICH [42], which we have used as a research vehicle and integrated our solutions to be used by high performance community at large. In chapter 12, we conclude and present our future directions.

CHAPTER 2

BACKGROUND

In this chapter, we provide the background information of our work. We begin with a brief introduction of Message Passing Interface (MPI); its communication semantics and MPI protocols. This is followed by a discussion on the InfiniBand architecture, the communication and path migration states of a queue pair. We also provide an introduction to InfiniBand adapters, access layers available for InfiniBand and other RDMA-enabled interconnects.

2.1 Introduction to Message Passing Interface

Message Passing Interface (MPI) [38, 39] is a programming model for inter-process communication. In the past decade or so, MPI-1 has been widely used to write parallel applications ranging from molecular dynamics to car crash simulations. More recently, MPI-2 has been announced with support for advanced features like one-sided communication, dynamic process management and advanced datatype processing. Figure 2.1 represents the communication semantics in MPI. In the upcoming sections, we explain the two-sided communication semantics (also known as point-to-point communication) and one-sided communication.

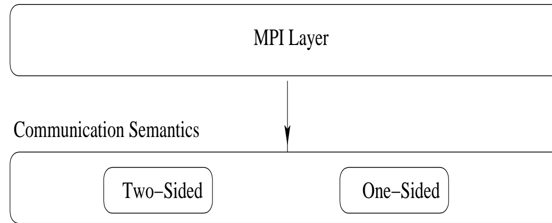


Figure 2.1: MPI Communication Semantics

2.1.1 MPI Point-to-Point Communication

Message Passing Interface (MPI) defines two types of communication protocols; *eager* and *rendezvous*. These protocols are handled by a component in the MPI implementation called *progress engine*. In the eager protocol, the message is pushed to the receiver side regardless of its state. In the rendezvous protocol, a handshake takes place between the sender and the receiver by control messages before the data is sent to the receiver side. Usually, the eager protocol is used for small messages and the rendezvous protocol is used for large messages. Figure 2.2 explains these protocols. The main communication paradigm of MPI is message passing. However, MPI is also implemented in systems which support shared memory [19, 24].

In an MPI program, two processes can communicate using MPI point-to-point communication functions. One process initiates the communication by using MPI_Send function. The other process receives this message by issuing MPI_Recv function. Destination processes need to be specified in both functions. In addition, both sides specify a *tag*. A send function and a receive function match only if they have compatible tags.

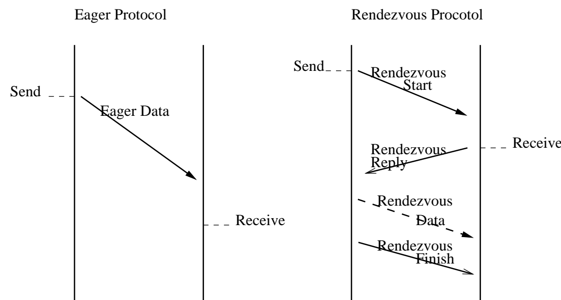


Figure 2.2: MPI Protocols

MPI_Send and MPI_Recv are the most frequently used MPI point-to-point functions. However, they have many variations. MPI point-to-point communication supports different *modes* for send and receive. The mode used in MPI_Send and MPI_Recv is called *standard* mode. There are other MPI functions that support other modes such as *synchronous*, *buffered* and *ready* modes. Communication buffers specified in MPI_Send and MPI_Recv must be contiguous. However, there are also variations of MPI_Send and MPI_Recv functions that supports non-contiguous buffers. Finally, any send or receive functions in MPI can be divided into two parts: one to initiate the operation and the other one to finish the operation. These functions are called *non-blocking* MPI functions. For example, MPI_Send function can be replaced with two functions: MPI_Isend and MPI_Wait. By using MPI non-blocking functions, MPI programmers can potentially overlap communication with computation, and hence increase the performance of MPI applications.

2.1.2 MPI One-Sided Communication

In many parallel scientific applications, the data distribution changes dynamically and the data access pattern is irregular. For such applications, each process computes the data it needs to access or update on other processes. However, a process may not know the location of data in its local memory, which needs to be read or updated by other processe(s). In some cases it may not even know the identification of the remote processe(s). Hence, in these situations, only one process in the communication is aware of all the parameters required to transfer the data. The emerging MPI-2 standard with one-sided communication operations specifically targets such communication patterns.

In MPI-2 one-sided communication, the sender can access the remote address space directly. Such one-sided communication is also referred to as *Remote Memory Access (RMA)* communication. In this model, the *origin process* (the process that issues the RMA operation) provides necessary parameters needed for communication. The area of memory on the *target process* accessible by the origin process is called a *Window*. MPI-2 specification defines various communication operations:

1. *MPI_Put* operation transfers the data to a window in the target process
2. *MPI_Get* operation transfers the data from a window in the target process
3. *MPI_Accumulate* operation combines the data movement to target with a reduce operation

As per the semantics of one-sided communication, the return of the one-sided operation call does not guarantee the completion of the operation. In order to

guarantee the completion of one-sided operation, explicit synchronization operations must be used. In MPI-2, synchronization operations are classified as *active* and *passive*. Active synchronization involves both sides of communication while passive synchronization only involves the origin side.

The period between two synchronization calls is called as *access epoch* and *exposure epoch* on the origin and target process, respectively. MPI-2 semantics allows multiple communication calls during an access epoch. This is done to amortize the overhead of synchronization over multiple communication operations.

2.2 Overview of InfiniBand Architecture (IBA)

InfiniBand Architecture (IBA) [26] is an industry standard that defines a System Area Network (SAN) to design clusters offering low latency and high bandwidth. A typical IBA cluster consists of switched serial links for interconnecting both the processing nodes and the I/O nodes. The IBA specification defines a communication and management infrastructure for inter-processor communication. IBA also defines built-in Quality of Service (QoS) mechanisms which provide virtual lanes on each link and define service levels for individual packets. In an InfiniBand network, processing nodes are connected to the fabric by Host Channel Adapters (HCA). HCAs are associated with processing nodes and their semantic interface to consumers is specified in the form of InfiniBand Verbs. Channel Adapters usually have programmable DMA engines with protection features.

IBA mainly aims at reducing the system processing overhead by decreasing the number of copies associated with a message transfer and removing the kernel from the critical messaging path. The InfiniBand communication stack consists

of different layers. The interface presented by Channel Adapters to consumers belongs to the transport layer. A Queue Pair (QP) based model is used in this interface. Figure 2.3 illustrates the InfiniBand Architecture.

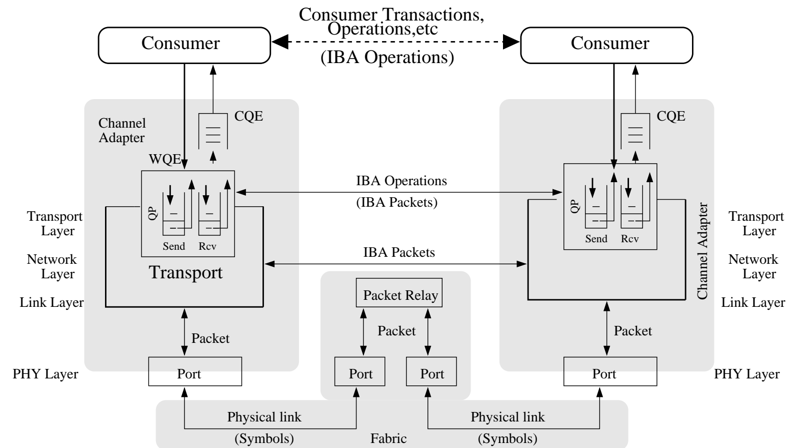


Figure 2.3: InfiniBand Architecture [26]

2.2.1 Transport Services

InfiniBand Architecture supports multiple classes of communication services at the transport layer. A queue pair can be configured for the following classes of services:

- Reliable Connection (RC)
- Reliable Datagram (RD)
- Unreliable Connection (UC)
- Unreliable Datagram (UD)

- Raw Datagram

In Reliable Connection service, each queue pair can only communicate with one queue pair at a remote node. Before communication, a connection must be established between the local queue pair and the remote queue pair. By using mechanisms such as acknowledgment and retransmission, InfiniBand ensures that the notification is reliable. Unreliable Connection is similar to Reliable Connection service. The difference is that reliability is not guaranteed.

Reliable Datagram and Unreliable Datagram provide connection-less transport services. A single queue-pair can communication with multiple queue pairs on remote nodes. In Reliable Datagram, reliability is provided by using *End-to-End Context*. UD service does not guarantee any reliability. Another restriction of UD is that the length of message size cannot exceed the Maximum Transfer Unit (MTU) of the InfiniBand network, which is typically 2048 bytes.

The purpose of Raw Datagram service is to provide interoperability of IBA and other networks. Its usage is out of the scope of this dissertation.

2.2.2 Management Infrastructure

Unlike many other interconnects, InfiniBand Architecture has a comprehensive management infrastructure. InfiniBand networks usually consist of smaller networks called *subnets*. From the network management perspective, InfiniBand defines an entity called *subnet manager*, which is responsible for discovery, configuration and maintenance of a network. Each InfiniBand port in a network is identified by one or more local identifiers (LIDs), which are assigned by the subnet manager. Since InfiniBand supports only destination based routing, each switch in

the network has a routing table corresponding to the LIDs in the subnet. However, InfiniBand supports only deterministic routing, and decisions to route messages adaptively cannot be taken by the intermediate switches. InfiniBand provides a mechanism, LID Mask Count (LMC), in which each port can be assigned multiple LIDs, to exploit multiple paths in the network. The current generation of subnet managers allow us to leverage the multiple paths provided by different topologies like Fat Tree [31].

2.2.3 Overview of Communication State Transition

As discussed in the previous section, InfiniBand supports different classes of transport services (Reliable Connection, Unreliable Connection, Reliable Datagram and Unreliable Datagram). In the Reliable connection model, each process-pair creates a unique entity for communication, called *Queue Pair (QP)*. Each QP consists of two queues; *send queue* and *receive queue*. Figure 2.4 shows the communication state transition sequence for a QP. Each QP has a combination of *communication state* and *path migration state*. Figure 2.4 shows the QP communication state transition diagram. Figure 2.5 shows a combination of communication and path migration state transition for the QP. In this section, we focus only on the communication state. In the next section, we discuss the combinations of these states.

At the point of QP creation, its communication state is RESET. From this state it is brought to the INIT state by invoking modify_qp function. The modify_qp function is provided by the access layer of InfiniBand for different transition states provided by InfiniBand specification [26]. During the RESET-INIT transition, the

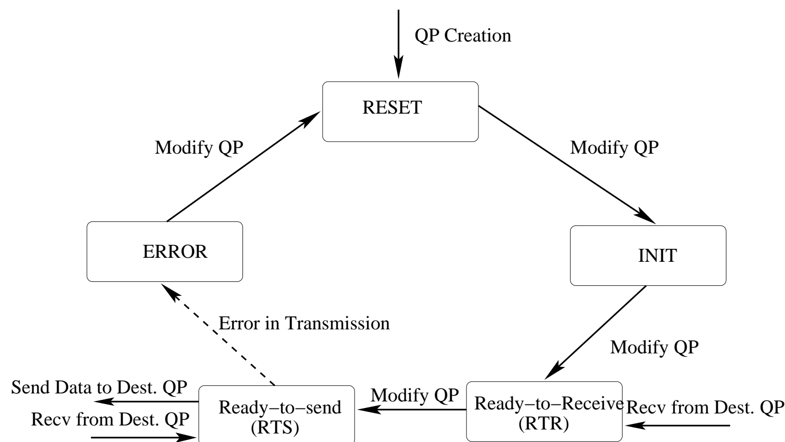


Figure 2.4: QP Communication State Diagram

QP is specified with the HCA port to use in addition to the atomic flags. Once in the INIT state, the QP is specified with the destination LID and the destination QP from which it will receive the messages. A modify_qp call brings it to READY-TO-RCV (RTR) state. At this point, the QP is ready to receive the data from the destination QP. Finally, QP is brought to READY-TO-SEND (RTS) state by specifying associated parameters and making the modify_qp call. At this point, the QP is ready to send and receive data from its destination QP. Should any error(s) occur on the QP, the QP moves to the ERROR state. At this state, the QP is broken and cannot communicate with its destination QP. In order to re-use this QP, it needs to be brought back to the RESET state and the above-mentioned transition sequence (RESET-RTS) needs to be executed.

The request(s) to send the data to the destination are placed on the send queue, by using a mechanism called *descriptor*. A descriptor describes the information necessary for a particular operation. For RDMA operation, it specifies

the local buffer, address of the peer buffer and access rights for manipulation of remote buffer. The completions of descriptors are posted on a queue called *completion queue*. Each entry in the completion queue is called *completion queue entry (CQE)*. This mechanism allows a sender to know the status of the data transfer operation. Different mechanisms for notification are also supported (polling and asynchronous).

2.2.4 Overview of Automatic Path Migration

Automatic Path Migration (APM) is a feature provided by InfiniBand which enables transparent recovery from network fault(s) by using the alternate path specified by the user. Automatic path migration is available for Reliable Connected (RC) and Unreliable Connected (UC) QP Service Type. For this feature, InfiniBand specifies *Path Migration States* associated with a QP. A valid combination of communication and path migration states are possible. This is shown further in Figure 2.5. In the figure, the path migration state of the QP is shown using oval shape. The possible communication states of the QP are shown using curly braces. At a point of time, only one of the communication states is applicable to a QP.

Once the QP is created, the initial path migration state for a QP is set to MIGRATED. At this point, the QP can be in RESET, INIT or RTS communication state. Once the transition sequence (RESET-RTS) is complete, the QP's path migration state goes back to MIGRATED. Hence, this state is valid for QPs having their communication state as RTS. APM defines a concept of alternate path, which is used as an escape route, should an error occur on the primary path of

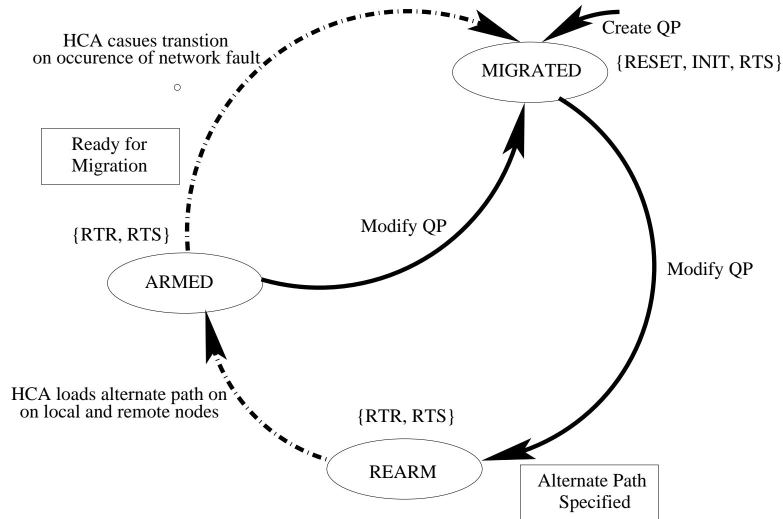


Figure 2.5: QP Path Migration State Diagram (Adaptation From InfiniBand Specification [26])

communication. The alternate path is specified by the user. This specification of the alternate path can be done at any point, beginning the INIT-RTR transition of the QP. Once this has been specified, the HCA can be requested to load this path. This is done by specifying the QP's path migration state to REARM. Once the path has been loaded, the path migration state of a QP is ARMED. During this state, the alternate path can be switched over to function as a primary path. This can be done by HCA automatically, should an error occur on the primary path of communication. This is shown with dotted line in Figure 2.5. As an alternative, a user can manually request the alternate path to be used as the primary path of communication. This is shown with solid line in Figure 2.5.

2.2.5 Quality-of-Service Support

InfiniBand has built-in Quality-of-Service support that consists of three components: Virtual Lanes (VLs), Service Levels (SLs), and Service Level to Virtual Lane (SL-VL) mapping. In an InfiniBand network, each physical link is divided into up to 16 Virtual Lanes. Each VL has different QoS characteristics. At end-points, SLs are assigned to communication and packets are marked with their service levels. As packets travel through the network, they are assigned to different VLs on each link according to SL-VL mapping.

2.2.6 Hardware Support for Congestion Notification

InfiniBand provides hardware support for congestion notification. This mechanism uses simple switch based Explicit congestion notification (ECN) mechanism and source response mechanism using rate control combined with a window limit [62, 20, 52]. Using Forward-Backward and Explicit congestion notification mechanisms, the end nodes and intermediate hops can be notified of congestion. Mechanisms from congestion recovery are beyond the scope of the InfiniBand specification.

2.3 Overview of InfiniBand Adapters and Access Layers

As discussed in the previous sections, the InfiniBand Architecture (IBA) [26] defines a System Area Network with a switched, channel-based interconnection fabric. IBA 4x can provide bandwidth up to 10 Gbps. Switches and Adapters with capabilities of 12x bandwidth have also become available in the market, providing performance upto 30Gb/s. InfiniBand defines Verbs for user-level applications

to leverage its capabilities. Verbs API (VAPI) by Mellanox [3] has been widely used for powering large scale clusters. In addition, an open source effort, OpenFabrics [44] has also become available. The Ammasso interconnect is a RDMA-enabled Gigabit Ethernet adapter [25]. It is a full duplex 1Gbps Ethernet Adapter also provides the interface for vanilla sockets based applications. Ammasso defines a CCIL interface, for applications to leverage its RDMA capabilities.

2.3.1 Overview of InfiniBand Adapters

In this section, we provide an overview of the InfiniBand Adapters we have used for evaluation during the preliminray work.

Overview of Mellanox Dual-Port Adapters

The first generation InfiniBand Adapters were designed by Mellanox Corporation [3]. The first adapters were dual-port based on the PCI-X bus based I/O interface. Each InfiniBand port was capable of providing a theoritical 10Gb/s in each direction. However, due to the limitations of PCI-X interface, a peak exchange bandwidth of only 10Gb/s could be achieved. Motherboards with support for multiple independent PCI-X interfaces could allow the usage of multiple adapters on a single node.

With the advent of PCI-Express, the peak theoritical bandwidth in each direction has increased to 20Gb/s. Mellanox Adapters with support of PCI-Express interface have also become available. Since each port still provided 10Gb/s, multiple ports can be used together to achieve the peak bandwidth. Recently, the

support for Double Data Rate (DDR) mechanism has been announced, which allows a 20Gb/s bandwidth to be achieved using a single port. In future, the support for Quad Data Rate (QDR) is being planned.

Overview of IBM 12x Dual-Port InfiniBand Adapter

Each IBM 12x HCA comprises of two ports. The local I/O interconnect used is GX+, which can run over different clock rates of 633 MHz-950 MHz. Figure 2.6 shows the block diagram of the IBM 12x InfiniBand HCA. In this dissertation, we have used GX+ bus with 950MHz frequency. As a result, a theoritical bandwidth of 7.6GB/s can be provided. However, each port can provide an aggregate theoritical bandwidth of 12x (3GB/s). Each 12x HCA port has multiple send and receive DMA engines. The aggregate link bandwidth of the send DMA engines and receive DMA engines is 12x in each direction, respectively. However, the peak bandwidth of each send/recv engine varies with different implementations.

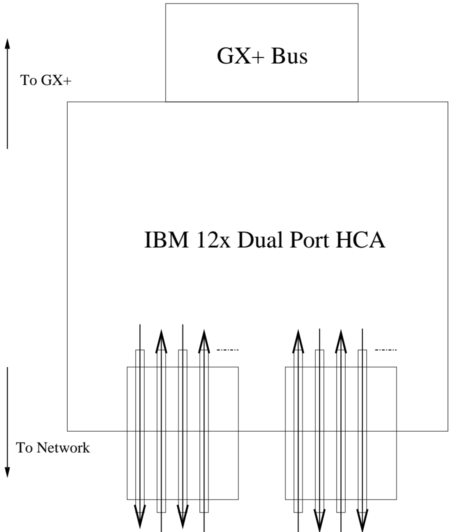


Figure 2.6: IBM 12x InfiniBand HCA Block Diagram

To schedule the data on a send engine, the hardware send scheduler looks at the send queues of different queue pairs with send descriptors, which are not being serviced currently. Given equal priority, the queue pairs are serviced in a round robin fashion. In the presence of traffic sufficient to keep the scheduling engines busy, an aggregate bandwidth of 12x can be achieved in each direction.

2.3.2 Overview of Access Layers

In this section, we discuss the access layers provided for InfiniBand and other RDMA enabled interconnects. Some of the access layers are specific to InfiniBand, however other layers are more generic. Figure 2.7 shows a relation between between these layers. uDAPL (user direct access provide library) is a generic layer providing a common interface for all RDMA-enabled interconnects. We discuss these layers in the upcoming section.

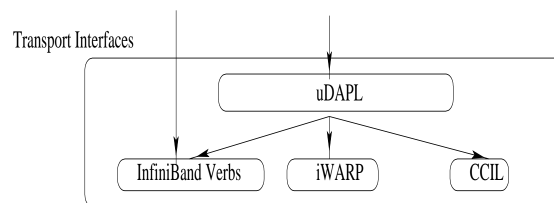


Figure 2.7: Access Layers for InfiniBand and Other RDMA-enabled Interconnects

InfiniBand and iWARP Specific Access Layers:

As mentioned in the previous sections, InfiniBand provides a plethora of hardware features for designing programming models and other user applications. APIs to leverage these features have emerged. Verbs API (VAPI) from Mellanox [3] has

become available for the last couple of years. Recently, an open source effort for design and development of API supporting InfiniBand and iWARP (Internet Wide Area RDMA protocols) [1] has emerged [44]. MPI implementations supporting iWARP interface have also become available [41].

Overview of uDAPL Interface:

As mentioned in the previous section, multiple interconnects have emerged that provide RDMA capabilities. However, these interconnects do not provide a common set of Application Programming Interfaces (APIs). In addition, upcoming interconnects face a similar challenge and the turn-around time for developing MPI [38, 39] on these adapters can be prohibitive. To alleviate this situation, Direct Access Transport (DAT) Collaborative [16] has defined a DAPL interface, providing a common interface for different interconnects. User Direct Access Programming Library (uDAPL) is a lightweight, transport-independent, platform-independent user-level library, potentially capable of providing high productivity for upcoming and existing interconnects.

uDAPL allows processes to communicate by defining *End Points (EPs)*. EPs need to be connected to each other, before communication can take place. *Work Requests or descriptors* can be posted on the EPs for sending or receiving data from other processes. uDAPL supports memory semantics by leveraging RDMA and channel semantics by providing *send/receive* mechanism. The completion status of the previously posted descriptors can be ascertained by using *completion queue* mechanism. Completion queue returns status of the posted descriptor, in terms of success/failure and the error code.

CHAPTER 3

PROBLEM STATEMENT AND RESEARCH APPROACHES

As discussed in the previous chapter, InfiniBand provides mechanisms for creation of multiple paths, hardware based error detection using Automatic Path Migration and mechanisms for data delivery notification, as discussed in the background section. However, a unified *communications and network fault tolerance layer*, which semantically abides the MPI layer with the InfiniBand features, is imperative for alleviating network and I/O hot-spots, and providing network fault tolerance. Hence, the question we address in this dissertation is:

> *How to design an efficient communications and network fault tolerance layer over multi-pathing leveraging the novel InfiniBand features?*

Figure 3.1 shows the communications and network fault tolerance layer and its interactions with MPI and InfiniBand layers. The communications and network fault tolerance layer itself should comprise of multiple functionalties. It should provide efficient utilization of multi-rail configurations in the absence of hot-spots and network faults. In the presence of hot-spots and congestion, it should utilize the hot-spot free paths efficiently. In the presence of network faults, this layer should do error detection and recovery from the network faults. This leads us to the following challenges, which are addressed in this dissertation:

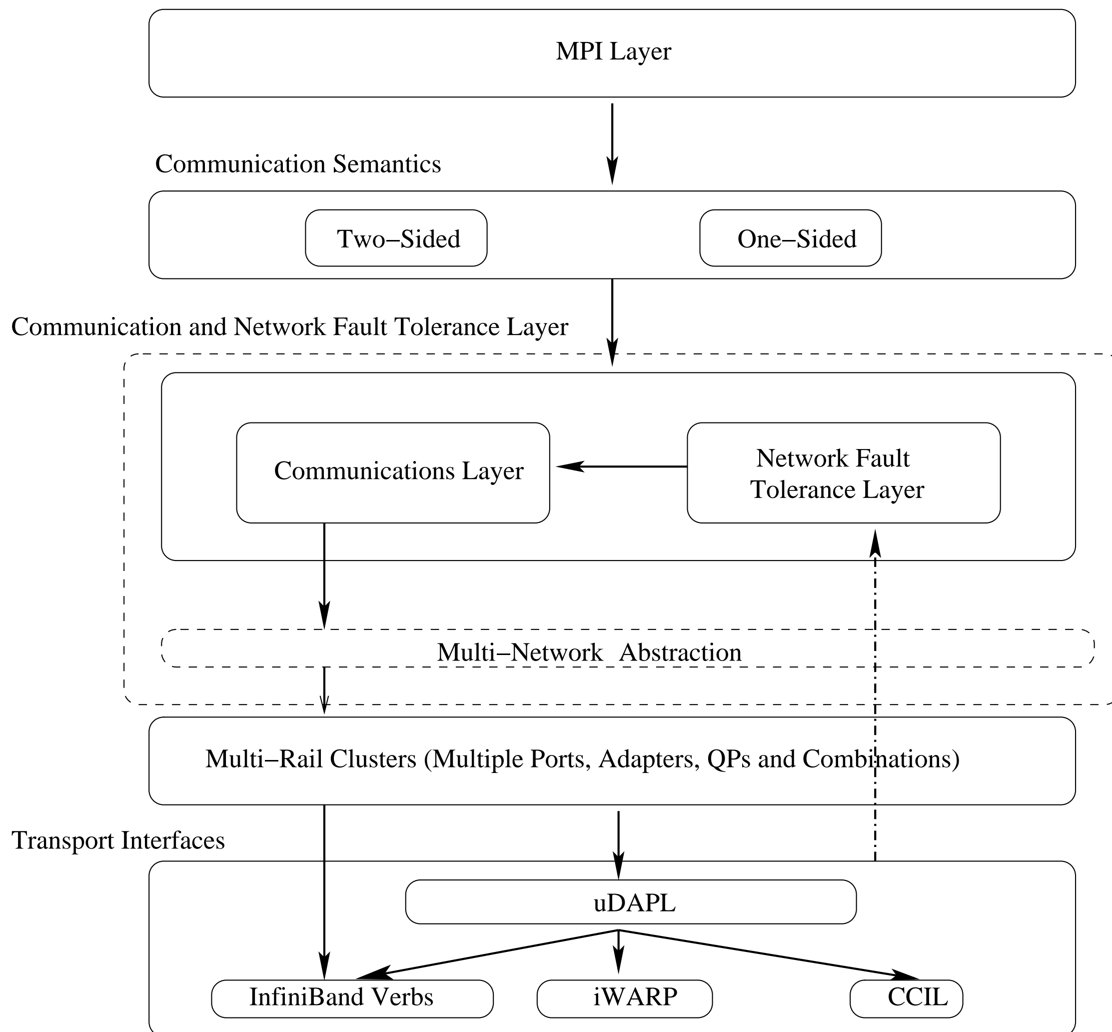


Figure 3.1: Overall Interaction of Communications and Network Fault Tolerance Layer

- How to design an efficient communications layer for various multi-rail configurations with different access layers?
- How to leverage InfiniBand features for avoiding hot-spots in the network and the end nodes?
- How to design efficient error detection and recovery, modules for fault tolerance layer using InfiniBand mechanisms?

Figure 3.2 shows the scope of the dissertation showing the relationship between the communications and network fault tolerance layer with InfiniBand networking mechanisms. In the upcoming sections, we present the design components associated with each of the above functionalities. The objective of the framework is to leverage the InfiniBand features efficiently, with minimal overhead and achieving optimal performance. We present the problem statement in detail as follows:

- Can we design a communications layer which provides efficient data transfer for MPI-1 two-sided communication semantics? - Multi-rail configurations comprising of multiple ports, multiple adapters, multiple send/recv engines per port are emerging with combinations. Efficient utilization of these multi-rail configurations is imperative to provide the best performance to MPI applications. The MPI library should provide an equivalent abstraction for such configurations and provide efficient scheduling policies for transfer with eager and rendezvous protocols. In addition, the benefits of using multi-rail configurations should provide benefits to point-to-point and collective communication primitives.

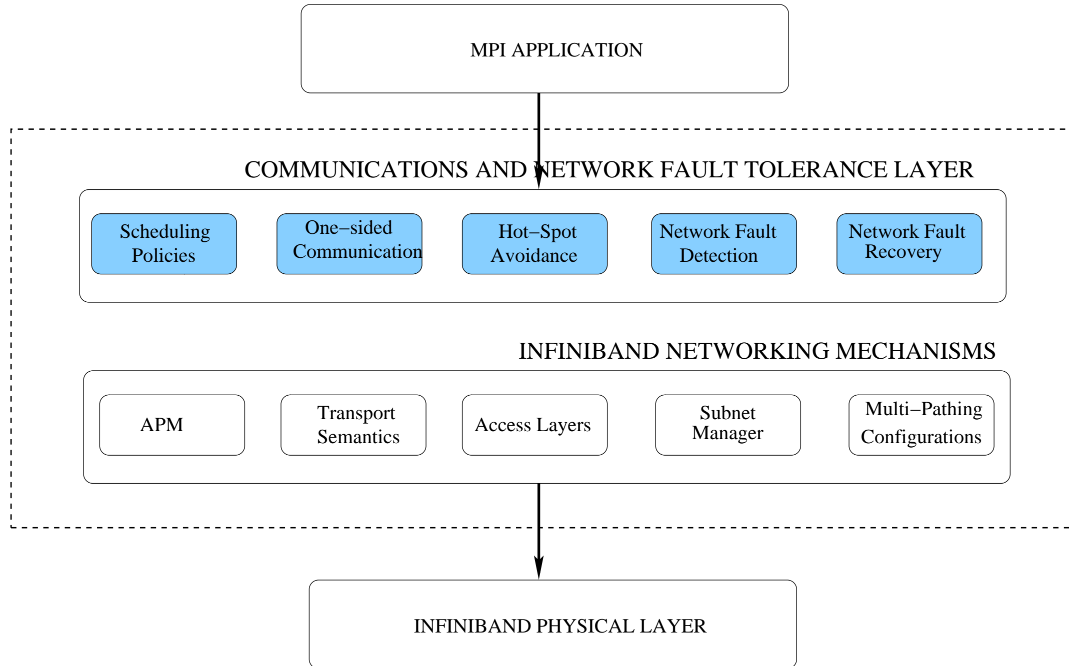


Figure 3.2: Proposed Mapping of Multi-Pathing MPI Design Components with InfiniBand Features

- Can we design efficient communications layer for one-sided communication semantics? - Different communication semantics employ different communication protocols. Traditional two-sided semantics require rendezvous protocol for communication of large messages as discussed in the background section. However, such protocols may be inefficient for one-sided communication. Out-of-ordering relaxation also makes it imperative to re-visit the discussion on scheduling policies. Unlike two-sided communication protocols which typically use either RDMA Write or Read, One-sided communication primitives map directly to RDMA Write and Read for MPI_Put and MPI_Get respectively. Efficient scheduling of these requests for utilizing peak bandwidth brings additional challenges to one-sided communication. The MPI

library should provide efficient scheduling policies for one-sided communication, taking advantage of ordering relaxation, absence of rendezvous protocol

- Can we design an MPI functionality for efficient utilization of network paths and providing hot-spot avoidance? - Looking at the TOP500 [6] list, a majority of InfiniBand clusters use *fat tree [31]* has become a popular interconnection topology for these clusters, since it allows multiple paths to be available in between a pair of nodes. However, even with fat tree, hot-spots may occur in the network depending upon the route configuration between end nodes and communication patterns in the application. To make matters worse, the deterministic routing nature of InfiniBand limits the application from effective use of multiple paths transparently and avoid the hot-spots in the network.

  To explain the problem, we take a cluster with a fat-tree switch and execute an MPI program using this switch to understand the contention and occurrence of hot-spots in the network. Figure 3.3 represents the switch topology used for our experimentation. Each switch block consists of 24 ports. The leaf switches (referred to as leaf blocks from here onwards) have 12 ports available to be used by the end nodes, the other 12 ports are connected to the spine switches (referred to as spine blocks from here onwards). In the figure, blocks 1 - 12 are leaf blocks; blocks 13 - 24 are spine blocks. The complete switch has 144 ports available for end nodes. Each block is a crossbar in itself.

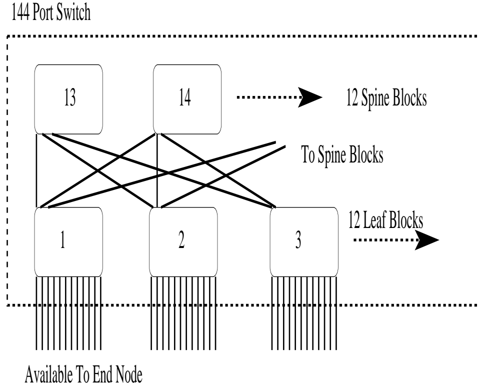


Figure 3.3: 144-port InfiniBand Switch Block Diagram

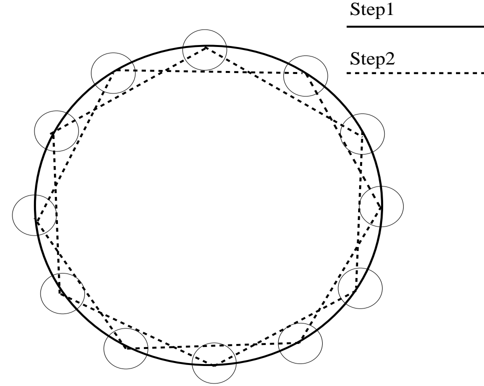


Figure 3.4: Communication Steps in Displaced Ring Communication

To demonstrate the contention, we take a simple MPI program, which performs ring communication with neighbor rank increasing at every step. The communication pattern is further illustrated in the Figure 3.4 (only step1 and step2 are shown for clarity). Executing the program with n processes takes n-1 steps. Let $rank_i$ denote the rank of the ith process in the program. and $step_j$ denote the jth step during execution. At $step_j$, an MPI process with $rank_i$ communicates with MPI process $rank_{i+j}$. This communication pattern is referred to as *DRC (Displaced Ring Communication).*

We take an instance of this program with 24 processes and schedule MPI processes with $rank_0$ - $rank_{11}$ on nodes connected to block 1 and $rank_{12}$ - $rank_{23}$ on block 2. Since each block is a crossbar in itself, no contention is observed for intra-block communication. However, as the step iteration increases, the inter-block communication increases and a significant link contention is observed. The link contention observed during the step 12 (each process doing

inter-block communication) is shown in Figure 3.5, with thicker solid lines representing more contentions.

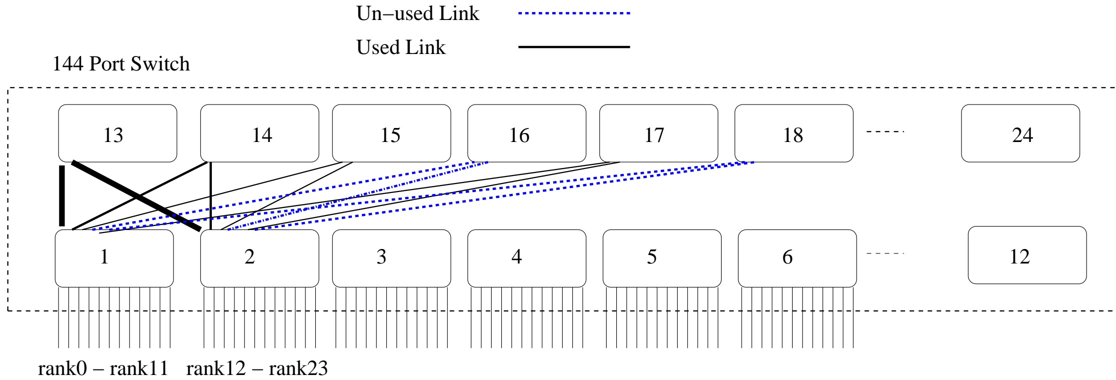


Figure 3.5: Link Usage with Displaced Ring Communication

From Figure 3.5, we can see that some links are over-used to a degree from four to zero. As the degree of link usage increases, the bandwidth is split amongst the communication instances using the link(s), making them *hot spots*. In our example, paths using block 13 split bandwidth for four different communication instances making the set of links using this block hot-spots.

Even though, there are sufficient links for an independent path of communication between block 1 and 2 (using spine blocks), DRC is not able to utilize them in a contention free manner. The usage of these links is highly dependent upon the path configuration done by the subnet manager. This route configuration done by the subnet manager may benefit one communication pattern and show contention for other patterns, leaving un-utilized links in

the network. Under such a scenario, the utilization of the links is the responsibility of the MPI implementation. Hence, designing an efficient MPI library, with effective use of multiple paths is critical to hot-spot avoidance.

As discussed in the previous section, reliable connection transport semantics provides notification of data delivery. This mechanism may be used in the estimation of the path bandwidth. At the same time, LMC mechanism may be used for the creation of multiple paths in the network. Using the LMC mechanism and reliable connection semantics, different methods of notification (polling and asynchronous) can be used for the determination of path bandwidth. Congestion notification mechanisms proposed in InfiniBand with window rate control can also be combined with the above mechanisms for controlling congestion and using hot-spot free paths.

- Can we design an efficient network fault tolerance layer with advance InfiniBand mechanisms and minimal performance penalty?

  Increasing scale of InfiniBand clusters has reduced the Mean Time Between Failures (MTBF) of components. Network component is one such component of clusters, where failures of network interface cards (NICs), cables or switches breaks the existing path(s) of communication. In the worst case, network partitions may be created. However, network failures should not lead to application abort. InfiniBand provides hardware and software based mechanisms for network fault detection and recovery. InfiniBand provides a hardware mechanism, *Automatic Path Migration (APM)*, which allows user transparent detection and recovery from network fault(s). In addition, the

completion queue based mechanism can be used for the completion status of data transfer requests. The Network fault tolerance layer should comprise of modules for *network fault detection* and *network failover*. The completion queue mechanism discussed in the previous section can be used for detection of network faults. Alternatively, the hardware mechanism may also be used for detection of network faults transparent to the user. As discussed in the previous section, the hardware mechanism, APM requires specification of an alternate path. Hence, selecting a maximal independent path can be used as the alternate path. For software based aproach, efficient detection of network faults, efficient re-transmission of data transfre requests and handling of network partitions without application restart.

3.1 Dissertation Overview

In this section, we present the overview of the dissertation. In next couple of chapters, we explain each of these approaches in detail.

In chapter 4, we present an in-depth study of designing high performance multi-rail InfiniBand clusters for MPI-1 primitives. We discuss various ways of setting up multi-rail networks with InfiniBand and propose a unified MPI-1 design that can support all these approaches. By taking advantage of RDMA operations in InfiniBand and integrating the multi-rail design with MPI communication protocols, our design supports multi-rail networks with very low overhead. Our performance results show that the multi-rail 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 compared with the original MPI.

In chapter 5, we present the challenges *(Multiple synchronization messages, handling multiple HCAs, scheduling policies, ordering relaxation)* associated with designing MPI-2 one-sided communication over multi-rail InfiniBand networks. We implement our design and presented the performance evaluation for micro-benchmarks. We observe that multi-rail InfiniBand clusters can significantly improve the performance for one-sided communication. Using a two rail cluster, we can achieve almost double the throughput and reduce the latency to half with *MPI_Put* and *MPI_Get* operations for large messages.

In chapter 6, we focus on designing an MPI substrate for IBM 12x InfiniBand Architecture. We discuss with the introduction of overall design, and present the limitations of previously proposed designs in achieving the peak performance of IBM 12x InfiniBand architecture. We present the need for re-visiting the scheduling policies, depending upon the communication pattern in the application. We present communication marker module, which resides in the ADI layer and differentiates between communication patterns.

Large scale InfiniBand clusters are becoming increasingly popular, as reflected by the TOP 500 [6] Supercomputer rankings. At the same time, *fat tree* [31] has become a popular interconnection topology for these clusters, since it allows multiple paths to be available in between a pair of nodes. However, even with fat tree, hot-spots may occur in the network depending upon the route configuration between end nodes and communication patterns in the application. In chapter 7, we present the design for an MPI functionality, Hot-Spot Avoidance with MVAPICH (HSAM) which provides hot-spot avoidance for different communication patterns, without apriori knowledge of the pattern. We leverage LMC (LID Mask Count)

mechanism of InfiniBand to create multiple paths in the network, and study its efficiency in creation of contention free routes.

In chapter 8, we re-visit the approach presented in the previous chapter for a better utilization of multiple independent paths in the network. To efficiently utilize physically independent paths, we propose a novel scheduling policy, which performs Batch-based Striping and Sorting (BSS) during the application execution to adaptively eliminate the path(s) with low bandwidth. Using MPI_Alltoall, we achieve an improvement of 27% and 32% in latency with different BSS policy configurations compared to the best configuration of the HSAM scheme on 32 and 64 processes, respectively.

In chapter 9, we design a set of modules; *alternate path specification module*, *path loading request module* and *path migration module*, which work together for providing network fault tolerance for user level applications. We integrate these modules for simple micro-benchmarks at the Verbs Layer, the user access layer for InfiniBand, and study the impact of different state transitions associated with APM. We also integrate these modules at the MPI [38, 39] (Message Passing Interface) layer to provide network fault tolerance for MPI applications. Our performance evaluation shows that APM incurs negligible overhead in the absence of faults in the system.

In chapter 10, we design a network fault tolerant MPI using uDAPL interface, making this design portable for existing and upcoming interconnects. Our design provides failover to available paths, asynchronous recovery of the previous failed paths and recovery from network partitions without application restart. In addition, the design is able to handle network heterogeneity, making it suitable for the

current state of the art clusters. To achieve these goals, we design a set of low overhead modules *completion filter and error-detection*, *message (re)-transmission* and *path recovery and network partition handling* which perform completion filter and detection, (re)-transmission and recovery from network partitions respectively.

CHAPTER 4

EFFICIENT MPI-1 DESIGN FOR MULTI-RAIL INFINIBAND CLUSTERS

One of the primary reasons for InfiniBand success is its high performance, in addition to the presence of a plethora of advance features. 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 multi-way SMP machines, in which multiple processes may run on a single node and must share the node bandwidth. Many-core machines proposed by chip makers including Intel and AMD are likely to aggravate this situation further. An important mechanism to overcome the bandwidth bottleneck is to use *multi-rail networks*. The basic idea is to have multiple independent networks (rails) to connect nodes in a cluster. With multi-rail networks, communication traffic can be distributed among different rails. There are two ways of distributing communication traffic. In multiplexing, messages are sent through different rails in a round robin fashion. In *striping*, messages are divided into several chunks and sent out simultaneously using multiple rails. By using these techniques, the bandwidth bottleneck can be alleviated [7, 21].

In this chapter, we present a detailed study in designing high performance MPI-1 with multi-rail InfiniBand clusters. We discuss various ways of setting up multi-rail 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 multi-rail design with MPI communication protocols [42, 45]. 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 study their impact on performance.

The rest of the chapter is organized as follows: In section 4.1, we present the basic architecture for supporting multi-rail InfiniBand networks. We discuss some of the implementation details in Section 4.2. In section 4.3, we present performance results of our multi-rail MPI. In section 4.4, we summarize the results and impact of this work.

4.1 Basic MPI Design for Multi-Rail Networks

The basic architecture of our design to support multi-rail networks is shown in Figure 4.1. 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*.

The Communication Scheduler is the central part of our design. It accepts protocol messages from the MPI Protocol Layer, and stripes (or multiplexes) them

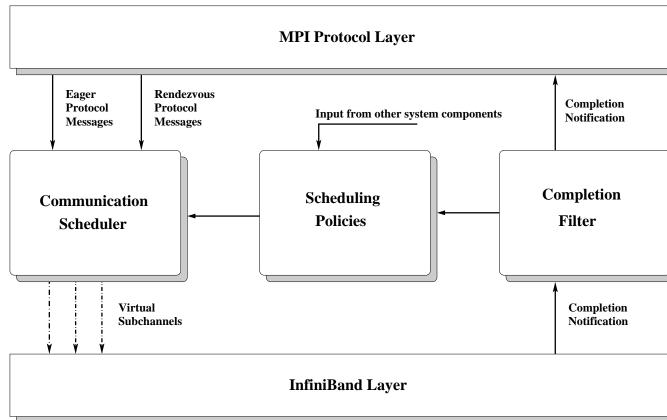


Figure 4.1: Basic Architecture

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 the information provided by the Scheduling Policies component.

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.

4.1.1 Virtual Subchannel Abstraction

Multi-rail networks can be built by using multiple Adapters on a single node, or by using multiple ports in a single HCA [14]. 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 may not necessarily correspond to a physical connection of the underlying communication layer.

In this work, we use an enhanced virtual channel abstraction to provide a unified solution to support multiple Adapters and multiple ports. 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 between the subchannels.

It is easy to see how this enhanced abstraction can deal with different multi-rail configurations, including multiple adapters and multiple ports. In the case of each node having multiple Adapters, subchannels for a virtual channel correspond to connections that go through different Adapters. In presence of multiple ports of the Adapters, subchannels can be set to have one connection for each port. Once all the connections are initialized, the same subchannel abstraction is used for communication in all cases. This idea is further illustrated in Figure 4.2.

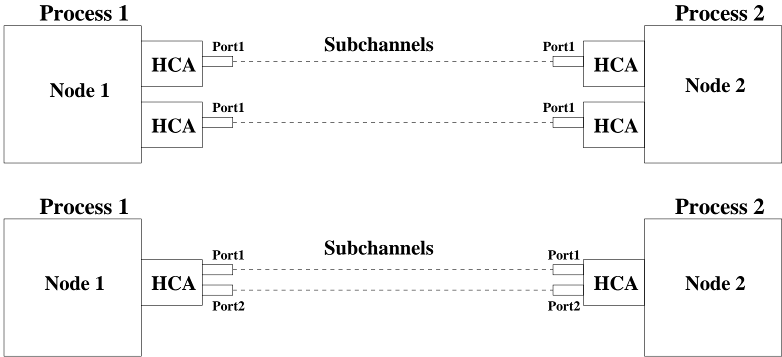


Figure 4.2: Virtual Subchannel Abstraction

4.1.2 Scheduling Policies

Different scheduling policies can be used by the Communication Scheduler to decide which subchannels to use for transferring each message.

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 [7]. 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.

4.2 Implementation Details

In this section, we present the implementation issues associated with our work. We begin with the discussion on handling multiple adapters, out-of-order messages and multiple RDMA completion notifications.

4.2.1 Handling Multiple Adapters

In the design section, we described how we can provide a unified design for multiple adapters and multiple ports. 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. Due to some restrictions in InfiniBand, there are two situations that must be handled differently for multiple Adapters:

- 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 [34, 32, 22, 13, 36]. 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 Adapter. Hence, we need to use multiple CQs for multiple Adapters. This results in slightly higher

overhead due to the extra polling of CQs. However, for bulk data transfer, this cost can be ignored.

Buffer registration also needs different handling for multiple Adapters. 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 Adapters, it must be registered with each one of them. Currently, we have used a simple approach of registering the complete buffer with all Adapters. Although this approach increases the registration overhead, this overhead can be largely avoided by using a registration cache [54].

4.2.2 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 en queued 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 *en queue*, *de queue*, 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.

4.2.3 Multiple RDMA Completion Notifications

In our design, large messages which use the rendezvous protocol are striped into multiple smaller 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 sub-channels. 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.

4.3 Performance Benefits with Multi-Rail Design on MPI-1 Benchmarks

In this section, we evaluate the performance of our multi-rail MPI design over InfiniBand. We show the performance benefit we can achieve compared with the original MPI implementation. Due to the limitation of our testbed, we focus on multi-rail networks with multiple Adapters in the section. The performance evaluation is done with two different configurations: Uni-Processor (UP) Mode and Shared Memory (SMP) Mode. In the former configuration, only one process is used per node. In the second configuration, two or more processes are used on every node. For the SMP mode, loopback is used for performance evaluation.

We use a set of MPI benchmarks for evaluation. We use a ping-pong MPI Latency Test, a Uni-directional Bandwidth Test and Bi-directional Bandwidth Test. In all of these tests, two processes are involved. A detailed description of these benchmarks is present at the MVAPICH homepage [42].

4.3.1 Experimental Testbed

Our testbed cluster of 8 SuperMicro SUPER X5DL8-GG nodes with Server-Works 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 Adapters 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 Adapters and only one port of each HCA is connected to the switch. The ServerWorks GC LE chipsets have two separate I/O bridges. To reduce the impact of I/O bus, the two Adapters are connected to 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 Adapters. The kernel version we used is Linux 2.4.22smp. The InfiniHost SDK version is 3.1 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.

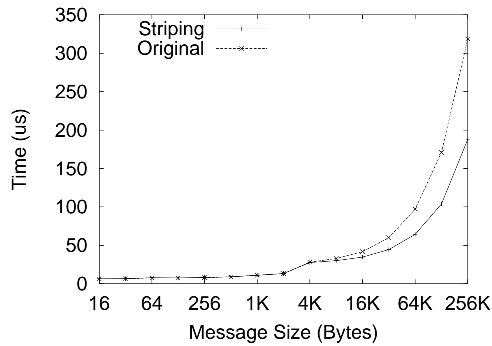


Figure 4.3: MPI Latency (UP mode)

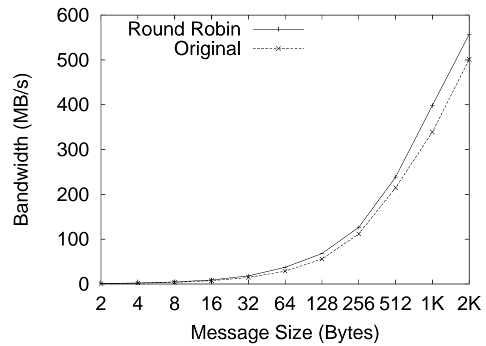


Figure 4.4: MPI Bandwidth (Small Messages, UP mode)

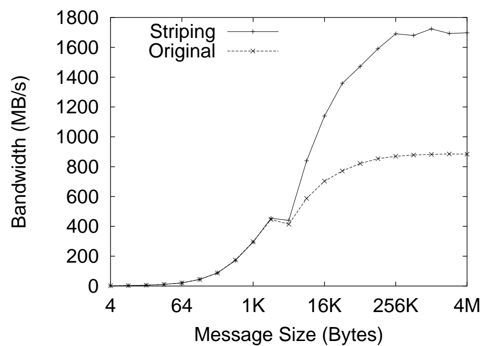


Figure 4.5: MPI Bandwidth (UP mode)

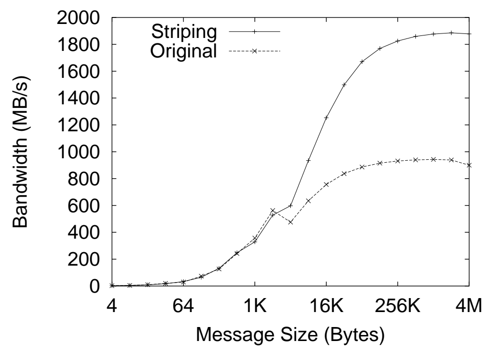


Figure 4.6: MPI Bidirectional Bandwidth (UP mode)

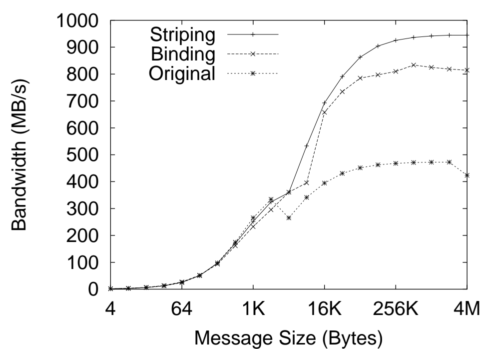


Figure 4.7: MPI Bandwidth (SMP mode)

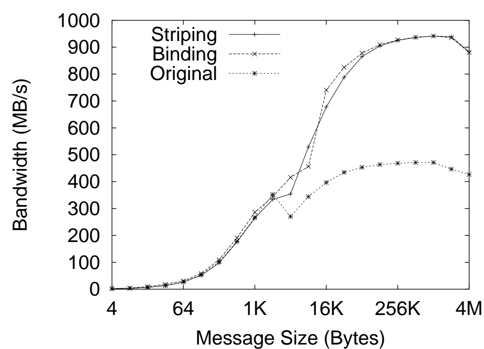


Figure 4.8: MPI Bidirectional Bandwidth (SMP mode)

4.3.2 Performance Evaluation with Point-to-Point and Collective Communication

To evaluate the performance benefit of using multi-rail networks, we compare our new multi-rail MPI with our original MPI implementation. In the multi-rail 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, bandwidth and bi-directional bandwidth. We then present results for collective communication by using Pallas MPI benchmarks [2]. Finally, we carry out application level evaluation by using some of the NAS Parallel Benchmarks [8] 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 4.3, 4.5 and 4.6, we show the latency, bandwidth and bidirectional bandwidth results in UP mode. We also show bandwidth results for small messages in Figure 4.4. (Note that in the x axis of the figures, unit K is an abbreviation for 2^{10} and M is an abbreviation for 2^{20}.) From Figure 4.3 we can see that for small messages, the original design and the multi-rail design perform comparably. The smallest latency is around 6 μs for both. However, as message size increases, the multi-rail design outperforms the original design. For large messages, it achieves about half the latency of the original design. In Figure 4.5, we can observe that multi-rail design can achieve significantly higher bandwidth. The peak bandwidth for the original design is around 884 MB/s. With the multi-rail 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 4.6

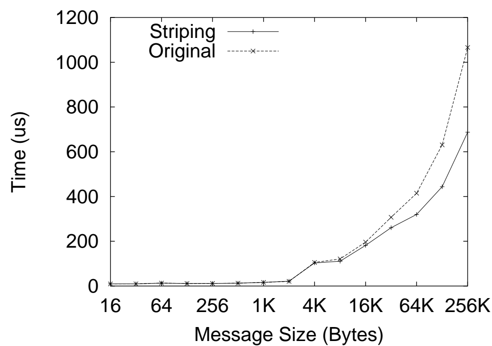
show a similar trend. The peak bidirectional bandwidth is around 943 MB/s for the original design and 1877 MB/s for the multi-rail design. In Figure 4.4 we can see that the round robin scheme can slightly improve bandwidth for small messages compared with the original scheme.

For Figures 4.7 and 4.8, we have used two processes on each node, each of them sending or receiving data from a process on the other node. It should be noted that in the bandwidth test, the two senders are on separate nodes. For the multi-rail design, we have shown results using both even striping policy and binding policy for large messages. Figure 4.7 shows that both striping and binding performs significantly better than the original design. We can also see that striping does better than binding. The reason is that striping can utilize both adapters in both directions while binding only uses one direction in each adapter. Since in the bidirectional bandwidth test in SMP mode, both Adapters are utilized for both directions, striping and binding perform comparably, as can be seen from Figure 4.8.

In Figures 4.9, 4.10, 4.11 and 4.12, we show results for Broadcast and 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 multi-rail design, we can achieve significant performance improvement for large messages compared with the original design.

4.3.3 Performance Evaluation with MPI Applications

In Figures 4.13 and 4.14 we show the application results. We have chosen the IS and FT applications (Class A and Class B) in the NAS Parallel Benchmarks



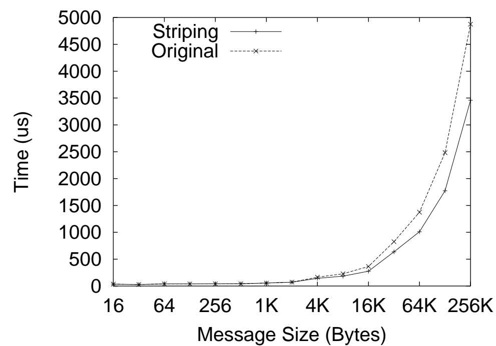
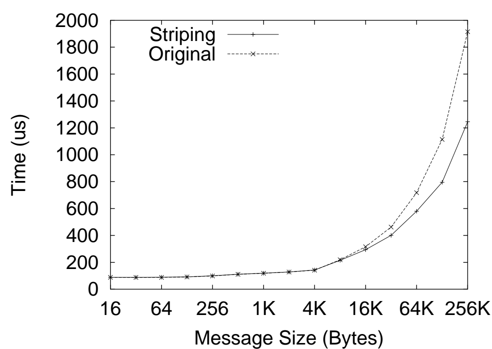


Figure 4.9: MPI_Bcast Latency (UP mode)

Figure 4.10: MPI_Alltoall Latency (UP mode)



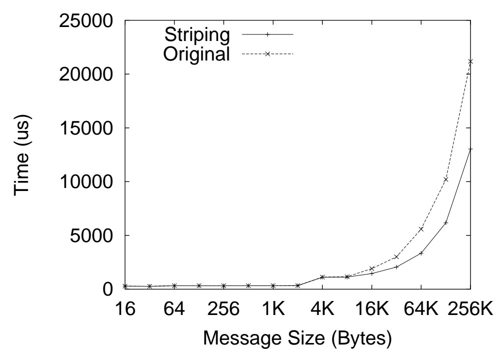


Figure 4.11: MPI_Bcast Latency (SMP mode)

Figure 4.12: MPI_Alltoall Latency (SMP mode)

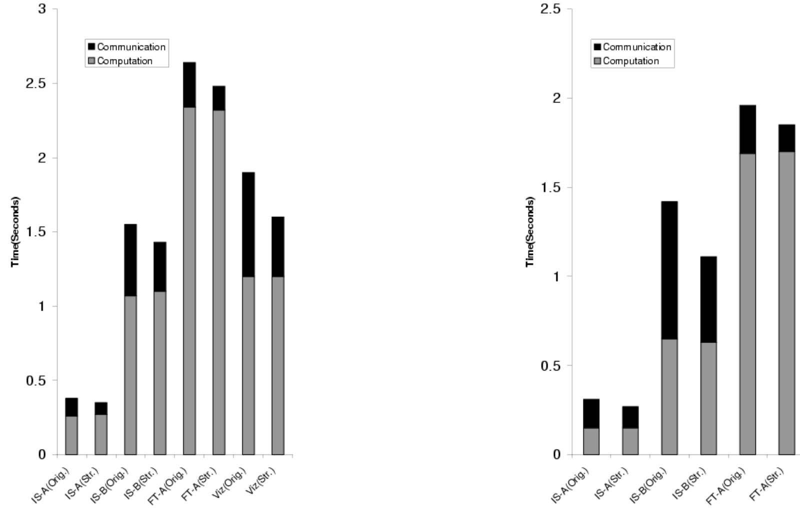


Figure 4.13: Application Results (8 processes, UP mode)

Figure 4.14: Application Results (16 processes, SMP mode)

because compared with other applications, they are more bandwidth-bound. We have also used a visualization application. We show performance numbers 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 multi-rail design results in significant reduction in communication time for all 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 see

that multi-rail design brings significant performance improvement to bandwidth-bound applications.

4.4 Summary

In this chapter, we have presented an in-depth study for designing high performance multi-rail InfiniBand clusters for MPI-1 communication semantics. We have discussed various ways of setting up multi-rail networks with InfiniBand and proposed a unified MPI-1 design that can support all these approaches. By taking advantage of RDMA operations in InfiniBand and integrating the multi-rail design with MPI communication protocols, our design supports multi-rail networks with very low overhead. Our performance results show that the multi-rail MPI can significantly improve MPI communication performance. With a two-rail InfiniBand network, we have achieved almost twice the bandwidth and half the latency for large messages compared with the original MPI. The multi-rail MPI design also significantly reduced the communication time as well as the execution time for bandwidth-bound applications.

CHAPTER 5

SUPPORTING ONE-SIDED COMMUNICATION WITH MULTI-RAIL INFINIBAND CLUSTERS

In the previous chapter, we designed an MPI over InfiniBand for supporting multi-rail clusters. However, the design challenges and performance evaluation was primarily focused for MPI two-sided communication. Compared to MPI-1, MPI-2 is the next generation MPI standard with one-sided operations (such as MPI_Put and MPI_Get). In this work, we propose a unified MPI-2 design with different configurations of multi-rail networks (*multiple ports, multiple HCAs* and *combinations*) for one-sided communication. We present various issues associated with one-sided communication (*multiple synchronization messages, scheduling of RDMA (Read, Write) operations, scheduling policies, ordering relaxation*) and discuss their implications on our design. We also implement our design and evaluate it with micro-benchmarks for one-sided communication. Our performance results show that multi-rail networks can significantly improve MPI-2 one-sided communication performance. With a two-rail InfiniBand cluster, we can achieve almost twice the MPI_Put bandwidth and half the MPI_Put latency for large messages compared to the single-rail MPI-2 implementation.

The rest of the chapter is organized as follows. In section 5.1, we present the design for the communications layer for one-sided communication with InfiniBand. In section 5.2, we present the detailed design issues with supporting one-sided communication over multi-rail InfiniBand clusters. In section 5.3, we evaluate the performance of one-sided benchmarks with our implementation. Finally, in section 5.4, we discuss the contributions and present the summary of the work.

5.1 Communications Layer for One-sided Communication with Multi-Rail Clusters

The basic architecture of our design to support multi-rail networks for MPI-2 one-sided communication is shown in Figure 5.1. In the figure, we can see that besides the MPI-2, Direct One-Sided layer and InfiniBand layer, our design consists of an intermediate layer, *Multi-rail Layer*.

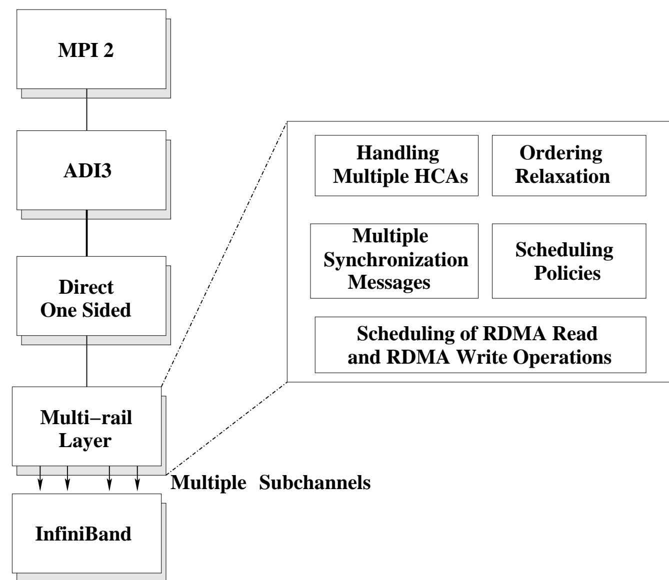


Figure 5.1: Basic Architecture

This layer takes the responsibility of scheduling messages on the available subchannels. Besides this, it takes care of the correctness issues like *Handling Multiple HCAs* and *Multiple Synchronization Messages* and efficiency issues like *Scheduling Policies*, *Ordering Relaxation* and *Scheduling of RDMA Read and RDMA Write Operations*.

In the previous chapter, we focused on virtual channel abstraction to unify different multi-rail configurations (*multiple ports, multiple adapters and combinations*). We incorporate a similar design to unify multi-rail configurations for MPI-2 in this chapter. At the channel level, we maintain a set of data structures to coordinate between different subchannels.

5.2 Detailed Design Issues

In this section, we discuss the design challenges involved for multi-rail MPI-2 design associated at the multi-rail Layer.

5.2.1 Multiple Synchronization Messages

In order to initiate the one-sided communication, the origin process calls *start* to open a window. The target process *posts* the buffers for the window. Once the one-sided communication is done, a synchronization message needs to be sent to the target process.

The receipt of synchronization message guarantees the data transfer of previously issued RMA operations. However, when multiple subchannels are used, data transfer on one subchannel might not have finished even though other subchannels would have received the synchronization message. Hence, we need to issue synchronization messages on each subchannel. Note that when the load on subchannels

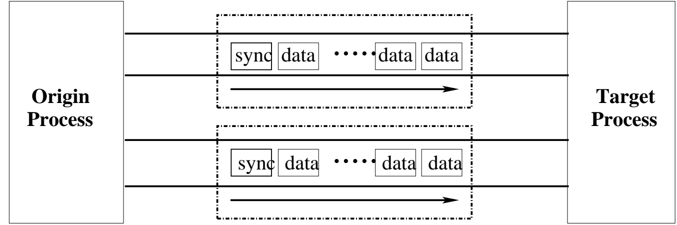


Figure 5.2: Multiple Synchronization Messages

is balanced, the transfer of synchronization messages along multiple subchannels takes place in parallel, incurring negligible overhead.

5.2.2 Scheduling of RDMA Read and RDMA Write Operations

In MPI-1, usually the two-sided communication uses either RDMA Write or RDMA Read for data transfer in InfiniBand. For many MPI-2 applications, in one-sided communication, the MPI_Put and MPI_Get operations are implemented using RDMA Write and RDMA Read respectively. Since RDMA Read and RDMA Write utilize bandwidth in different directions, it is important to schedule them independently with respect to each other's load on different subchannels.

In order to achieve this, we propose a load based fragmentation policy discussed in the next section, which maintains independent queues of MPI_Put and MPI_Get operations issued in an *epoch*. Trivially, this policy would fragment the messages equally on all subchannels in the presence of only one kind of a one-sided operation. In presence of a combination of one-sided operations, each having the same size, this policy would fall back to equal fragmentation.

5.2.3 Scheduling Policies Classification based on Message Size

In this chapter, we classify the policies used for scheduling based at different layers. As proposed in [23], we use *reordering* and *no reordering* policies at the CH3' (Direct One-Sided) Layer. At the multi-rail layer, we do a classification of the policies based on the message size. We employ the following policies:

- *Round Robin*
- *Load Balanced Fragmentation*

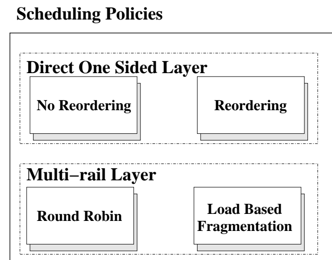


Figure 5.3: Scheduling Policies at different layers for one-sided communication

For small messages, we employ round robin policy. In this policy, the completion message is sent using one of the available subchannels in a round robin fashion. Fragmentation incurs overhead of posting descriptors on multiple subchannels, which is significant for small messages. Hence, we employ a *switchover threshold*, messages of size less than this threshold are scheduled in a round robin fashion. For large messages, we primarily use Load Balanced Fragmentation policy. In this policy, we divide the message in chunks and schedule them, so that

the load on all subchannels is balanced. This policy leads to optimal utilization of all subchannels for large messages. In case of one-sided communication, the switchover threshold can be reduced significantly in comparison to two-sided communication, due to the absence of rendezvous protocol. As a result, medium size messages can also benefit from fragmentation. We present the actual benefits in the performance evaluation section.

5.2.4 Absence of Rendezvous Protocol

For regular two-sided communication, small messages typically use *eager* protocol and large messages use *rendezvous* protocol [34]. For medium size messages, protocol selection depends upon the overhead of control messages for rendezvous protocol and credits available for *eager* protocol communication. One-Sided communication does not require a handshake for data transfer between the origin and target processes. The absence of handshake overhead allows the medium size messages also to take the advantage of the multi-rail configurations.

5.2.5 Ordering Relaxation

Regular two-sided communication requires messages to be processed in order by the receiver side. Message(s) scheduled on multi-rail networks may be received out of order by the receiver side. To maintain correctness, the receiver side needs to queue the *out-of-order* messages and poll periodically on them. For large scale clusters, this bookkeeping may incur overhead, potentially reducing the benefits of multi-rail networks.

One-Sided communication imposes no ordering requirements for messages within an *epoch*, by the definition from the semantics. As a result, the one-sided approach

does not need to maintain ordering at the receiver side. We simplify our design by incorporating this fact, reducing the overhead of bookkeeping at the receiver side.

5.3 Performance Benefits of Multi-Rail Design for One-Sided Communication

To evaluate the performance benefits of our multi-rail MPI-2 design, we compare it with our original MVAPICH2 design [42], which can only use only one-port of a NIC. In the multi-rail design, we use *load balanced fragmentation* for large messages and *round robin* scheme for small messages. In the next couple of sections, we present the performance evaluation of multi-rail design with MPI-2 one-sided benchmarks. We begin with a description of our experimental testbed.

5.3.1 Experimental TestBed

We evaluated our implementation with multiple HCAs on 32-bit systems (referred to IA32 cluster from here onwards) comprising of independent PCI-X buses, and on 64-bit systems (referred to as EM64T cluster from here onwards) comprising of PCI-Express bus and multiple ports per adapter. Our experimental testbed comprises of two clusters.

IA32 Cluster with Multiple HCAs:

This cluster consists of two 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 Dual-Port 4x HCAs from Mellanox. 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 contention, the two HCAs are connected to separate PCI-X buses connected to different I/O bridges. The kernel version we used is Linux 2.4.22smp. The IBGD version is 1.6.1 and HCA firmware version is 3.3.2. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 2 GB of PC2100 DDR-SDRAM.

EM64T Cluster with Multiple ports:

This cluster consists of two EM64T nodes having 8x PCI Express slots. Each node has two Intel Xeon CPUs running at 3.4 GHz processors, 512 KB L2 cache and 1 GB of main memory. This cluster uses III Generation MT25208 4x Dual Port HCAs from Mellanox. A combined unidirectional bandwidth of 8x can be used, when both ports are used for communication. The kernel version we used is Linux 2.4.21-15.EL. The IBGD version is 1.6.1 and HCA firmware version is 4.6.2. The Front Side Bus (FSB) of each node runs at 800MHz.

5.3.2 Micro-benchmark Evaluation for MPI_Put Operation

In Figures 5.4 and 5.6, we present the results for MPI_Put bandwidth, bidirectional bandwidth and latency respectively for the IA32 cluster with multiple HCAs. We show the results for EM64T with two-ports on PCI-Express in Figures 5.5 and 5.7.

In Figure 5.4, we observe that for small messages (less than or equal to 1KBytes) , both multi-rail design and the original implementation perform comparably. For large messages, multi-rail design outperforms the original implementation considerably. With multi-rail design, we can achieve a maximum peak unidirectional

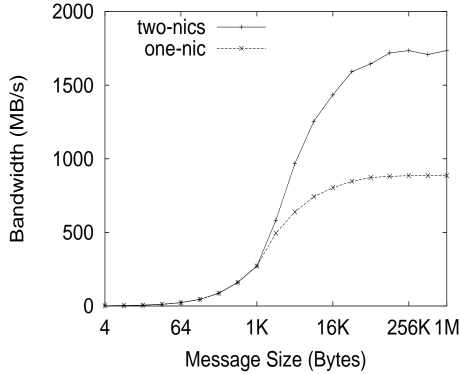


Figure 5.4: MPI_Put Bandwidth on the IA32 Cluster

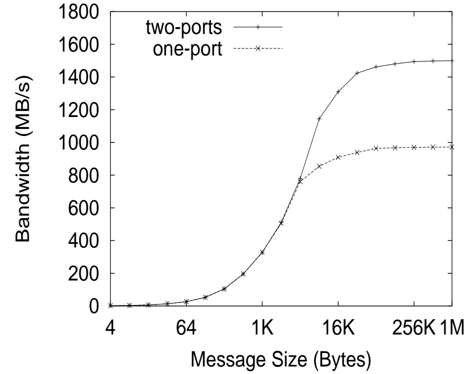


Figure 5.5: MPI_Put Bandwidth on the EM64T Cluster

MPI_Put bandwidth of 1750 MB/s in comparison to 880 MB/s for our original implementation. We also notice, that due to the absence of rendezvous protocol, medium size messages (2KB - 16KB), can take advantage of load balanced fragmentation policy for multi-rail design.

We observe a similar trend for dual-port on EM64T in Figure 5.5. For messages of size greater than 8KBytes, we use fragmentation policy. We can achieve a peak bandwidth of 1500 MB/s using multi-rail design, in comparison to 971 MB/s for the original implementation capable of using only one-port of a NIC.

In figures 5.6 and 5.7, we compare the performance of MPI_Put bidirectional bandwidth for IA32 cluster and EM64T cluster, respectively. For IA32 cluster, due to the bottleneck of PCI-X, we can achieve only 941 MB/s for original implementation. However, using multi-rail design we can achieve a peak bidirectional bandwidth of 1810 MB/s.

On IA32, we observe a slight drop in bandwidth at 16Kbytes. This occurs due to significant increase in bidirectional bandwidth in comparison with unidirectional

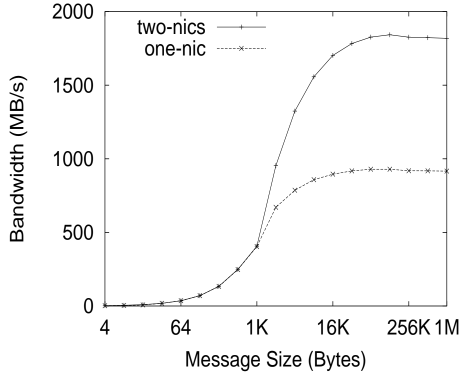


Figure 5.6: MPI_Put Bidirectional Bandwidth on the IA32 Cluster

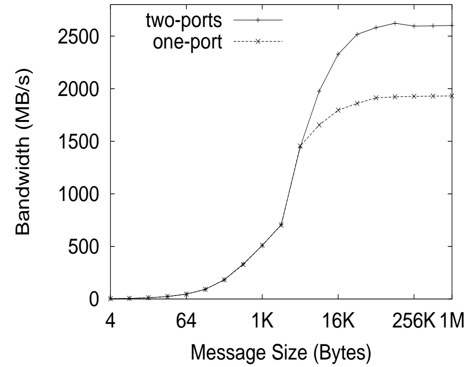


Figure 5.7: MPI_Put Bidirectional Bandwidth on the EM64T Cluster

bandwidth, and usage of same switchover threshold for them. For EM64T cluster, we can achieve a peak bidirectional bandwidth of 2620 MB/s with two-ports in comparison to 1910 MB/s using the original implementation.

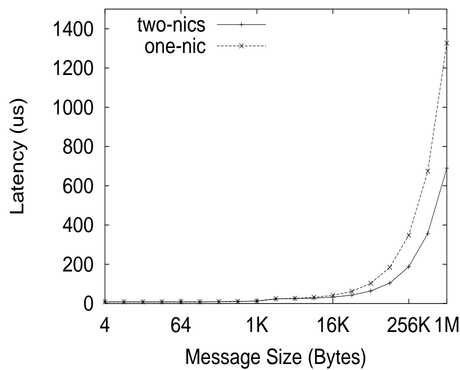


Figure 5.8: MPI_Put Latency on the IA32 Cluster

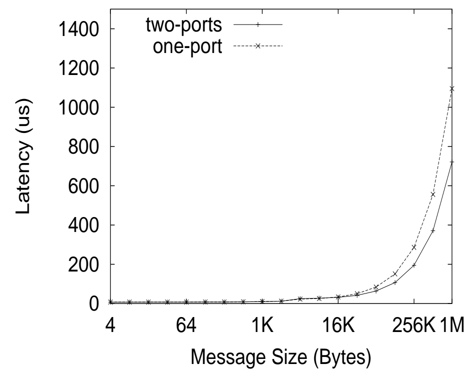


Figure 5.9: MPI_Put Latency on the EM64T Cluster

In figures 5.8 and 5.9, we present the results for MPI_Put latency for IA32 and EM64T cluster, respectively. We observe that we perform almost similar with the

original implementation for small messages. For large messages, we can improve the latency by 46% for IA32 cluster and 32% for EM64T cluster by using the multi-rail design.

5.3.3 Micro-Benchmark Performance Evaluation for MPI Get Operation

Figure 5.10 shows the bandwidth for MPI_Get operation for the IA32 cluster. We observe similar trends in performance as MPI_Put operation. Using the multi-rail design, we can achieve a peak bandwidth of 1705 MB/s, compared to 881 MB/s for the original implementation. For the EM64T cluster, we can achieve a peak bandwidth of 1494 MB/s, compared to 969 MB/s for the original implementation.

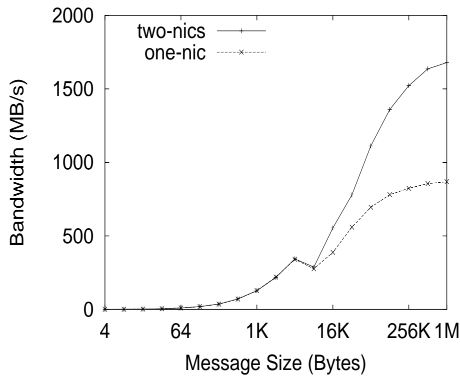


Figure 5.10: MPI_Get Bandwidth on the IA32 Cluster

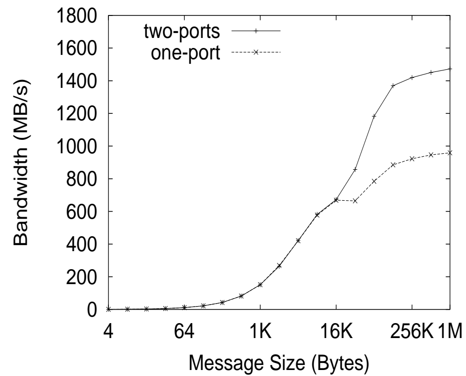


Figure 5.11: MPI_Get Bandwidth on the EM64T Cluster

In figures 5.12 and 5.13, we present the results for MPI_Get latency for IA32 and EM64T cluster, respectively. We observe that we perform almost similar with the original implementation for small messages. For large messages, we can improve

the latency by 45% for IA32 cluster and 33% for EM64T cluster by using multi-rail design.

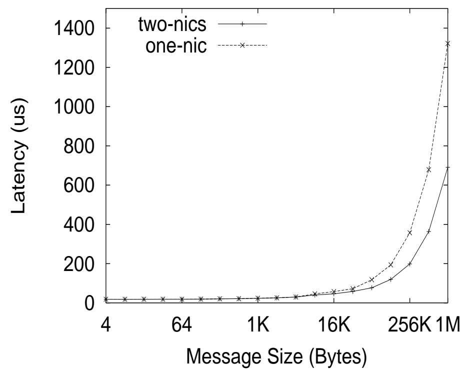


Figure 5.12: MPI_Get Latency on the IA32 Cluster

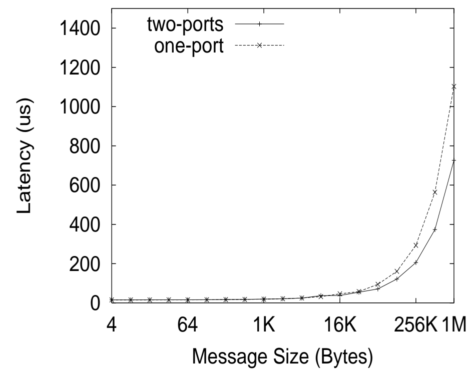


Figure 5.13: MPI_Get Latency on the EM64T Cluster

5.3.4 MPI-2 One-Sided Interleaving Test

To show the impact of re-ordering of one-sided communication, we use a throughput test which involves two processes. The first process starts a window access epoch and then issues 16 MPI_Put and 16 MPI_Get operations of the same size. The second process just starts an exposure epoch. The same sequence of operations are repeated for several iterations and we measure the maximum throughput we can achieve (in terms of Million Bytes/sec). In our previous work, we have explained this test in further detail [23].

Fig. 5.14 shows the case where CH3' (Direct One-Sided) layer issues all one-sided operations in order. For MPI_Put, the data mainly flows from the origin process to the target process, while for MPI_Get, the data mainly flows from the

target process to origin process. So only one direction of the network link is fully used at one time. We can use the network bandwidth more efficiently if we re-order the one-sided operations, as shown in Fig.5.15. Here we interleave the put and get operations so that we can almost achieve bi-directional bandwidth provided by the link.

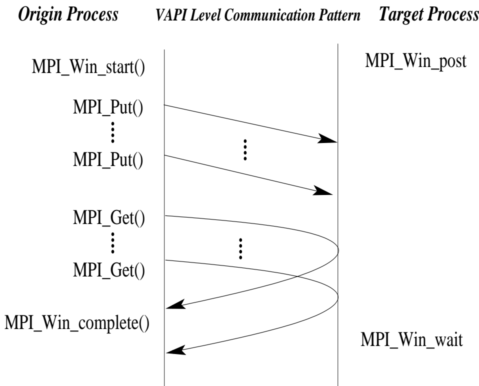


Figure 5.14: Ordered issue of one-sided operations

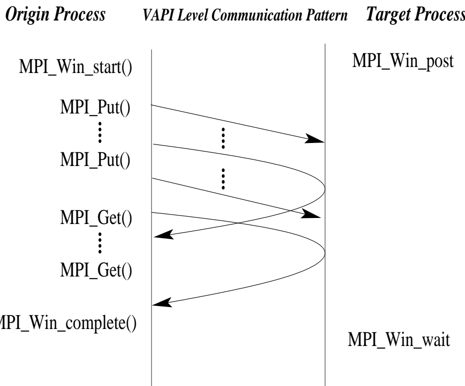


Figure 5.15: Re-ordered issue of one-sided operations

Figure 5.16 shows the performance achieved by a combination of policies at the CH3' layer and multi-rail layer. At the multi-rail layer we use load balanced fragmentation policy. At the CH3' layer, we compare impact of reordering with no reordering, when combined with the multi-rail policy specified above.

For IA32 cluster using two-nics, we can achieve almost 1703 MB/s without reordering, which is close to the multi-rail peak unidirectional bandwidth. With single-rail implementation, we can achieve a peak bandwidth of 880 MB/s without reordering. We notice, that with reordering for two-nics, we can almost achieve

1800 MB/s, almost the peak bidirectional bandwidth with two-nics. With single-rail implementation, due to the limitation of PCI-X, we can achieve only 907 MB/s.

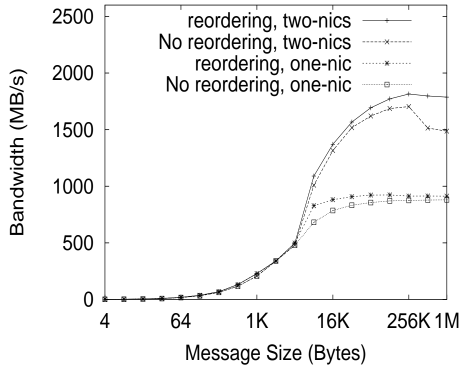


Figure 5.16: Interleaved throughput on the IA32 Cluster

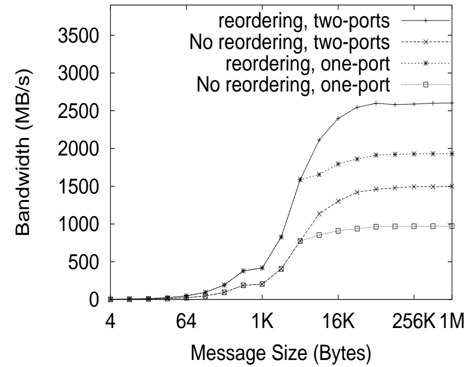


Figure 5.17: Interleaved throughput on the EM64T Cluster

In figure 5.17, we evaluate the performance of CH3' layer reordering, compared to the no reordering policy for the EM64T cluster. We use the load balanced fragmentation at the multi-rail layer. Using two-ports and reordering, we can achieve 2604 MB/s, which is almost the peak bidirectional bandwidth available with two-ports. It is interesting to notice, that reordering with single-rail implementation outperforms the combination of no reordering with multi-rail implementation. We attribute it to the fact that, PCI-Express can achieve 8x bidirectional bandwidth with one-port. However, due to the limitations of PCI-Express, we cannot achieve a combined 8x unidirectional bandwidth using two-ports. Using reordering with single-rail implementation, we can achieve 1900 MB/s. However we can only achieve a peak bandwidth of 1474 MB/s using multi-rail implementation with no reordering. With no reordering for single-rail implementation, we can

achieve 962 MB/s, which is close to the unidirectional bandwidth available with the single-rail implementation.

5.4 Summary

In this chapter, we have presented the challenges *(Multiple synchronization messages, handling multiple HCAs, scheduling policies, ordering relaxation)* associated with designing MPI-2 one-sided communication over multi-rail InfiniBand networks. We have implemented our design and presented the performance evaluation for micro-benchmarks. We have observed that multi-rail InfiniBand clusters can significantly improve the performance for one-sided communication. Using a two rail cluster, we have achieved almost doubled the throughput and reduced the latency to half with *MPI_Put* and *MPI_Get* operations for large messages. We have also observed that reordering policy can significantly improve the performance for communication patterns with a mix of one-sided operations.

CHAPTER 6

IMPROVING PERFORMANCE WITH IBM 12X INFINIBAND ARCHITECTURE

In the previous chapters, we have studied the design for supporting multiple communication semantics with multi-rail InfiniBand Clusters. The primary focus has been to support multiple adapters and multiple ports. As discussed in the background section, 12x adapters with support for multiple send/receiv engines has been introduced. The designs proposed in the previous chapters are not sufficient in extracting the potential performance of these adapters.

In this chapter, we propose a unified MPI design for taking advantage of multiple send/receive engines on a port, multiple ports and HCAs. We study the impact of various communication scheduling policies *(binding, striping and round robin)* and discuss the limitations of these individual policies for different communication patterns, in context of IBM 12x InfiniBand Adapter. To overcome this limitation, we present a new policy, EPC (*Enhanced point-to-point and collective*), which incorporates different kinds of communication patterns *point-to-point (blocking, non-blocking) and collective communication,* for data transfer. To enable this differentiation, we design a *communication marker* and discuss the need to integrate it with the ADI layer for obtaining the optimal performance. We

implement our design and evaluate it with micro-benchmarks, collective communication and NAS parallel benchmarks. Using EPC on a 12x InfiniBand cluster, we can significantly improve the execution times of micro-benchmarks, collective communication and MPI application kernels.

The rest of the chapter is organized as follows. In section 6.1, we present the overall design for supporting MPI with 12x InfiniBand Architecture. In section 6.2, we present the need for scheduling policies for different communication patterns. In section 6.3, we present the evaluation of our implementation. We conclude and present the summary in section 6.4.

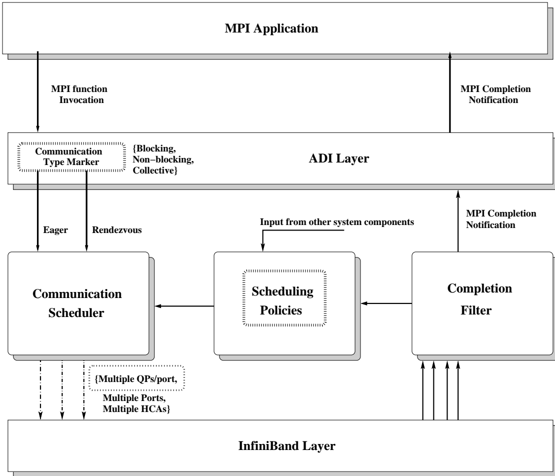


Figure 6.1: Overall MPI Design for IBM 12x InfiniBand Architecture

6.1 Overall MPI Design for 12x InfiniBand Architecture

Figure 6.1 represents our overall design. Our previous design presented in [33, 59] supports using multiple ports and multiple HCAs. In our new design

presented here, we enhance it by adding support for utilizing multiple send/receive engines per port. In addition, in our enhanced design, we present a *communication marker*, which differentiates between communication patterns, to obtain optimal performance for point-to-point and collective communication. These enhancements are shown with dotted boxes in Figure 6.1. A detailed description is also presented in our publication [55].

6.2 Discussion of Scheduling Policies for different Communication Patterns

In this section, we present the discussion on scheduling policies. Even though, in our previous work, we have presented an initial discussion on scheduling policies, we discuss the limitations of the previously proposed scheduling policies for utilizing multiple send/recv engines in an efficient manner. We begin with a discussion on point-to-point communication.

6.2.1 Point-to-Point Communication

Point-to-point communication can be classified as *blocking* and *non-blocking* type of communication. In the blocking communication, only one message is outstanding in communication between a pair of processes. Round Robin policy uses the available QPs one-by-one in a round robin fashion [33, 59]. Using round robin policy may lead to under-utilization of the available send and receive DMA engines for such kind of communication. *Striping* divides the messages amongst available queue pairs providing a much better utilization of available DMA engines. Similarly, for *non-blocking* communication, *striping* can provide benefits by exploiting parallelism in send and receive DMA engines.

However, a large percentage of MPI applications mainly use medium size messages for data transfer. In our previous work [33, 59], our design and evaluation comprised mostly of two queue pairs (one queue pair per port), hence the impact of striping on the performance of medium size messages is negligible. However, using multiple send/receive engines per port requires usage of multiple queue pairs per port. As the number of queue pairs increase, the cost of assembly and dis-assembly due to striping becomes significant. This cost is mainly due to posting descriptors for each stripe, and acknowledgment overhead of the reliable connection transport service of InfiniBand. Hence, using round robin policy for communication may outperform the striping policy.

From the above discussion, the need to differentiate between point-to-point communication patterns is clear. We incorporate this using a *communication marker* presented in the later part of the section.

6.2.2 Collective Communication

Collective communication primitives based on point-to-point use MPI_Sendrecv primitive for various steps in the algorithm. Each MPI_Sendrecv call can further be divided in one function call of MPI_Isend and MPI_Irecv each for the partners. For blocking MPI Collectives, each step in the algorithm is completed before executing the next step. As described in the previous section, this is a non-blocking form of communication, and round robin policy would be used, as concluded previously. However, only one outstanding non-blocking call is available for each send/receive engine, which may lead to insufficient usage of available send DMA engines. Thus,

we clearly need to differentiate among the non-blocking calls received from point-to-point communication and collective communication.

From the above discussion, we can conclude that a single scheduling policy is not sufficient for data transfer with different patterns. Some policies benefit blocking communication, while other benefit the non-blocking communication. In addition, for the non-blocking communication, the usage by collective communication can further complicate the scheduling policy decision. To resolve the above conflicts of policy selection, we present a policy, Enhanced point-to-point and collective (EPC), which falls back to optimal policies for respective communication patterns. For non-blocking communication, it uses *round robin*, for blocking communication, it uses *striping*. For collective communication, even though we have non-blocking calls, it falls back to *striping*. The efficiency of this policy is dependent upon the ADI layer to be able to differentiate between such communication patterns. Next, we present such a module, called *communication marker* module, which resides in the ADI layer and takes advantage of ADI layer data structures and parameters for differentiating amongst communication patterns.

6.2.3 Communication Marker

The communication marker module resides in the ADI layer of our design. The main purpose of this module is to be able to differentiate amongst different communication patterns invoked by the MPI Application. In essence, it differentiates between:

- Point-to-point
  - Blocking

  - Non-blocking

- Collective

Since our design is based on MPICH, this differentiation at ADI layer is possible. For collective communication, a separate tag is used, which can be used to differentiate an ADI function call from point-to-point communication. In case of tag conflicts, the differentiation is not accurate, however, it does not lead to any performance degradation. In addition, the ADI layer decides the communication protocol eager/rendezvous depending upon the message size. We have used a rendezvous threshold of 16KBytes in performance evaluation. This value is also used as the *striping threshold*, the messages of size equal and above are striped on all available queue pairs equally.

6.3 Performance Evaluation with MPI over IBM 12x InfiniBand Architecture

In this section, we present performance evaluation of IBM 12x HCAs using MPI Benchmarks. We compare the performance of our enhanced design with MVAPICH release version (referred to as original from here on). The 1QP/port case is referred to as the original version of MVAPICH. We show the performance results for simple micro-benchmarks, latency, bandwidth and bi-directional bandwidth. This is followed by performance evaluation on NAS Parallel Benchmarks [8].

6.3.1 Experimental Testbed

Our experimental testbed consists of an InfiniBand cluster with IBM nodes built with Power6 processor. The cluster is connected using IBM 12x Dual-Port

HCAs. Each node in the cluster comprises 4 processors, shared L2 and L3 caches along-with 32 GB DDR2 533MHz main memory. Each node has multiple GX+ slots, which run at a speed of 950 MHz and CPU speed of 2.4 GHz. We have used 2.6.16 linux kernel and InfiniBand drivers from OpenIB-Gen2, revision 6713. For our experimentation, we have used only one GX+ bus, one HCA and one port of an HCA. The objective is to evaluate the performance of multiple send/receive engines on one HCA. However, the experimentation can definitely be extended to usage of multiple ports, HCAs and combinations.

6.3.2 Performance Evaluation with Micro-Benchmarks

In Figure 6.2, we present the results for the latency test. For small messages, it is not beneficial to stripe the message across multiple queue pairs as the startup time is dominant. Hence, even with increasing number of queue pairs, we use only one of the QPs for communication. The objective is to see the performance degradation from our design compared to the original case. From the figure, it is clear that our design adds negligible overhead compared to the original case.

Figure 6.3 shows the results for large message latency, comparing a set of parameters: scheduling policy and number of queue pairs used per port. The objective is to understand the efficiency of the communication marker for differentiating between communication types. Hence we compare the performance of EPC and policies proposed in the previous work. We notice that using 4QPs/port, EPC and striping perform comparably. Both Binding and Round Robin are not able to take advantage of multiple queue pairs, since they use only one queue pair for an MPI message. An improvement of 33% is observed using EPC and striping.

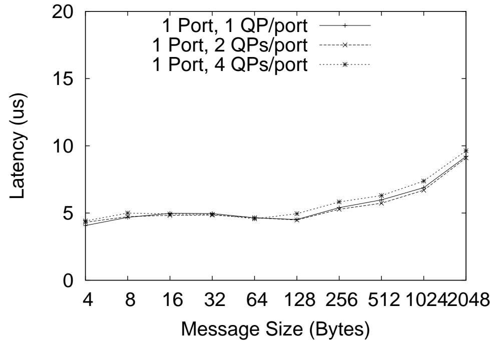


Figure 6.2: MPI Latency For Small Messages

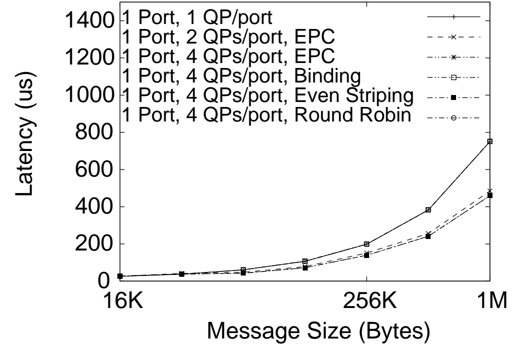


Figure 6.3: MPI Latency For Large Messages

Figure 6.4 and 6.5 compare the performance of EPC and round robin policy with the original case. Since messages are still on a smaller range, we do not use striping. Comparing 2QPs with 4QPs, we observe that performance gains are observed after 1K. For very small messages (less than 1K), the startup time limits the usage of multiple queue pairs efficiently. However, from 1K-8K message range, as the transfer time increases, 4QPs show improvement in performance. The performance is similar to the round robin policy. For the bi-directional bandwidth case, we notice that increasing number of queue pairs from 2-4 does not help. Since we use eager protocol for small messages, copy based approach is used. In the bi-directional bandwidth test, the process needs to copy the data out to the peer and also copy the data sent by the peer. We notice that the memory copy bandwidth limits the improvement in performance.

Figures 6.6 and 6.7, show the performance of uni-directional and bi-directional bandwidth tests for large messages. We compare the performance of EPC with the originally proposed even striping [33, 59]. Using both policies we are able

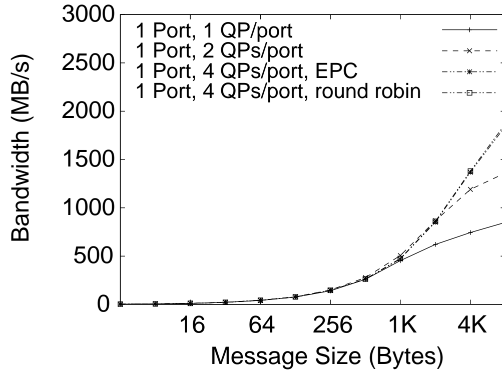


Figure 6.4: Impact of Scheduling Policies on Small Message Unidirectional Bandwidth

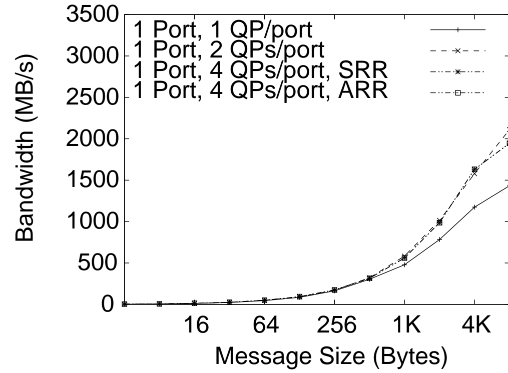


Figure 6.5: Impact of Scheduling Policies on Small Message Bidirectional Bandwidth

to achieve, 2745 MB/s and 5263 MB/s results respectively for the above tests in comparison to 1661 MB/s and 3079 MB/s using the original implementation. However, even striping performs much worse than EPC for medium size messages (16K - 64K). This can be attributed to the fact that dividing the data into multiple chunks leads to inefficient use of send engines, as they do not have enough data to pipeline, posting of descriptors for each send engine and receipt of multiple acknowledgments. For very large messages, the data transfer time is reasonably high, and as a result, the performance graphs converge.

Figure 6.8 shows the performance of MPI_Alltoall using our enhanced design. We use 2x4 configuration for performance evaluation, where two nodes and four processes per node are used for communication. However, for MPI_Alltoall, even for medium range of messages, we can see an improvement, due to efficient utilization of available send and receive DMA engines in comparison to single-rail implementation. Hence, differentiation at the ADI layer between non-blocking

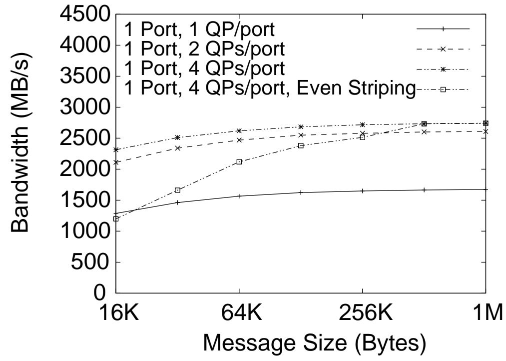


Figure 6.6: Large Message Unidirectional Bandwidth

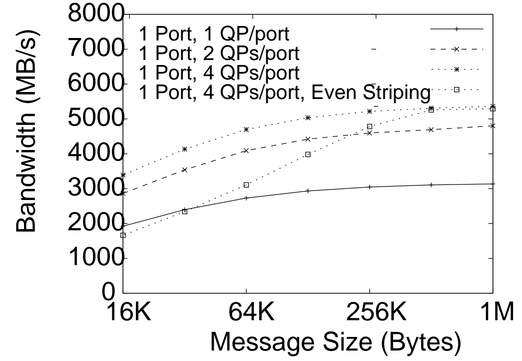


Figure 6.7: Large Message Bidirectional Bandwidth

communication and collective communication significantly helps the performance of collective operations.

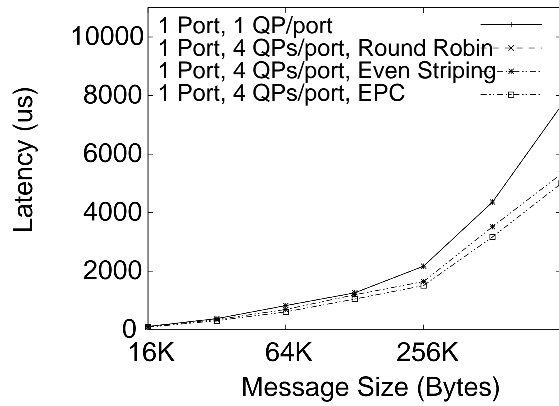


Figure 6.8: Alltoall, Pallas Benchmark Suite, 2x4 Configuration

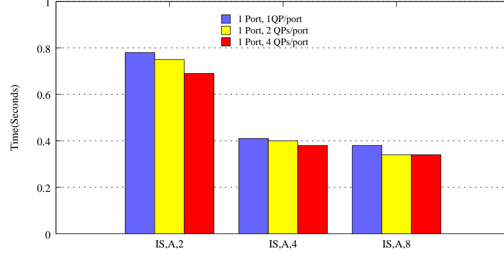


Figure 6.9: Integer Sort, NAS Parallel Benchmarks, Class A

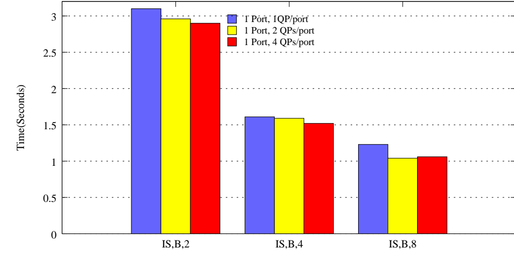


Figure 6.10: Integer Sort, NAS Parallel Benchmarks, Class B

6.3.3 Performance Evaluation with NAS Parallel Benchmarks

Figures 6.9 and 6.10 show the results for Integer Sort, Class A and Class B respectively. We compare the performance for 2 (2x1), 4 (2x2) and 8 (2x4) processes respectively. Using two processes on Class A and B, the execution time improves by 13% and 9% respectively with 4 QPs/port. We use only EPC policy for comparison, since it performs equal or better than previously proposed policies, as shown by results from micro-benchmarks. For 4 processes, the execution time improves by 8% and 7% respectively. Since we use shared-memory communication for processes on same node, the percentage of network communication decreases with increasing number of processes and the performance benefits follow a similar trend. However, we do not see any performance degradation using our enhanced design.

Figures 6.11 and 6.12 show the results for Fourier Transform, Class A and Class B, respectively. We see around 5%-7% improvement with increasing number of processes. Although, not included in the paper, we have not seen performance degradation using other NAS Parallel Benchmarks.

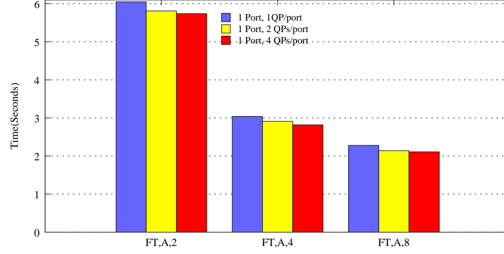


Figure 6.11: Fourier Transform, NAS Parallel Benchmarks, Class A

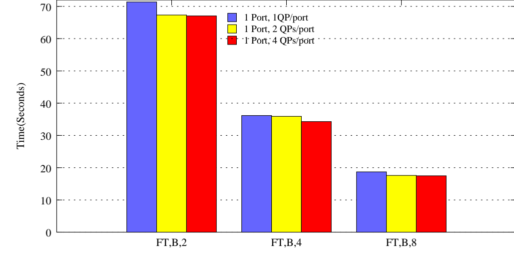


Figure 6.12: Fourier Transform, NAS Parallel Benchmarks, Class B

6.4 Summary

In this chapter, we have presented an MPI design for IBM 12x InfiniBand architecture comprising of multiple send/receive DMA engines. We have studied the impact of various communication scheduling policies *(binding, striping, and round robin)*, and presented a new policy, EPC (*Enhanced point-to-point and collective*), which incorporates different kinds of communication patterns; *point-to-point blocking, non-blocking and collective communication,* for data transfer. We have discussed the need to strongly integrate our design with the ADI layer to obtain optimal performance. Our performance results show that 12x HCAs can significantly improve MPI communication performance. Using EPC on a 12x InfiniBand cluster with one HCA and one port, we can improve the performance by 41% with ping-pong latency test and 63-65% with the unidirectional and bi-directional throughput tests compared with the default single-rail MPI implementation. We have concluded that none of the previously proposed policies alone provide optimal performance in these communication patterns. Using NAS Parallel Benchmarks,

we see an improvement of 7-13% in execution time along with a signification improvement in collective communication using Pallas benchmark suite.

CHAPTER 7

HOT-SPOT AVOIDANCE WITH MULTI-PATHING USING LMC MECHANISM

Large scale InfiniBand clusters are becoming increasingly popular, as reflected by the TOP 500 [6] Supercomputer rankings. At the same time, *fat tree* [31] has become a popular interconnection topology for these clusters, since it allows multiple paths to be available in between a pair of nodes. However, even with fat tree, hot-spots may occur in the network depending upon the route configuration between end nodes and communication patterns in the application. To make the matters worse, the deterministic routing nature of InfiniBand limits the application from an effective use of multiple paths transparently and avoid the hot-spots in the network. Simulation based studies for switches and adapters to implement congestion control have been proposed in the literature [62, 20, 52]. However, these studies have focused on providing congestion control for the communication path, and not on utilizing multiple paths in the network for hot-spot avoidance. Literature proposed for Ethernet is not completely relevant for InfiniBand, since these InfiniBand supports link level flow control [18, 61, 27]. Other studies have proposed adaptive routing mechanisms for InfiniBand, however these

mechanisms are not available with the current generation of InfiniBand adapters and switches [40, 37, 51, 49, 29, 35, 50].

In this chapter, we present the design for an MPI functionality, Hot-spot Avoidance with MVAPICH (HSAM), which provides hot-spot avoidance for different communication patterns, without apriori knowledge of the pattern. We leverage LID Mask Count (LMC) mechanism of InfiniBand to create multiple paths in the network, and study its efficiency in creation of contention free routes. We also present the design issues (scheduling policies, selecting number of paths, scalability aspects) associated with our MPI functionality. We implement our design and evaluate it with collective communication and MPI applications. Our experimental evaluation shows that HSAM is able to benefit the performance of collective communication primitives and MPI application kernels significantly.

The rest of the chapter is organized as follows. In section 7.1, we present the limitations of existing designs in providing network hot-spot avoidance. We present the design issues of our work, including the proposal to adaptive striping policy, which uses path bandwidth estimation for hot-spot avoidance. In section 7.2, we present the performance evaluation of our work In section 7.3, we conclude and summarize our contributions.

7.1 Limitations of Existing Designs

In the designs presented in previous chapters, we presented an initial framework using multi-rail networks and presented different scheduling policies for their efficient utilization. We leverage this framework for designing hot-spot avoidance functionality. However, the existing framework suffers from following limitations:

- The existing design assumes the uniformity in the network bandwidth for different paths. It also does not provide a mechanism for estimation of path bandwidth.

- A key functionality missing in the existing framework is to leverage the LMC mechanism for hot-spot avoidance. Using the subnet manager to configure disjoint paths and their efficient usage is a major functionality, which is added to the existing framework.

- The existing framework assumes the presence of one network path. However, it is possible that some of the network paths are being heavily utilized, and some are being under utilized. Utilizing very few of these paths may not significantly help to avoid hot-spots. However, utilization of all the existing paths has practical implications due to startup costs, and accuracy in the estimation of path bandwidth. We study this issue in detail in our design and evaluation.

- Increasing number of paths leads to increased memory utilization. In the existing framework, we did not address the memory scalability issue with increasing number of queue pairs. However, with increasing number of queue pairs, this issue needs to be addressed. We address this problem using InfiniBand's shared receive queue mechanism.

We address each of the above issues in this work. We call our enhanced design, HSAM (Hot-Spot Avoidance MVAPICH). In the section 4.1, we presented an initial design for leveraging multi-rail clusters. We use this framework as much as possible for providing hot-spot avoidance. Specifically, we enhance this framework with

dynamic scheduling policies, which adjust themselves based on input from other components of the system. We also enhance the design to support usage of LMC mechanism to leverage the presence of multiple paths.

7.1.1 Leveraging Multiple Paths Using LMC

Depending upon the communication pattern, presence of other jobs in the network, it is possible that some paths become hot-spots, while other paths are left under-utilized. An important mechanism to efficiently use the available paths is by changing the routing table of each switch block. Using this mechanism, contention free paths can be created for a particular communication pattern. The subnet manager allows a user to input its own routing table for different switches, which can be used for communication. However, this mechanism suffers from the fact that the optimization can be done only for a single communication pattern. At the same time, presence of other jobs in the network, exact scheduling of each MPI task can complicate the generation of user-assisted routing tables.

To overcome the above limitations, we leverage the LID (Local Identifier) Mask Count (LMC) mechanism of InfiniBand, which allows multiple paths to be created between a pair of nodes. Using an LMC value of x, we can create 2^x paths, with 7 being the maximum value allowed for LMC. We use OpenSM [58], a popular subnet manager for InfiniBand to configure these paths. Using *trace-route* mechanism of InfiniBand, we calculate the exact path (set of switch blocks and ports) taken by each pair of source and destination LID for different values of LMC. We notice that the subnet manager is able to configure paths utilizing different spine blocks in the switch. Hence, LMC mechanism provides us as many contention free paths

as possible. However, efficient utilization of these paths is dependent upon the scheduling policy for data transfer. In the next section, we discuss the scheduling policies used for evaluation.

7.1.2 Adaptive Striping

As we have discussed in section 4.1, 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 paths to their respective link bandwidths. However, this method fails to address the problem of handling hot-spots for different situations mentioned in the previous sections.

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 path between every pair of nodes), it fails to solve the problem for handling varying hot-spots.

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 path 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 paths 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 path 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 the figure 7.1

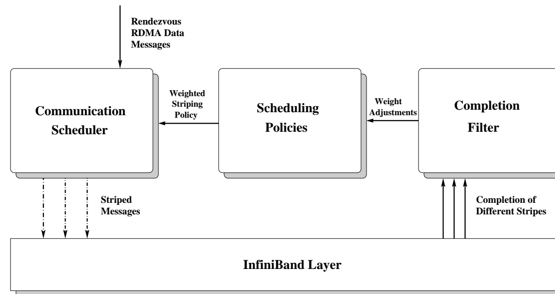


Figure 7.1: Feedback Loop in Adaptive Striping

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 re-calculated 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 path i is W_i, the path bandwidth of path i is BW_i, the message size is S, and the stripe delivering time for path i is t_i, we then have the following:

$$BW_i = \frac{S \cdot \frac{W_i}{W_{total}}}{t_i} = \frac{S \cdot W_i}{t_i \cdot W_{total}} \tag{7.1}$$

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

$$BW_i \propto \frac{W_i}{t_i} \tag{7.2}$$

Therefore, new weight distributions can be done based on Equation 7.2. Suppose W_i' is the new weight for path i, the following can be used to calculate W_i':

$$W_i' = W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in paths} \frac{W_k}{t_k}} \tag{7.3}$$

In Equation 7.3, 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:

$$W_i' = (1 - \alpha) \cdot W_i + \alpha \cdot W_{total} \cdot \frac{\frac{W_i}{t_i}}{\sum_{k \in paths} \frac{W_k}{t_k}} \tag{7.4}$$

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 path. 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 7.4, 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 7.4, 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.

7.1.3 Selecting Number of Paths

In topologies like Fat Tree, there are multiple paths available for communication between every pair of processes, even though there is only one physical port

available at the end node. Using the maximum value of LMC allowed by InfiniBand specification, we can create 128 virtual paths. However, some of the paths may physically overlap with each other. As per the InfiniBand specification connection model, a queue pair/path is needed to use them simultaneously. Once a path is specified for a queue pair, it cannot be changed during the communication (An exception is Automatic Path Migration for InfiniBand, which is beyond the scope of this work). However, there are some practical considerations in leveraging all the paths simultaneously:

- Sending a message stripe through each path requires posting a corresponding descriptor. Hence, this may lead to significant startup overhead with increasing number of paths.

- For each message stripe, a completion is generated on the sender side. With increasing number of paths, more completions need to be handled, which can potentially delay the progress of the application.

- The accuracy of path bandwidth is significantly dependent upon the discovery of the completions, as mentioned in the adaptive striping policy section. With increasing number of paths, the accuracy may vary significantly.

- The memory usage increases with increasing number of paths. We handle this issue in the scalability section later.

Hence, a judicious selection of number of paths is imperative to performance and memory utilization of the MPI library. In the next section, we discuss the memory utilization aspect of our design.

7.1.4 Scalability Aspects of HSAM

In the previous section, we discussed that increasing number of paths leads to more memory utilization. Essentially, the memory utilization per path can be represented as follows:

$$mem_{qp} = mem_{qp-context} + ne_s * mem_{sqe} + ne_r * mem_{rqe} \tag{7.5}$$

where mem_{qp} is the connection memory usage per path, ne_s and ne_r are number of send and receive work queue elements, mem_{sqe} and mem_{rqe} are the sizes of each send queue and receive queue elements respectively. $mem_{qp-context}$ is the size of each QP context, corresponding to each path in our design.

To make our design more scalable, we use the shared receive queue mechanism of InfiniBand to handle the scalability aspect for receive queue. In the previous design, receive queues corresponding to different processes was shared. We allow different paths for the same set of processes to be attached to the shared receive queue. As a result the memory usage can be represented as:

$$mem_{qp} = mem_{qp-context} + ne_s * mem_{sqe} \tag{7.6}$$

Although our current design focuses only on reducing the memory usage for receive queue, additional methods such as setting up connections only as needed (on-demand connection management) have also been shown to significantly reduce memory usage and can be used in conjunction with our design. In future, we plan to address these issues.

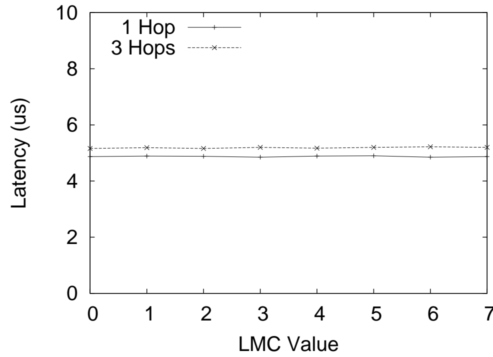


Figure 7.2: MPI Latency

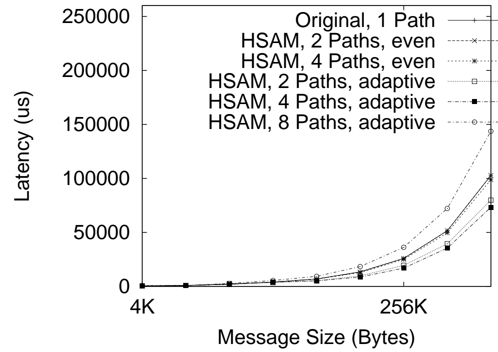


Figure 7.3: MPI Alltoall Personalized (48x1)

7.2 Performance Evaluation

In this section, we evaluate the performance of HSAM and compare its performance with the existing solution (referred to as Original for the rest of the section). We use 1 process per node (hence 48 process run is referred to as 48x1). Our evaluation consists of two parts. In the first part, we show the performance benefit we can achieve compared to the original MPI implementation using collective communication. In the second part, we provide an evaluation of our design with MPI applications. We use NAS Parallel Benchmarks [8] and PSTSWM [4], a shallow water modeling application, for our evaluation.

7.2.1 Experimental Testbed:

Our testbed cluster consists of 64 nodes; 32 nodes with Intel EM64T architecture and 32 nodes with AMD Opteron architecture. Each node with Intel EM64T architecture is a dual socket, single core with 3.6 GHz, 2 MB L2 cache and 2 GB DDR2 533 MHz main memory. Each node with AMD Opteron architecture is a

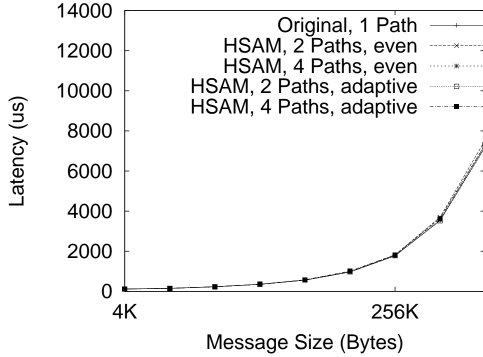


Figure 7.4: MPI AllReduce (48x1)

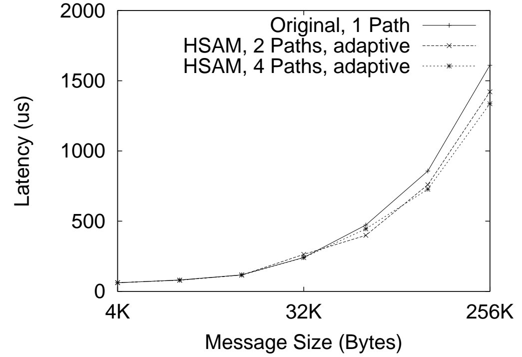


Figure 7.5: MPI Reduce Scatter (48x1)

dual-socket, single core with 2.8 GHz, 1 MB L2 cache and 4 GB DDR2 533 MHz main memory. On each of these systems, the I/O bus is 8x PCI-Express with Mellanox MT25208 dual-port DDR Mellanox HCAs attached to 144-port DDR Flextronics switch. The firmware version is 5.1.400. We have used Open Fabrics Enterprise distribution (OFED) version 1.1 for evaluation on each of the nodes and OpenSM as the subnet manager, distributed with this version.

7.2.2 Performance Benefits of HSAM with Collective Communication:

Figure 7.2 shows the ping-pong latency achieved using 2 processes by a 4-byte message with increasing value of LMC. The motivation is to understand the impact of increased routing table size (present on each switch block), with increasing value of LMC, since the number of entries in the table grow exponentially. We notice that increasing LMC does not impact the latency. The figure also represents the performance, when both processes are located on same block (1-hop) and different

blocks (3-hops). We notice that 3-hops increases the latency by 0.25 us, an increase of around 0.12 us every switch block.

In Figure 7.3, we show the performance of MPI Alltoall Personalized for 48 processes. We compare a combination of HSAM parameters; number of paths, striping policy, LMC use with original implementation. In our design, the completion filter waits for the completion of all stripes, before notifying the application with the completion. The time is dominated with the slowest stripe, even though other stripes may have finished earlier, and as a result the benefit from using hot-spot free path is nullified. Hence, using even striping does not improve the performance compared to the original implementation. For rest of the evaluation, we only focus on adaptive striping policy with HSAM. Using adaptive striping with HSAM improves the performance significantly (both 2 paths and 4 paths).

However, using 8 paths, we see a performance degradation in comparison to the original design. We have noticed that inaccuracy in estimation of path bandwidth is due to a combination of factors mentioned in the design section. Particularly, pulling off multiple completions from the completion queue adds significantly to the inaccuracy. However, we see an improvement in performance with 4 paths compared to the two paths case and an improvement of 25% with 4 paths compared to the original case. Hence, for most of our evaluation, we use 4 paths by default and the adaptive scheduling policy for our evaluation.

Figure 7.6 shows the results for 24 processes with HSAM. We can clearly notice that the corresponding dark spots in the original case are much lighter with HSAM. We are able to improve the average bandwidth by 21%.

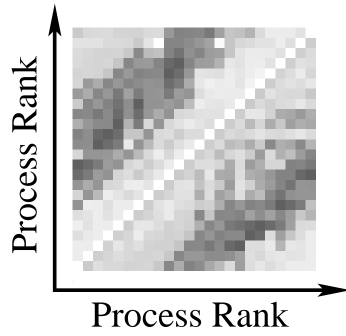


Figure 7.6: Displaced Ring Communication, 24x1, HSAM, 4 Paths, adaptive

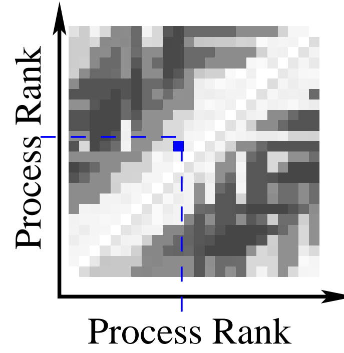


Figure 7.7: Displaced Ring Communication, 24x1, Original, 1 Path

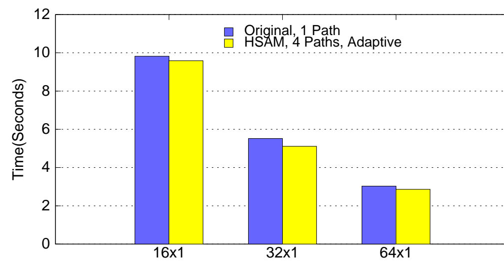


Figure 7.8: NAS Parallel Benchmarks, FT, Class B

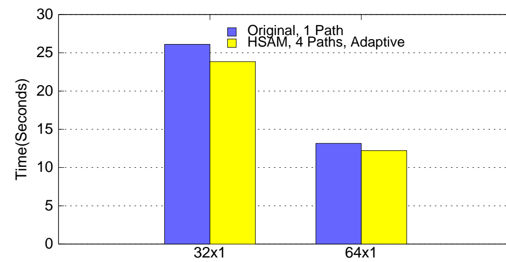


Figure 7.9: NAS Parallel Benchmarks, FT, Class C

7.2.3 Performance Benefits at Application Level

In this section, we present the results for MPI applications with HSAM. We use NAS Parallel Benchmarks [8] and PSTSWM [4]. For NAS Parallel Benchmarks, we focus on the FT benchmark, which uses MPI All-to-all personalized for communication. We use class B and class C problem size for evaluation. For PSTSWM, we use medium problem size for evaluation. Although not included in this evaluation, we have not seen performance degradation for rest of the NAS Parallel Benchmarks using HSAM.

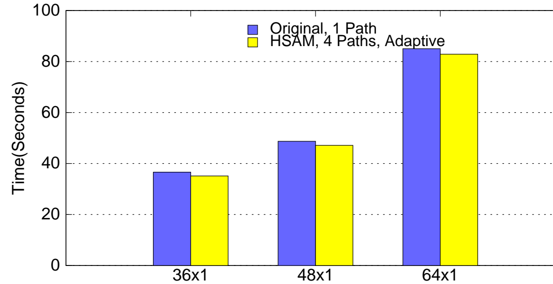


Figure 7.10: Performance Evaluation with PSTSWM

Figure 7.8 shows the results for FT benchmark, Class B problem size, for 16, 32 and 64 processes. We compare the performance of HSAM's adaptive striping with the original policy. Using 16 processes, we do not see any improvement, since the contention in the network is negligible. However, with 32 processes, we see an improvement of 8% with HSAM and 6% with 64 processes, compared to the original design. Figure 7.9 shows the results for Class C problem size with FT benchmark. With 32 processes, we see an improvement of 9%. The increased

improvement is attributed to the increased size of data transfer during MPI All-to-all phase. With 64 processes, an improvement of 8% is observed, compared to the original design.

Figure 7.10 shows the results for 36, 48 and 64 processes respectively with PSTSWM. We notice that using HSAM, we can improve the performance for 64 processes by around 4%. For rest of the combinations, we do not see significant performance improvement.

7.3 Summary

In this paper, we have designed an MPI functionality which provides hot-spot avoidance for different communication patterns, without apriori knowledge of the pattern. We have leveraged LID Mask Count (LMC) mechanism of InfiniBand to create multiple paths in the network, and studied its efficiency in creation of contention free routes. We have also presented the design issues (scheduling policies, selecting number of paths, and scalability aspects) associated with our MPI functionality. We have implemented our design and evaluated it with collective communication and MPI applications. On an InfiniBand cluster with 64 processes, we have observed an average improvement of 23% for displaced ring communication pattern. For collective communication like MPI All-to-all Personalized, we have observed an improvement of 27%. Our evaluation with NAS Parallel Benchmarks has shown an improvement of 6-9% in execution time for FT Benchmark, with class B and class C problem size.

CHAPTER 8

ENHANCED DESIGN FOR AVOIDING HOT-SPOTS WITH BETTER NETWORK PATHS UTILIZATION

In the previous chapter, we presented the HSAM functionality, which is capable of providing network hot-spot avoidance by leveraging multiple paths in the network. The primary short-coming although is its inability to leverage more than four paths in the network, due to striping overhead. As a result, we were not able to leverage the presence of other paths, with lesser contention.

In this chapter, we re-visit the adaptive approaches for hot-spot avoidance. A useful approach for hot-spot avoidance is Automatic Path Migration. We discuss the benefits and limitations of this approach. Primarily focusing on the short-comings of HSAM functionality, we present a new MPI functionality, which performs Batch Based Striping and Sorting (BSS) to leverage paths un-used by HSAM. We implement our design and evaluate it with different MPI benchmarks. The experimental evaluation shows that the BSS functionality is able to outperform the HSAM functionality, show-casing the BSS capabilities.

The rest of the chapter is organized as follows. In section 8.1, we re-visit the adaptive approaches to hot-spot avoidance. In section 8.2, we present the design for BSS based MPI design. In section 8.3, we present the implementation details

of BSS, which is followed by performance evaluation in section 8.4. In section 8.5, we conclude and summarize our results.

8.1 Adaptive Approaches for Hot-Spot Avoidance

In this section, we discuss the solution space with adaptive approaches for hot-spot avoidance. We begin with discussion of an approach which leverages Automatic Path Migration (APM) [26, 57]. We also discuss “Hot-Spot Avoidance with MVAPICH (HSAM)” [42, 56] proposed in the previous chapter.

8.1.1 Automatic Path Migration Based Hot-Spot Avoidance

As discussed in the background section, APM uses an alternate route for network failover. In addition, it allows a user to leverage the alternate path for load balancing. Hence, this approach can also be used for hot-spot avoidance. A user can specify an alternate path using the LMC mechanism and request the alternate path to be loaded. Figure 8.1 illustrates this approach. Once the primary path has

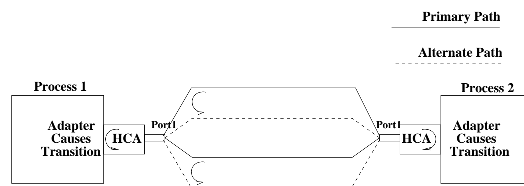


Figure 8.1: APM Based Hot-Spot Avoidance Approach

been detected as a hot-spot, a user may request the transition to be performed. This approach may also be combined with multi-pathing, with a subset of paths

as primary paths of communication and the other subset as their alternate paths. However, APM state transitions are expensive and transition of alternate path to primary path of communication may not be feasible for avoiding hot-spots [57]. APM affects the on-going communication significantly, and as a result, the estimation of path bandwidth for the communication instances may be fairly inaccurate. Hence, we do not implement and evaluate this approach in this chapter. We plan to re-visit this approach with upcoming InfiniBand products like ConnectX [3], which promises to be a low overhead solution for APM.

8.1.2 Current Software Based Approaches and Limitations

Recent literature reflects the efforts done by researchers for providing hot-spot avoidance with InfiniBand networks. We primarily focus on the HSAM scheme [56]. Under this approach, reliable connection transport model is used for data transfer and completion notification mechanism is used for path bandwidth estimation [33]. Multiple paths are defined using the LMC mechanism and weights are adjusted according to the bandwidth estimations. This approach provides significant benefits for collective communication primitives as compared to the no-multi-pathing case. However, the HSAM [56] scheme suffers from the following limitations:

- The HSAM scheme is able to use only a subset of paths for communication and increase in the number of paths beyond four leads to performance degradation [56] due to increased striping overhead.
- The HSAM scheme statically selects a number of paths to be used for communication. Although weights for different paths are updated adaptively, all

selected paths are used even though some of the paths may have very low bandwidths compared to the best path.

- Although the HSAM scheme potentially provides benefits for sequential mapping of MPI ranks to nodes, it is not clear, whether the benefits are available to different mappings of processes.

To overcome the above limitations, we use a novel approach for hot-spot avoidance. We use a Batch based Striping and Sorting scheduling policy. This approach is described in the upcoming section.

8.2 BSS Policy Based MPI Design

In this section, we propose the BSS scheduling policy followed by its integration with the MPI layer. We selected MVAPICH2 [42] as the framework of our design due to its advanced features and presence of support for multi-rail configurations (multiple adapters and ports).

8.2.1 The BSS Scheduling Policy

In this section, we propose the BSS scheduling policy. Let n_{paths} represent the total number of paths between every pair of processes and n_{batch} represent the batch of paths used during a communication instance for a message. Each BSS configuration is represented as a 2-tuple: (n_{paths}, n_{batch}), where $n_{batch} \leq n_{paths}$. These values can be specified by the user. However, as discussed later, increasing n_{paths} beyond the number of physically disjoint paths may not be beneficial. Let W_{total} represent the aggregated weight of n_{paths} and w_i represent the weight of the ith path between a pair of processes. To begin with, we initialize all the paths

with equal weights.

$$w_i = \frac{W_{total}}{n_{paths}} \tag{8.1}$$

Since BSS policy is topology and communication pattern agnostic, this weight assignment reflects the generic nature of BSS policy. This by no means is a limitation of the framework and other initial weight assignments may be plugged in as well. For every communication instance, BSS policy selects the first n_{batch} of paths from a non-decreasing weight sorted array. The data is striped on these paths proportional to their weights. Let W_{batch} denote the total weight of the paths in the batch and M be the size of the data to be transferred. The ith path sends $\frac{w_i}{W_{batch}} \cdot M$ amount of data. Let t_i denote the time taken by the stripe on the ith path. As mentioned in the previous chapter, t_i is calculated using the data delivery notification mechanism of reliable connection transport model. The updated weight w'_i of the path is represented by the following equation:

$$w'_i = W_{batch} \cdot \frac{\frac{w_i}{t_i}}{\sum_{k \in batch} \frac{w_k}{t_k}} \tag{8.2}$$

The variance in bandwidth estimation is alleviated using a linear model of path updation.

$$w'_i = (1 - \alpha) \cdot w_i + \alpha \cdot W_{batch} \cdot \frac{\frac{w_i}{t_i}}{\sum_{k \in batch} \frac{w_k}{t_k}} \tag{8.3}$$

To enable faster convergence on the paths to use, we use a higher value of α. Once the paths are updated, the array of path weights is sorted again. We also note that each communication instance only impacts the weights of the paths which have been used during the communication. Although, it does affect the ordering of the paths to be used for next iteration. In this regard, our weight updation policy is rather a heuristic, where the optimal algorithm would update the local

weights keeping global weights into account. However, the latter approach is not scalable, since it requires addition global exchange of weight arrays. Hence, we only implement the heuristic for our evaluation.

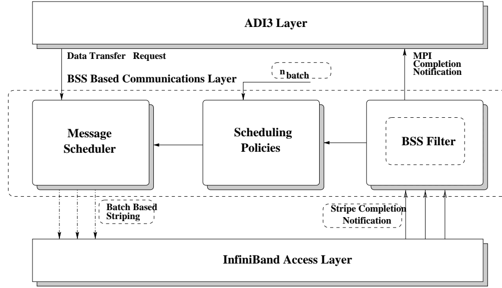


Figure 8.2: BSS Based MPI Design

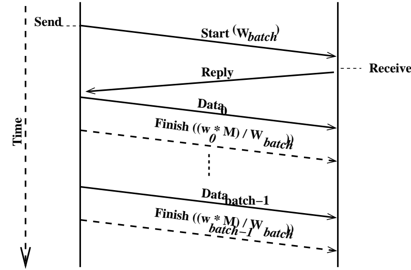


Figure 8.3: BSS Integration with Rendezvous Protocol

8.2.2 BSS Policy Based Communications Layer

In this section, we present the BSS policy based design for communications layer. The BSS communications layer has following components: Message Scheduler, Scheduling policies and BSS Filter. The message scheduler is responsible for striping the data and requesting the data transfer on n_{batch} paths. The scheduling policies component is responsible for specifying the scheduling policy and accepting user parameters. We allow the user to specify n_{batch}, and also specify the maximum number of paths to be used for communication. The scheduling policies component has been discussed in detail in the previous chapters.

Our path bandwidth estimation leverages the completion notification mechanism with the reliable connection transport model of InfiniBand. The BSS completion filter is responsible for time-stamping the acknowledgments of multiple

stripes, weight updation and sorting of the weights. It is also responsible for the notification to the MPI protocol layer, once all the stripes of a message have been received.

8.3 Detailed Design Issues

In this section, we present the detailed design issues with the BSS scheduling policy.

8.3.1 Selecting Number of Paths:

Figure 8.4 shows the block diagram of the switch used in our performance evaluation. A maximum of twelve physically independent paths are available between every pair of nodes. Using the *ibtracert* mechanism provided by InfiniBand access layer, we have concluded that the subnet manager is able to configure these paths for usage. Hence, we use twelve as the maximum number of paths with BSS. In addition, we also evaluate other BSS configurations (4,2), (8,2) and (12,3).

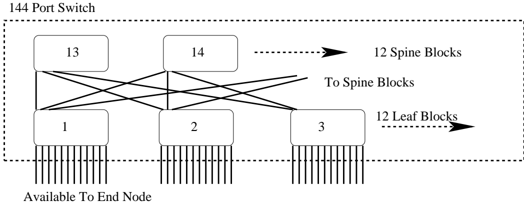


Figure 8.4: 144-port Switch Block Diagram

8.3.2 Scalability Issues:

The scalability issues with reliable connection transport model have been a focus of research for many researchers [42, 45]. Most of the researchers have focused on scalability issues with "no-multi-pathing" case. Multi-pathing requires creation of multiple QPs, which aggravates this issue significantly. Although, beyond the scope of this paper, we hereby briefly mention possible solutions for alleviating these problems:

- On-demand connection management with unreliable datagram based approach for small messages and reliable connection for large messages.

- Maintaining an upper bound on the number of reliable connection queue pairs. This would provide an upper bound on the connection memory taken by the MPI library [42, 45].

- ConnectX [3], the next generation InfiniBand promises to provide QP sharing benefits for processes on the same node. This would be beneficial for clusters based of many-core architectures and reduce the connection memory consumption significantly.

8.3.3 Integration with MPI Rendezvous Protocol:

Our design focuses on data transfer involving large messages. For small messages, software based approaches for hot-spot measurement are not feasible. In future, we plan to work on hot-spot avoidance for small messages. Each rendezvous data transfer involves a send-handle and receive-handle data structure on

the sender and receiver side, respectively. These data structures are used for notification upon the completion of data transfer on sender and receiver side. Compared to the HSAM [56] scheme, each data transfer in the BSS policy involves different W_{batch} value. To address this issue, we piggyback W_{batch} value with the rendezvous start message. This is further illustrated in Figure 8.3. The receive-handle stores this value at the receipt of the rendezvous-start message. Since RDMA is used for actual data transfer, the receiver is un-aware of the individual data size written to its memory by different stripes. Hence, we piggyback the $\frac{w_i}{W_{batch}} \cdot M$ information with the *ith* finish message. Once the aggregated value of weight received from different finish messages is equal to the W_{batch}, the MPI protocol layer is notified of the completion of data transfer.

8.3.4 Discussion

The design of the BSS policy discussed in the previous sections is completely topology agnostic. As a result, every process pair starts with the same paths to begin with (since all path weights are equal). However, in some cases it may be possible to utilize a heuristic, where different process pairs may start with different path IDs to begin with for lesser contention with no multi-pathing case. OpenSM, the subnet manager used in our implementation configures disjoint paths for every pair of ports. However, it does not consider the paths already assigned to a pair of ports for configuring the rest of the paths. A randomized relationship is currently observed among paths between different set of ports. As a result, we did not apply any heuristic as a starting point for BSS policy.

Another point for discussion is comparing BSS policy with the optimal case. An ideal case would be the presence of only one job in the network and the application has a single communication pattern. In addition, the mapping of MPI ranks to nodes is also known. For MPI_Alltoall, which represents very dense communication in the network, we wanted to evaluate the optimal case. However, we noticed that we could not find a completely disjoint path for every iteration. The main problem stems from the fact that the forward and reverse paths between a pair of InfiniBand ports do not follow the same set of links. At the same time, the path ID to be used by processes for a QP has to be the same, as per the InfiniBand specification [26]. Hence, we did not compare the performance with optimal case. As mentioned above, we are working on a routing engine, which can generate completely contention free paths for the case discussed above.

8.4 Performance Benefits with the BSS Policy Based MPI Design

In this section, we evaluate the performance of the BSS policy using a 64 node InfiniBand cluster. Different configurations of the BSS policy ((4,2), (8,2) and (12, 3)) are evaluated and compared with the HSAM [56] design. Although, more combinations are possible, we show the results for best value of n_{batch}, keeping n_{paths} constant. In section 8, we mentioned that HSAM is a special case of the BSS scheme, when n_{paths} and n_{batch} are same. We also compare the performance of the BSS policy with no multi-pathing case, the scenario commonly used in most MPI implementations over InfiniBand. This is referred with "Original" in the performance graphs, unless mentioned otherwise. Our evaluation consists of two parts. In the first part, we use Intel MPI Benchmark [2] for collective communication. In

the second part, we focus on NAS Parallel Benchmarks [8], particularly, Fourier Transform, which primarily uses MPI_Alltoall. We begin with a brief description of our experimental testbed.

8.4.1 Experimental TestBed

Our testbed cluster consists of 64 nodes: 32 nodes with Intel EM64T architecture and 32 nodes with AMD Opteron architecture. Each node with Intel EM64T architecture is a dual socket, single core with 3.6 GHz, 2 MB L2 cache and 2 GB DDR2 533 MHz main memory. Each node with AMD Opteron architecture is a dual-socket, single core with 2.8 GHz, 1 MB L2 cache and 4 GB DDR2 533 MHz main memory. On each of these systems, the I/O bus is x8 PCI-Express with Mellanox [3] MT25208 dual-port DDR Mellanox HCAs attached to 144-port DDR Flextronics switch. The firmware version is 5.1.400. We have used Open Fabrics Enterprise distribution (OFED) version 1.1 for evaluation on each of the nodes and OpenSM as the subnet manager, distributed with this version.

8.4.2 Performance Evaluation with Collective Communication

In this section, we present the evaluation of BSS, HSAM and Original for various collective communication patterns. We use MPI_Alltoall, MPI_Allgather and MPI_Allreduce for collective communication. We also evaluate different mappings of process ranks to nodes in the network:

- *Sequential Mapping:* The processes are mapped to the nodes in a sequential fashion. As an example, let the *ith* output switch port be represented by

$port_i$, as shown in the Figure 8.4. A process with MPI rank i is scheduled on $port_i$.

- *Default Mapping:* The processes are assigned randomly to the nodes. This is also the default behavior of various program launchers (Multi-purpose daemon (MPD) is an example used by MVAPICH2). However, for a consistent comparison between different configurations of BSS, HSAM and Original implementations, same mapping is used. Default mapping also represents the nodes assigned to a job, due to fragmentation in the cluster aggravated by the completion of previous jobs.

Figures 8.5 and 8.6 shows the results for MPI_Alltoall with 48 and 64 processes respectively. Pair-wise exchange algorithm is used for MPI_Alltoall [30]. However, the exchange partner for non-power-of-2 case (48 processes) is different from power-of-2 case (64 processes). We see that BSS (12, 3) performs the best, reducing the latency to half in comparison to the original case. Compared to the HSAM, latency decreases by 27%. Compared to the BSS (4, 2) case, the improvement in

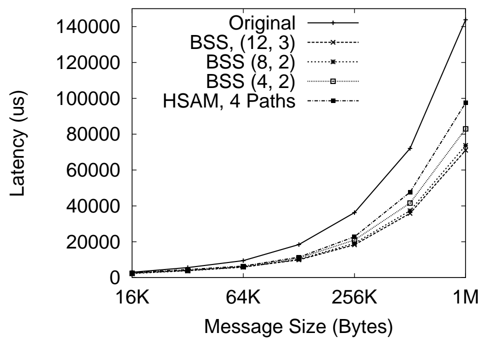


Figure 8.5: MPI_AlltoAll (48x1), Sequential Mapping

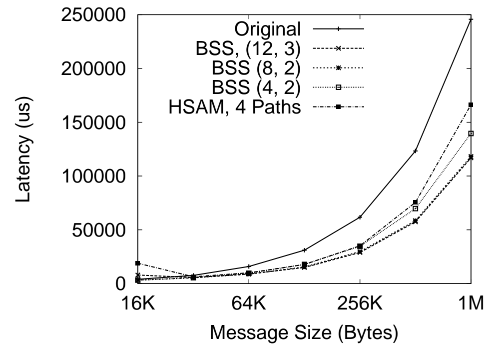


Figure 8.6: MPI_AlltoAll (64x1), Sequential Mapping

performance is 15%. The improvement in performance is due to the adaptation of path weights at the run-time by the BSS scheme. The HSAM scheme is also able to adapt to the path weights, but it does not use all the paths and ends up with sub-optimal paths for usage. For 64 process case, compared to the HSAM scheme, the improvements with the BSS scheme are 16%, 31% and 32% for (4, 2), (8, 2) and (12, 3) cases respectively. Figures 8.7 and 8.8 show the results for MPI_Alltoall with a default mapping of processes. We see that the benefits are significant compared to the Original case as well as HSAM. Hence, BSS provides benefits with different scheduling of processes.

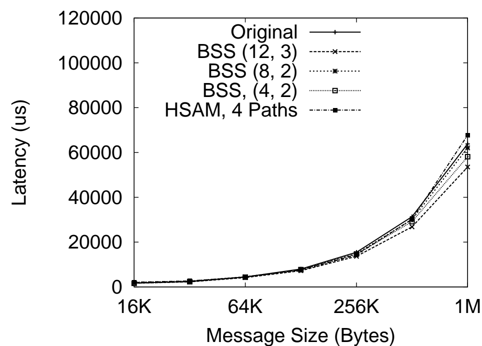


Figure 8.7: MPI_AlltoAll (32x1), Default Mapping

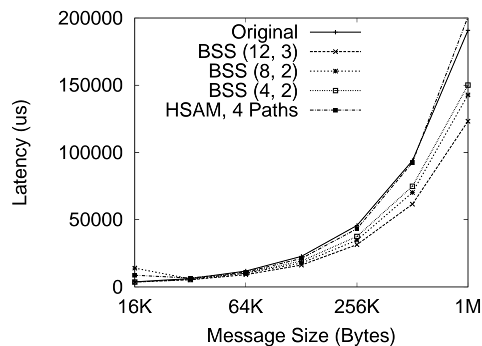


Figure 8.8: MPI_AlltoAll (64x1), Default Mapping

Figures 8.9 and 8.10 show the results for MPI_Allgather with 48 and 64 processes respectively. For the message sizes presented in these figures, MPI_Allgather uses the ring algorithm [30]. Looking from the topology perspective, each switch block has exactly one out-bound and in-bound communication. Rest of the communication is within a leaf block, as shown in Figure 8.4. As a result, insignificant contention is observed, and all the cases perform similarly. However, under random

allocation of nodes to a job, the number of in-bound and out-bound communication instances increases and the contention increases significantly. Figure 8.11 and

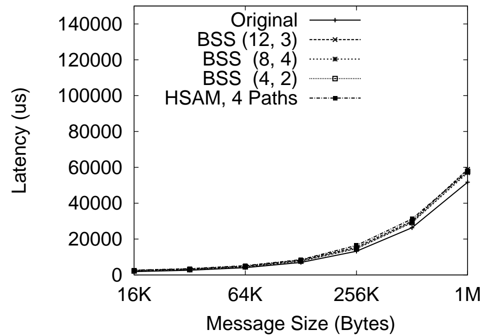


Figure 8.9: MPI_Allgather(48x1), Sequential Mapping

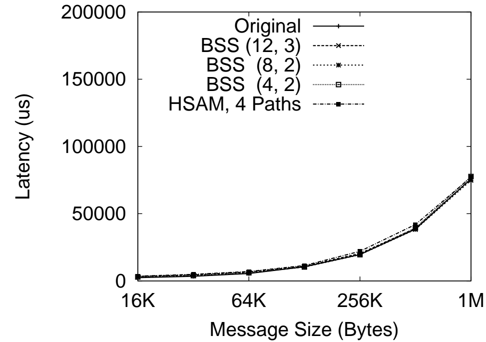


Figure 8.10: MPI_Allgather (64x1), Sequential Mapping

Figure 8.12 show the results for such a scenario. For 32 process case, we see that the MPI_Allgather latency reduces by 32% from the original case. Albeit, different combinations of the BSS scheme do not show improvement compared to each other, since the amount of contention in the network is lesser compared to MPI_Alltoall. However, for 64 processes, default distribution of processes leads to significantly more contention than 32 processes. As a result, the improvement compared to Original and the HSAM scheme is 43% and 32%, respectively.

Figures 8.13 and 8.14 show the results for MPI_Allreduce with 48 and 64 processes respectively. For power-of-2 number of processes, MPI_Allreduce uses MPI Reduce-Scatter followed MPI_Allgather. MPI Reduce-Scatter itself uses pairwise exchange algorithm for the message sizes shown in the graph. Hence, one of the steps is hot-spot free (MPI_Allgather, as shown above), however the other step has contention (pair-wise exchange). Hence, for 64 processes, we see benefits in the

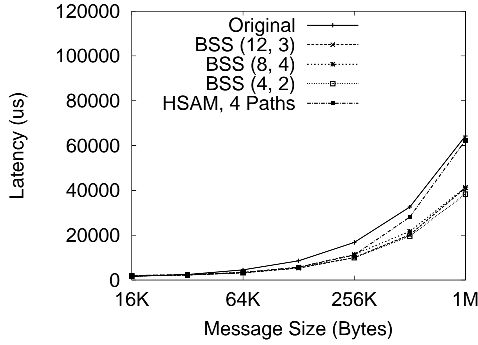


Figure 8.11: MPI_Allgather (32x1), Default Mapping

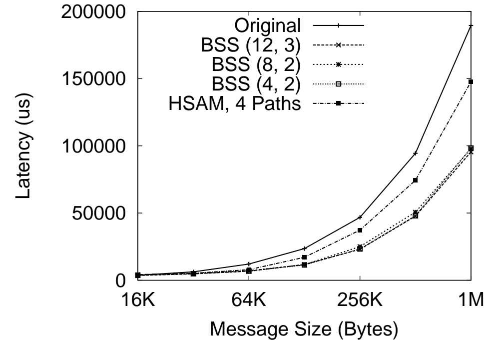


Figure 8.12: MPI_Allgather (64x1), Default Mapping

first phase. For 48 processes, recursive doubling algorithm is used. As a result, no contention is observed.

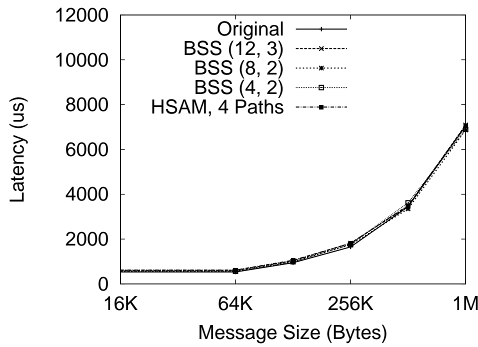


Figure 8.13: MPI_AllReduce (48x1), Sequential Mapping

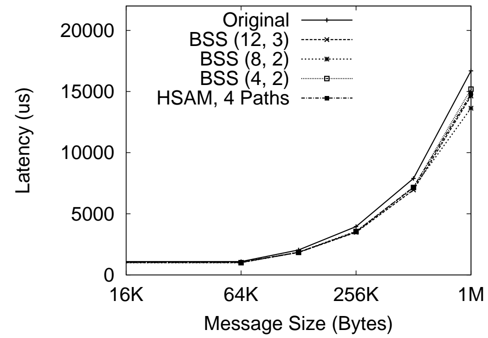


Figure 8.14: MPI_AllReduce (64x1), Sequential Mapping

Figures 8.15 and 8.16 show the results for MPI_Allreduce with default mapping of processes. Even though the algorithms remain the same, the amount of contention increases and as a result different configurations of BSS show significant benefits in comparison to HSAM and Original implementations.

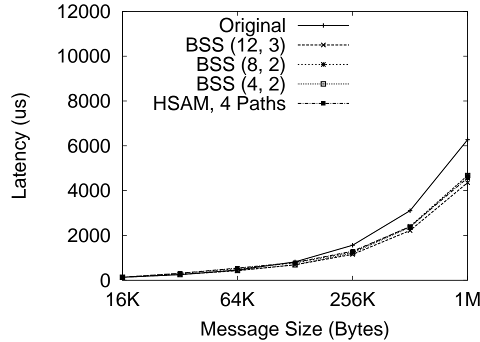


Figure 8.15: MPI_AllReduce (32x1), Default Mapping

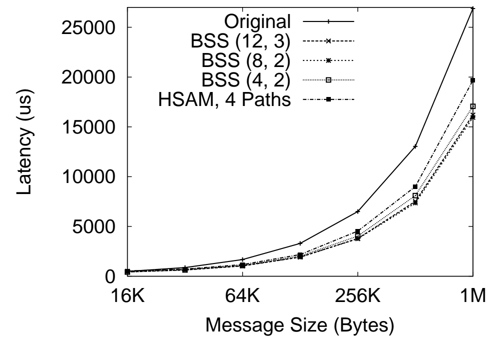


Figure 8.16: MPI_AllReduce (64x1), Default Mapping

8.4.3 Performance Evaluation with MPI Applications

Figures 8.17 and 8.18 represent the results for NAS Parallel Benchmarks [8] with Fourier Transform with Class B and Class C problem size respectively. Other NAS Parallel Benchmarks do not show any degradation in performance, as shown in the Figure 8.19. In all experiments, we only use the sequential mapping. As seen in the previous section, sequential mapping of processes produces the least performance improvement. We expect that the benefits shown with the sequential mapping of processes are the least a job will achieve for any mapping of processes.

Fourier Transform benchmark uses collective communications primitives like MPI_Alltoall, MPI_Reduce and MPI_Bcast. For MPI_Alltoall, we observed significant performance benefits in the previous section. We notice that the benefits are transferred to the Fourier Transform benchmark.

For Class B, compared to the Original case, different versions of the BSS policy show performance benefits ranging from 10%-11% in the execution time. We also notice that the performance of our implementation of the HSAM scheme performs

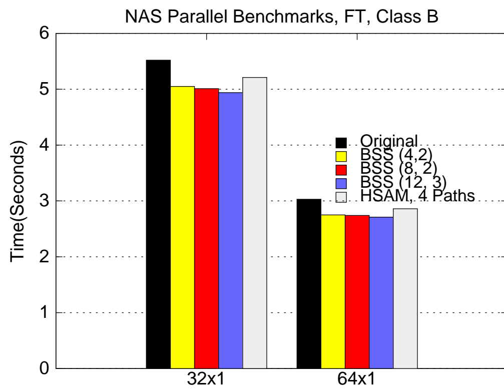


Figure 8.17: NAS Parallel Benchmarks, FT, Class B

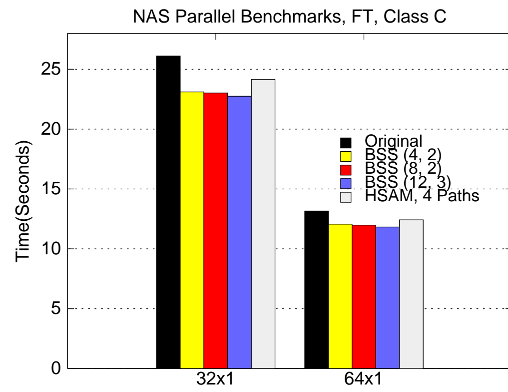


Figure 8.18: NAS Parallel Benchmarks, FT, Class C

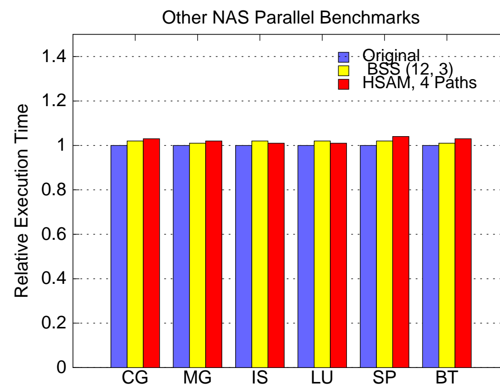


Figure 8.19: Other NAS Parallel Benchmarks, Class C, 64 Processes

very closely to the results presented in the previous chapter. Different configurations of BSS further improve the execution time by 6-7%. Similar improvements are seen for both 32 and 64 process case. For Class C problem size, BSS configurations perform similarly. Compared to the original case, BSS configuration improve the execution time by 13% for 32 process case and 6.5% compared to the HSAM scheme. Similar performance improvement is observed with 64 process case.

8.5 Summary

In this chapter, we presented enhanced approaches to efficiently utilize physically independent and paths unused by HSAM. We proposed a novel scheduling policy, which performs Batch-based Striping and Sorting (BSS) during the application execution to adaptively eliminate the path(s) with low bandwidth. We discussed the detailed design issues including the scalability aspects. We implemented our design and evaluated with collective communication primitives and applications. We compared the performance of different BSS configurations, the best configuration of the HSAM [56] scheme and the original case (no multi-pathing at all). Using MPI_Alltoall, we achieved an improvement of 27% and 32% in latency with different BSS policy configurations compared to the best configuration of the HSAM scheme on 32 and 64 processes, respectively. A default mapping of tasks in the cluster shows similar benefits. Using MPI_Allgather and MPI_Allreduce with a default mapping of tasks, an improvement of 32% in latency is observed for 64 processes. Using the Fourier Transform benchmark from NAS Parallel Benchmarks [8]

with different problem sizes, the execution time can be improved by 5-7% with different BSS policy configurations compared to the best HSAM configuration and 11-13% from the original implementation.

CHAPTER 9

NETWORK FAULT TOLERANCE USING AUTOMATIC PATH MIGRATION

As discussed in the previous chapters, InfiniBand is being widely accepted as the next generation interconnect due to its open standard and high performance. As a result, clusters based on InfiniBand are becoming increasingly popular, as shown by the TOP 500 [6] Supercomputer rankings. However, increasing scale of these clusters has reduced the Mean Time Between Failures (MTBF) of components. Network component is one such component of clusters, where failures of network interface cards (NICs), cables or switches breaks the existing path(s) of communication. InfiniBand provides a hardware mechanism, *Automatic Path Migration (APM)*, which allows user transparent detection and recovery from network fault(s). However, the current InfiniBand literature lacks the understanding of associated design trade offs with APM and performance analysis.

In this chapter, we design a set of modules; *alternate path specification module*, *path loading request module* and *path migration module*, which work together for providing network fault tolerance for user level applications. We integrate these modules for simple micro-benchmarks at the Verbs Layer, the user access layer for InfiniBand, and study the impact of different state transitions associated with

APM. We also integrate these modules at the MPI [38, 39] (Message Passing Interface) layer to provide network fault tolerance for MPI applications.

The rest of the chapter is organized as follows. In section 9.1, we present the overall design followed by network fault tolerance modules in section 9.2. In section 9.3, we discuss the interaction of main thread and asynchronous thread. In section 9.4, we present the performance evaluation with MPI applications and Verbs level benchmarks. We present the summary and conclude in section 9.5.

9.1 Overall Design

In this section, we present the overall design of network fault tolerance modules, their interactions with a user-level application and the communication layer of a user-level application. The interaction is shown in Figure 9.1.

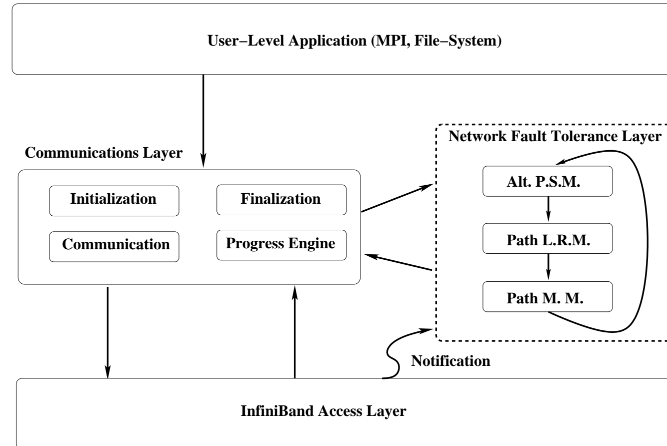


Figure 9.1: Overall Design of Network Fault Tolerance Modules and Interaction with User Applications

For simplicity, we have assumed that the interface between the access layer and the user-level application consists of a communication layer with the modules; *Initialization Module*, *Communication Module*, *Progress Engine* and *Finalization Module*. For different user-level applications, some of the modules may have more functionality than the others. Nevertheless, the modules are portable for different user-level applications (MPI, File-Systems etc).

Figure 9.1 also shows the order in which the network fault tolerant modules can be called by the communications layer modules. The alternate path specification module can be called at any point during the execution of the program. The path loading request module can be called in conjunction with the alternate path specification module. It can also be called separately during the execution of the application. The path migration module can be called only if the QP(s) for which the request is made are in the ARMED state. The notifications for different transition states of the APM are handled by the network fault tolerance modules.

9.2 Design of Network Fault Tolerance Modules

In this section, we present the modules which form the core in providing network fault tolerance for our design. There are three modules which work in conjunction; *alternate path specification*, *path loading request module* and *path migration module*. The alternate path specification module is responsible for deciding the alternate path to be used in the presence of network fault(s). The path loading request module is responsible for requesting the alternate path to be loaded in the path migration state machine. The path migration module is responsible for transition of alternate path to the primary path of communication.

9.2.1 Alternate Path Specification Module

This module is responsible for specifying an alternate path to be used by a queue pair. The request for alternate path to be used can be done manually, or automatically by the HCA, should an error occur on the primary path of communication.

In our design, the alternate path can be specified by the user or chosen automatically by the module. Specification of the alternate path requires providing a couple of parameters; alt_{DLID} (the destination LID of the alternate path), alt_{PORT} (the HCA port for the alternate path), $alt_{SRC-PATH-BITS}$ (the LMC value to be used for the alternate path). A primary benefit of using APM is that the connection remains established during the transition of path. This is achieved by keeping qp_{num} (the QP number) to be the same for the alternate path. An example of alternate path specification is shown in Figure 9.2. The primary path and the alternate path can take any values (the alternate path can be same as the primary path of communication). As shown in the figure, the first case uses the alternate port as the alternate path. The second case uses the same port, however an alternate path in the network, which can be used by specifying a different value for $alt_{SRC-PATH-BITS}$. In this chapter, we select the other port of the HCA as the alt_{PORT} and corresponding LID as the alt_{DLID}, if the other port is also available for communication. Otherwise, we use a different path in the network by selecting a different $alt_{SRC-PATH-BITS}$ value from the primary path. In the latter case, the alt_{DLID} and alt_{PORT} remain the same as the primary path.

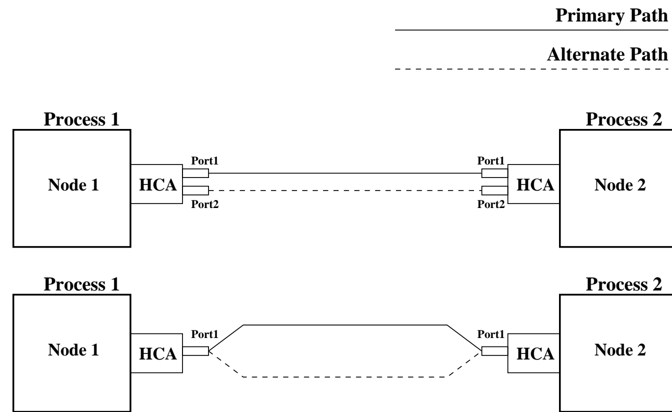


Figure 9.2: An Example of Primary and Alternate Path of Communication used by PSM

9.2.2 Path Loading Request Module

This module is responsible for initiating the loading of the alternate path for a QP. The module accepts a parameter for the list of the processes, for which this step needs to be done. This module can be invoked during anytime of the program execution after the RESET-INIT transition sequence has been completed for the QP(s). The completion of the request can be done using asynchronous events or polling mechanism. We discuss the trade offs of these approaches as follows.

Completion of Path Loading Request: The completion of the request for alternate path can be done using notification mechanism. Alternatively, the Verbs API provides a $query-qp$ function call to check the path migration state of a QP. Using the $query-qp$ mechanism, we can ascertain the path migration state of a QP (path migration state should be ARMED to call path migration module, should be migrated to call the path loading request module). We have noticed that the cost of querying a QP is higher than the overhead generated with the asynchronous

notification. Hence, we use an asynchronous thread based notification handling of these events. The completion of the request(s) is notified by asynchronous event(s), which we refer to as the $event_{ARMED}$ in this paper.

9.2.3 Path Migration Module

This module is invoked when a user wants to use the alternate path to be used as a primary path of communication, in the absence of a network fault. This functionality is useful in providing load balancing with the available paths. Alternatively, if an error occurs during transmission, the HCA requests the alternate path to be loaded as the primary path of communication, without intervention from the user application. This module assumes that the path loading request module has successfully loaded the alternate path, and the alternate path is in a healthy state. The completion of this sequence is notified with the help of asynchronous events, which are referred as $event_{MIGRATED}$ in this work. The asynchronous thread discussed in the previous section is enhanced to handle these events. In the performance evaluation section, the invoking of this module is referred by Armed-Migrated legend.

9.3 Interaction of Main Execution Thread and the Asynchronous Thread With Network Fault Tolerance Modules

In this section, we present the interactions of Main Execution Thread and the Asynchronous Thread with the Network Fault Tolerance Modules. Figure 9.3 shows the possible interactions. The interactions from the main execution thread are shown with solid lines, the interactions with asynchronous thread are shown

with dotted lines. Although, both threads can interact with the network fault tolerance modules, the main execution thread can execute the modules at any stage of the application execution. The asynchronous thread can call the path migration module on the occurrence of $event_{ARMED}$. On the occurrence of $event_{MIGRATED}$, the asynchronous thread can call alternate path specification module and path loading request module or the alternate path specification module only. We limit the asynchronous thread to execute the modules only at the occurrence of events, since the thread is active only on the occurrence of events.

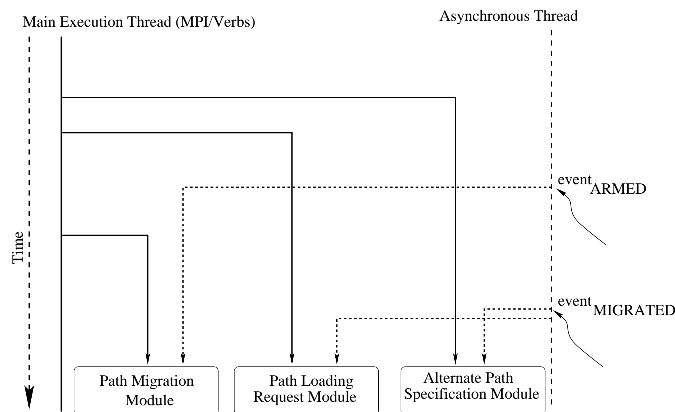


Figure 9.3: Interaction of Network Fault Tolerance Modules with Main Execution Thread and Asynchronous Thread

9.3.1 Integration of Network Fault Tolerance Modules at Verbs and MPI Layer

We implement our network fault tolerance modules, so that various user level applications can leverage them without any changes to the modules specific to

the application. For the micro-benchmarks at the Verbs Layer, we extend the micro-benchmark suite discussed in our previous work [33].

9.4 Performance Evaluation with the APM Based Network Fault Tolerance Modules

In this section, we evaluate the performance of our Network Fault Tolerance modules over InfiniBand. At the Verbs layer, we design a ping-pong latency test and a computation test. We study the impact on performance for different transition states in APM, when they are requested at different points during the execution of the test. This is followed by the study with the MPI applications and the impact of these state transitions on the execution time, in the absence and the presence of faults. We begin with a brief overview of our experimental testbed.

9.4.1 Experimental Testbed

Our Experimental Testbed consists of a set of Intel Xeon nodes each having a 133 MHz PCI-X slot. Each node has two Intel Xeon CPUs running at 2.4 GHz , 512 KB L2 cache and 1 GB of main memory. This cluster uses 2nd Generation MT23218 4X Dual Port HCAs from Mellanox [3]. We used the Linux 2.6.9-15.EL kernel version [5] and Verbs API (VAPI) from Mellanox provided with the InfiniBand Gold CD (IBGD). The HCA firmware version used is 3.3.2. The nodes are connected with a 144-port Single Data Rate (SDR) switch. The switch uses OpenSM; a popular subnet manager provided with IBGD. Since each HCA has two ports, we connect both ports to the switch, and use first port as the primary path and second port as the alternate port for communication.

9.4.2 Evaluation of the Network Fault Tolerance Modules at the Verbs Layer

In this section, we evaluate the performance of the network fault tolerance modules at the Verbs layer. To study this performance, we design a ping-pong latency test. The test uses two processes: sender and receiver. The sender posts a send descriptor corresponding to the ping message and posts a receive descriptor for the pong message (to be sent by the receiver). The sender then polls on the completion queue for receiving a receive and a send completion queue entry (CQE) each. The receiver polls on its completion queue for a receive CQE. Once a receive CQE is obtained, the receiver sends a pong message back to the sender. This step is repeated for a large number of iterations. The sender reports the latency as the half of the total time for above operation. To understand the impact on a large scale cluster, we create multiple QPs between these processes. These QPs are used in a round-robin fashion for communication. The legend corresponding to original is the case when none of the network fault tolerance modules are invoked and 1 QP is used.

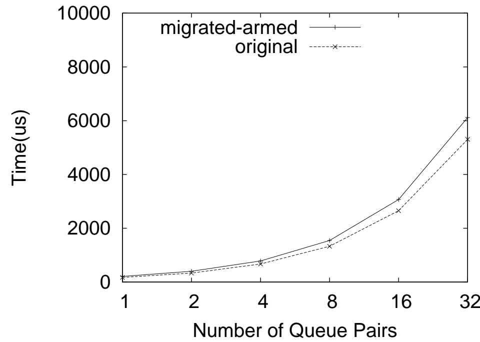


Figure 9.4: Transition From INIT-RTS, Small Number of QPs

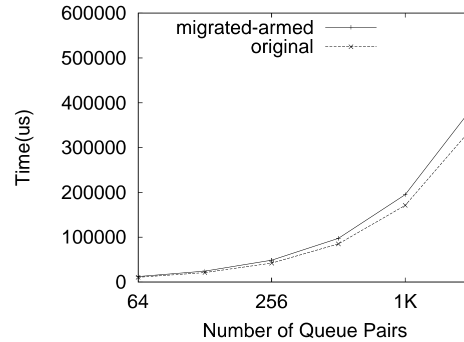


Figure 9.5: Transition From INIT-RTS, Large Number of QPs

In Figure 9.4 and Figure 9.5, we present the time consumed in INIT-RTS transition sequence, when the alternate path specification module and the path loading request module are invoked during the RTR-RTS phase. We compare its performance with the original case. We notice that the total time taken by each of the lines is linear with increasing number of queue pairs (the x-axis is a log scale). An increased time in execution by around 15% is noticed compared to the original case. These graphs reflect the timings for requesting the APM sequence. The completion of these requests is notified with the help of asynchronous events. For all the remaining tests, first port is used as the primary path of communication and the second port is used as the alternate path of communication for all QPs.

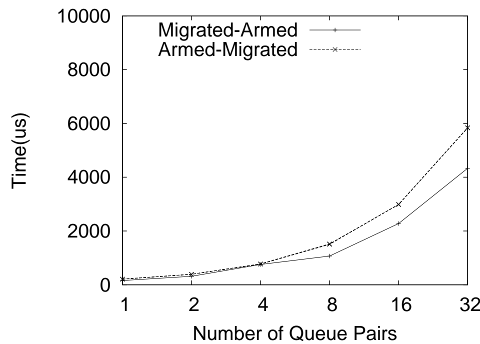


Figure 9.6: Timings for different transition states in APM, Small Number of QPs

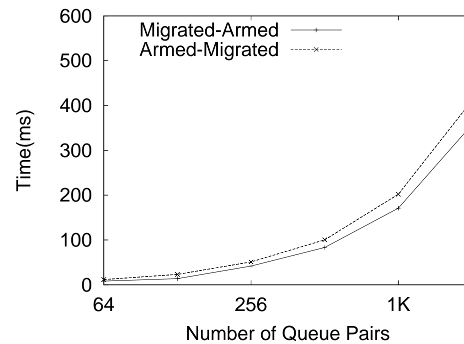


Figure 9.7: Timings for different transition states in APM, Large Number of QPs

9.4.3 Impact of PSM and LRM on QP Transitions

In Figure 9.6 and Figure 9.7, we present the time consumed in Migrated-Armed and Armed-Migrated transition sequences with the increasing number of QPs between the processes. To calculate the timings for Migrated-Armed transition, the alternate path specification module is invoked during INIT-RTR phase and time is calculated till $event_{ARMED}$ for all QPs is received by the asynchronous thread. For calculating the time for the Armed-Migrated transition, path migration module is invoked for all QPs. Once the asynchronous thread receives $event_{MIGRATED}$ for all QPs, the shared data structures between the main thread and the asynchronous thread are updated. A linear trend is observed with the increasing number of QPs in these transitions. For small number of QPs, Armed-Migrated transition takes around 30% more time than Migrated-Armed transition. For larger number of QPs, the time reduces to around 16%. The main purpose of the above tests is to calculate the maximum penalty observed by a user-level application. However, since these requests are non-blocking, it remains to be seen, how these transitions impact the ongoing communication.

9.4.4 Impact of Network Fault Tolerance Modules on Latency

Figure 9.8 compares the performance of the original case with different transition sequences using the ping-pong latency test. We slightly modify the test to report the latency observed at every iteration to clearly understand the impact of different transitions on the latency. In our evaluation, we note that the latency observed increases, till all the events corresponding to a transition sequence are

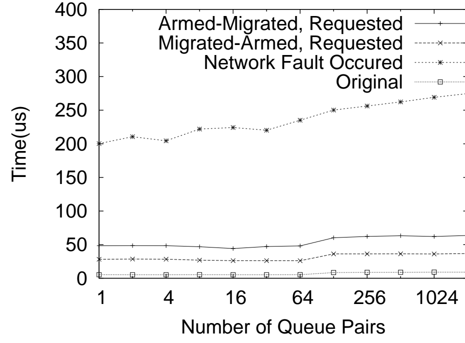


Figure 9.8: Impact on Latency for 128 Byte Message with Increasing Number of QPs

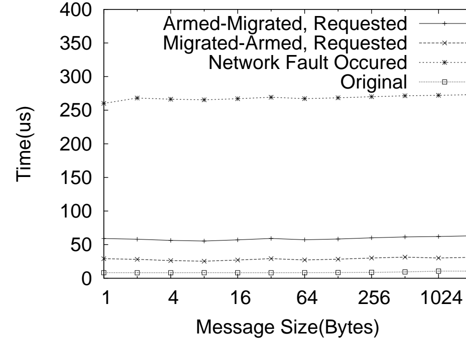


Figure 9.9: Impact on Latency for Small Messages using 512 QPs with Increasing Message Size

received. For those latency values, we calculate the average and report them in the figure. We have also observed that the number of iterations in the test, which have impact on latency is very close to the number of QPs for which the transition is requested. As a result, we almost see a flat curve for the average latency. The results show that both Migrated-Armed and Armed-Migrated requests add significant overhead to the ongoing communication. However, this overhead remains constant with the increase in the message size.

We now show the results for our acid test, the impact of performance on latency, when a network fault occurs. After the alternate path is loaded, we disable the primary path of communication by un-plugging the cable corresponding to the primary path of communication on the sender side. The HCA automatically moves the alternate path as a primary path of communication for the currently used QP. Since QPs are used in a round robin fashion, this step is executed for all QPs. We measure the average latency observed till the $event_{MIGRATED}$ for all QPs has been generated. This test helps us understand, the impact on latency for small messages

on large scale clusters, when each process pair uses one QP for communication. Figure 9.9 shows the impact on latency for small messages, when 512 QPs are used for communication. We notice that the amount of overhead remains almost same with increasing message size. Hence, the overhead incurred per QP remains the same independent of the message size.

9.4.5 Impact of Network Fault Tolerance Modules on Computation

Figures 9.10 and 9.11 show the impact on computation for different APM transition sequences. Since the $event_{ARMED}$ and $event_{MIGRATED}$ are handled by the asynchronous thread, both processes are executed on the same node of a dual-processor machine. The test initiates APM transition request for all QPs and performs the computation. A computation loop large enough is chosen empirically, such that all interrupts are generated before the computation loop finishes. The time spent in requesting the transition is subtracted and the performance is reported. For different computation loops, the total time of computation is varied to understand the performance impact with increasing computation. We observe that the computation time increases significantly with the increasing number of QPs. The time taken to handle the $event_{MIGRATED}$ and $event_{ARMED}$ contributes significantly to this overhead. However, with the increasing size of the computation loop, the overhead incurred remains same for the same number of QPs. For 2048 QPs, the overhead incurred is almost 60 ms.

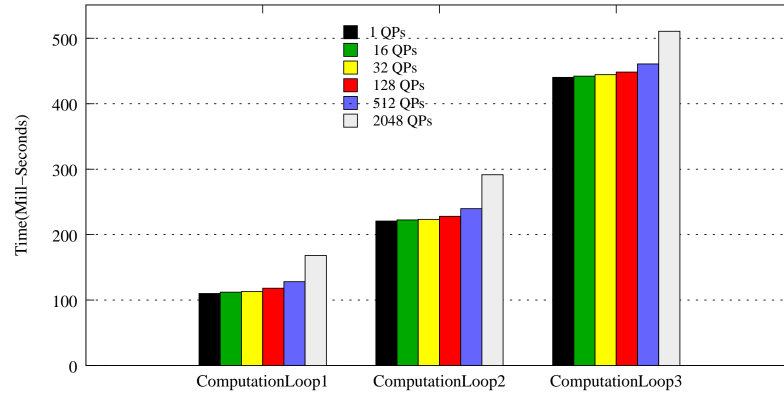


Figure 9.10: Impact on Computation, Migrated-Armed Requested During Computation

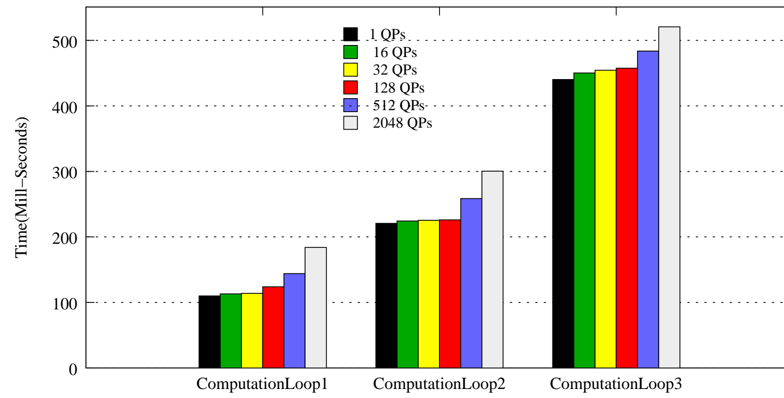


Figure 9.11: Impact on Computation, Armed-Migrated Requested During Computation

9.4.6 Evaluation of the Network Fault Tolerance Modules at the MPI Layer

In this section, we present the performance for NAS parallel benchmarks [8], when different APM sequence transitions are requested. The impact on performance in the presence of network faults is also studied. A 4x2 configuration (4 nodes and 2 processes per node) is used for executing the applications. The applications are profiled to make sure that network fault tolerance modules are invoked during the critical execution phase of the application. The primary communication path is broken by unplugging the cable at different points in the application execution for sixteen runs. The average performance observed is presented.

Figure 9.12 and Figure 9.13 show the results with different transitions sequences in APM using Integer Sort kernel, with Class A and Class B problem size. The results in the presence of network faults are also presented. In the absence of network faults, different APM transition sequences incur some overhead for Class A. In the presence of network fault, a very significant amount of overhead is observed. Since the results reflect an average case, they show a healthy mixture of the cases, when the application was busy computing, busy in communication and their combinations. Increasing the number of QPs to emulate a large scale cluster also shows an interesting trend. In the presence of a network fault, all QPs used in the round robin fashion observe a transition of alternate path to the primary path. Figure 9.13 shows the results for Class B problem size. The execution time is longer for the problem size. The impact of different APM transition sequences is lesser as a result. The time for QP transitions in the absence of faults and presence of faults largely remains independent of the message size as shown during the

performance evaluation with the tests at the Verbs layer. The number of events generated are also largely dependent upon the number of QPs used. Hence as the execution time of an application increases, the relative overhead shown due to APM in both the absence and the presence of faults decreases.

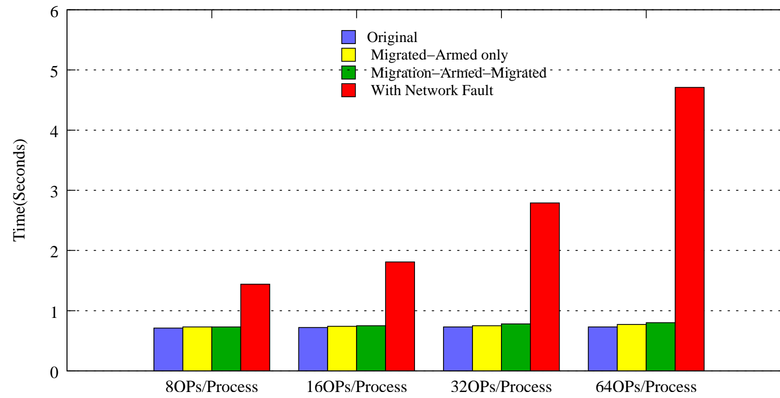


Figure 9.12: Performance Evaluation on IS, Class A, 4x2 Configuration

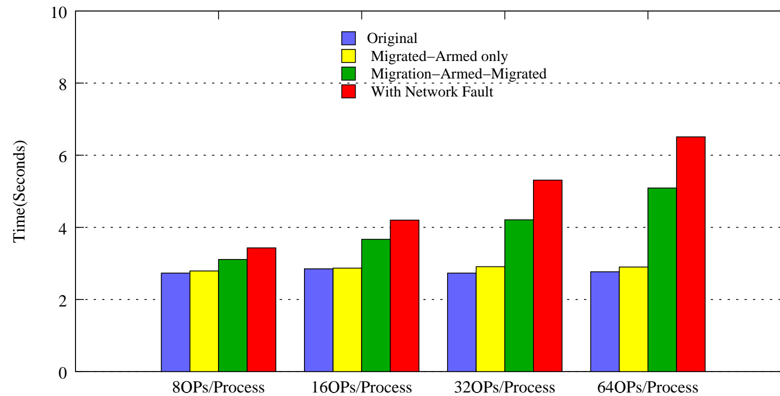


Figure 9.13: Performance Evaluation on IS, Class B, 4x2 Configuration

Figure 9.14 and Figure 9.15 show the results for NAS FT Class B and LU Class B respectively. Since the overhead incurred per QP almost remains same, when a

network fault occurs, we notice that the percentage of performance degradation is much lesser in these cases. Even with increasing the number of QPs/process to 64, we only notice around 5-6% degradation in performance. For LU class B in particular, the execution time is around 256 seconds, and hence the overhead of state transitions is amortized with the long running application. Hence for applications running for reasonably long time, APM incurs almost negligible overhead in the overall execution time.

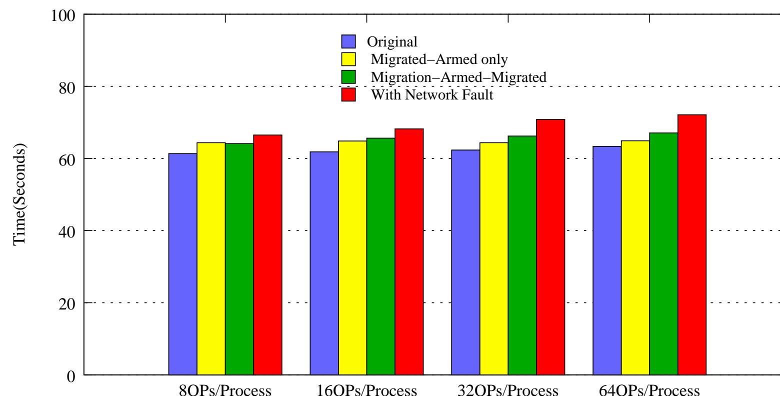


Figure 9.14: Performance Evaluation on FT, Class B, 4x2 Configuration

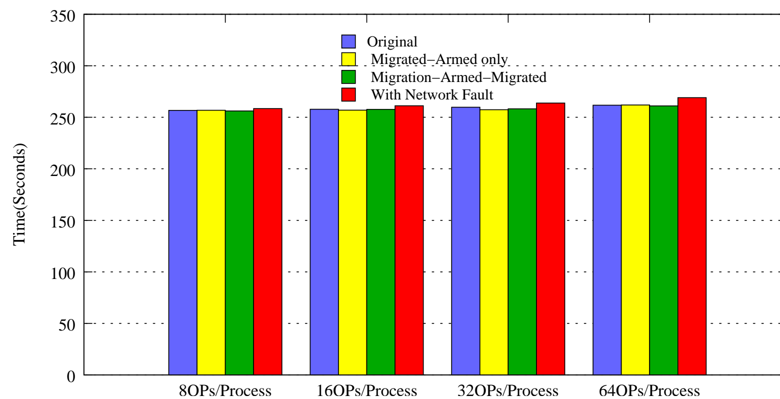


Figure 9.15: Performance Evaluation on LU, Class B, 4x2 Configuration

9.5 Summary

In this chapter, we have designed modules; *alternate path specification* module, *path loading request module* and *path migration module*, which work together for providing network fault tolerance with APM for user level applications. We have integrated these modules for simple micro-benchmarks at the Verbs Layer; the user access layer for InfiniBand, and studied the impact of different state transitions associated with APM. We have also integrated these modules with the MPI layer to provide network fault tolerance for MPI Applications. Our performance evaluation has shown that APM incurs negligible overhead in the absence of faults in the system. For MPI applications executing for reasonably long time, APM causes negligible overhead in the presence of network faults. For Class B, FT and LU NAS Parallel Benchmarks with 8 processes, the degradation is around 5-7% in the presence of network faults.

CHAPTER 10

SOFTWARE BASED NETWORK FAULT TOLERANCE ON CLUSTERS WITH UDAPL INTERFACE

In the last couple of years, Network APIs like uDAPL (user Direct Access Provider Library) are being proposed to provide a network-independent interface to different RDMA-enabled interconnects. Clusters with combination(s) of these interconnects are being deployed to leverage their unique features, and to provide network failover in wake of transmission errors. However, wide variety of interconnects pose portability issues. This limits their different combinations to be used in network failures, in addition to providing optimal performance.

In this chapter, we design a network fault tolerant MPI using uDAPL interface, making this design portable for existing interconnects. Our design provides failover to available paths, asynchronous recovery of the previous failed paths and recovery from network partitions without application restart. In addition, the design is able to handle network heterogeneity, making it suitable for the current state of the art clusters. To achieve these goals, we design a set of low overhead modules; *completion filter and error-detection*, *message (re)-transmission* and *path recovery and network partition handling*, which perform completion filter and detection, (re)-transmission and recovery from network partitions, respectively. We implement

our design and evaluate it with micro-benchmarks and applications. Our performance evaluation shows that the proposed design provides significant performance benefits to both homogeneous and heterogeneous clusters. Experiments reveal that the proposed network fault tolerance modules incur very low overhead and provide optimal performance in wake of network failures for simple MPI micro-benchmarks and applications. In addition, in the absence of such failures, using a heterogeneous 8x1 configuration of InfiniBand Architecture (IBA) and Ammasso-GigE, we are able to improve the performance of NAS Parallel Benchmarks by 10-15% for different benchmarks. For simple micro-benchmarks, we are able to improve the throughput by 15-20% for uni-directional and bi-directional bandwidth tests. Even though, the evaluation in the chapter has been done using InfiniBand and Ammasso-GigE, there are emerging interconnects, which plan to support uDAPL interface and are not yet available in market commercially. The proposed design is generic and capable of supporting any interconnect with uDAPL interface.

The rest of the chapter is organized as follows. In section 10.1, we present the overall design of uDAPL based network fault tolerant MPI. In section 10.2, we present the basic infrastructure for network fault tolerance design. In section 10.3, we discuss the design of the communications and network fault tolerance layer. In section 10.4, we present the evaluation of uDAPL based MPI. In section 10.5, we conclude and present the summary.

10.1 Overall Design for uDAPL Based Network Fault Tolerant MPI

In this section, we present the overall design for our uDAPL based network fault tolerant MPI. This is further illustrated in Figure 10.1. The figure represents the

overall design and an example node configuration consisting of both IBA and GigE devices. In section 10.2, we present *multi-network* abstraction layer, which provides a uniform interface to our design for clusters with network heterogeneity. We also discuss the implementation issues associated with using multiple interconnects.

In section 10.3, we present the main component of our design, *communications and network fault tolerance layer*. Figure 10.4 presents the interaction of different components in communications and network fault tolerance layer. This layer comprises of modules, which work together for scheduling the communication in an efficient manner for providing network fault tolerance. The *message (re)-transmission module* in this layer is responsible for scheduling the communication on available paths according to the scheduling policy. The *completion filter and error detection* module detects error and provides information to *message (re)-transmission* about the failed work request(s). The *path repository* maintains the available paths for every pair of communication nodes. This layer also consists of *path recovery and network partition handling module*, which is responsible for recovery of failed paths and dealing with network partitions.

10.2 Basic Infrastructure For Network Fault Tolerance Design

In this section, we discuss the basic design, which acts as an infrastructure for providing network fault tolerance. Our design is capable of providing network fault tolerance for clusters using single interconnect, in addition to a combination of interconnects supporting uDAPL interface. We begin with the introduction of multi-network abstraction layer.

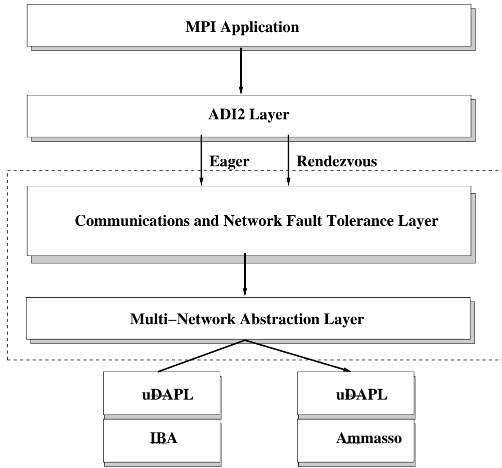


Figure 10.1: Overall Design of Network Fault Tolerant MPI with a node comprising of multiple networks

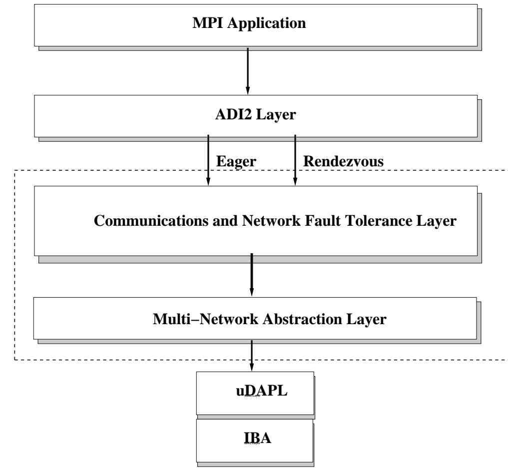


Figure 10.2: A Node with only IBA Network

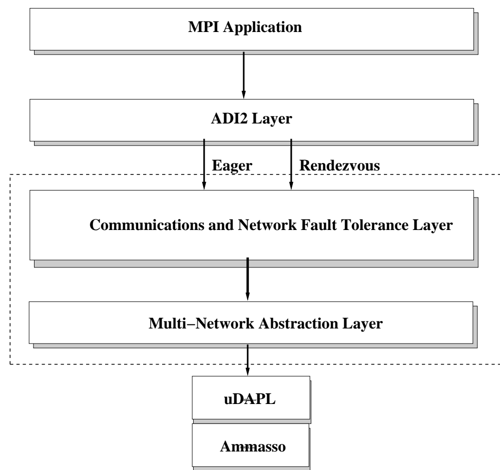


Figure 10.3: A Node with only Amasso Network

10.2.1 Multi-Network Abstraction Layer

As shown in Figures 10.1, 10.2 and 10.3, our design is capable of combining clusters with a combination of interconnects, supporting uDAPL interface to the user applications. Homogeneous clusters are a special case of this configuration. Each interconnect specifies its own uDAPL library, built over the interconnect's access layer. Hence, presence of an equivalent abstraction is imperative to hiding network heterogeneity. This layer provides an equivalent interface of multiple uDAPL interfaces to the communications and fault tolerance layer. To provide such an abstraction, this layer maintains unified data structures for end point(s), public service point(s), completion queue(s) and available paths between processes.

10.2.2 Implementing Abstraction Layer over uDAPL

In our previous work with uDAPL [12], we have presented *asynchronous* and *polling* based connection management schemes to connect EPs associated with different processes. In the design, the EP(s) information is exchanged, followed by mandatory *ep_connect* function call to connect them as specified by the uDAPL specification. However, the design assumed the presence of only one network interface. To support network heterogeneity, each node exchanges its node configuration at the MPI initialization phase. Node configuration comprises of *uDAPL provider* information, and associated parameters with different interconnects. This information is communicated to peers at the time of EP exchange phase, to avoid multiple messages being sent for node configuration exchange. Thread-based EP connection scheme is used for connecting various EPs. At the end of this step,

each node updates its *path repository* for communication to every other node in the cluster.

10.2.3 Communication Methodology for Multiple Interconnects

As mentioned in the background section, uDAPL allows an user to use RDMA for data transfer. One of the key requirements is that the user buffer be registered with the corresponding interconnect. Since our design supports multiple interconnects, for simplicity, we register the complete buffer with all interconnects. In addition, for the rendezvous protocol, completion notifications need to sent on all interconnects participating in data transfer to the communicating process. Presence of multiple paths also leads to *out-of-order* messages. MPI requires messages to be processed in order. Hence, we maintain out-of-order queues, and periodically poll on them.

As discussed in [33, 59], scheduling policies have a great impact on performance, when a combination of paths are available. Simple policies like *even striping*, *round robin*, *process binding* and *weighted striping* provide comparable performance for a combination of interconnects with similar peak bandwidth. However, these policies provide sub-optimal performance when different paths have different bandwidths. In our previous work, we have also shown that *adaptive striping* stands out the best candidate in such scenarios. Hence, we use this policy, so that our design leverages multiple networks in an optimal fashion, in addition to using them for failover. A *striping threshold* value is used, below which the primary network for communication is used. In the performance evaluation section, we have used *adaptive striping* policy by default, unless mentioned otherwise. We have used

InfiniBand as the primary path of communication, wherever possible, for messages below the striping threshold.

10.3 Design of Communications and Network Fault Tolerance Layer

In this section, we discuss the design of communications and network fault tolerance layer. We discuss various modules associated with this layer and their interactions. This is shown in more detail in the Figure 10.4. We begin with the error detection module.

10.3.1 Completion Filter and Error Detection Module

uDAPL library allows a user application to make work requests by posting send work requests or descriptors. The status of these requests can be determined by using the completion queue mechanism. As shown in Figure 10.4, completion notifications generated from the network are stored in the completion queue. uDAPL also provides completion notification interrupt to be generated for solicited work requests. However, this mechanism leads to increased latency, particularly for small messages. In our design, we use *polling* on the completion queue to determine the status of the work request. It is to be noted, that a completion queue entry (CQE) is generated, independent of success/failure in completing the work request. Upon receipt of a successful CQE, this module updates the weight(s) of different path(s) of communication to the communicating process, as shown in Figure 10.4. However, on receiving a failed CQE, associated error code in the CQE is used to determine the cause of the failure. We leverage this uDAPL capability to check the failure in completion of a send or a receive work request. The *remote*

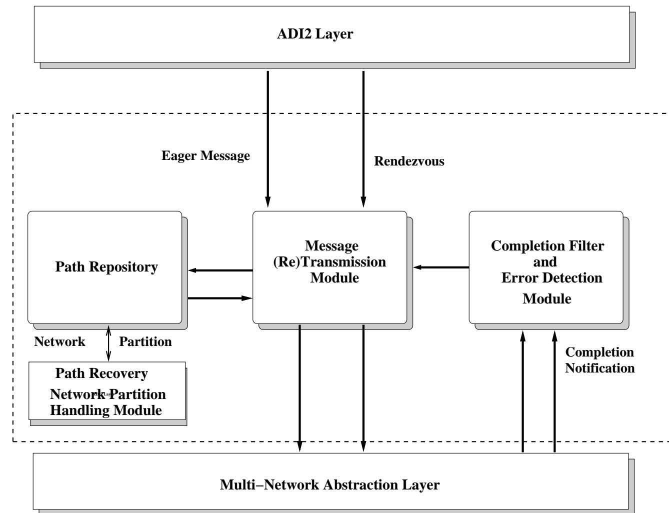


Figure 10.4: Communications and Network Fault Tolerance Layer

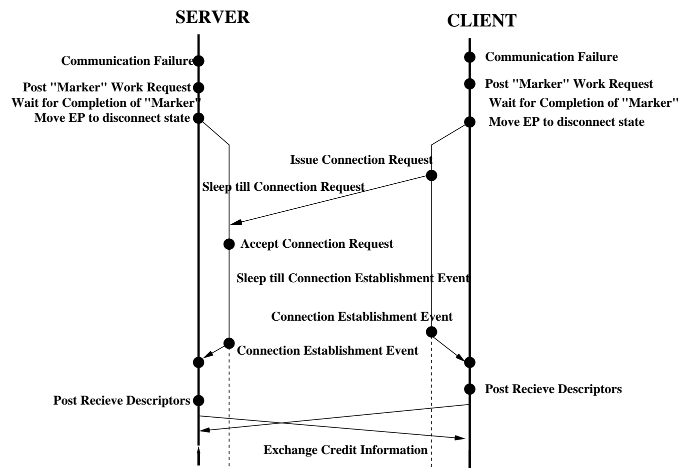


Figure 10.5: Communication Protocol For Recovery from Network Partitions and Previously Failed Paths

access error failure opcode shows the un-reachability of the *remote destination*. This failure implies that even after multiple re-tries by the Network Interface Card (NIC), the path could not be reached. Such a failure can also occur, when the *rkey* value for RDMA operation is wrong. However, in both cases, occurrence of even a single failure on an end point breaks the connection and all posted work requests (send or receive) result into error. The recovery mechanism of the broken EP is handled in the *Path Recovery and Network Partitioning Module*. Once the error is detected, the control is transferred to *message re-transmission module*.

10.3.2 Message (Re)Transmission Module

This module is activated upon receiving an input from the completion filter and error detection module or receiving an input from the ADI layer for message transmission. If the request is received from the ADI layer, the appropriate scheduling policy is used for message transmission. The interaction is further illustrated in Figure 10.4. Upon receipt of a failed CQE from completion filer and error detection module, the first step is to update the path repository, marking the associated communication path to the destination process *unavailable*. Upon receipt of a failed CQE with receive *opcode*, the corresponding buffer is simply released, however another receive descriptor is not posted, since the connection is already broken. As mentioned before, posting another work request on a broken connection results in error. When a CQE with failed send opcode is received, *path repository* is queried for the available paths to the destination rank. The return from the path repository can be *success* with a list of the available path(s) or a *failure* in case of network partition (it is not be noted that the sender may still

have communication paths to other processes). The failed send descriptor consists of information about the length of the work request. To post this descriptor to available paths, the length of each work request is adjusted in conjunction with scheduling policy and associated *lkey* for interconnect is used. In addition, if an RDMA operation is requested, the associated rkey is updated for data transfer.

10.3.3 Path Recovery and Network Partition Handling Module

The design mentioned upto now provides failover, when network paths fail and message re-transmission in such cases. However, network errors can be transient and this should not limit the application from re-using the corresponding paths upon recovery. In addition, an application should not abort in the presence of network failures, since the process state is intact. Long running applications should also be able to use the recovered paths, and be able to extract the best performance out of the System Area Network. This layer meets the above requirements by using an asynchronous thread based recovery mechanism.

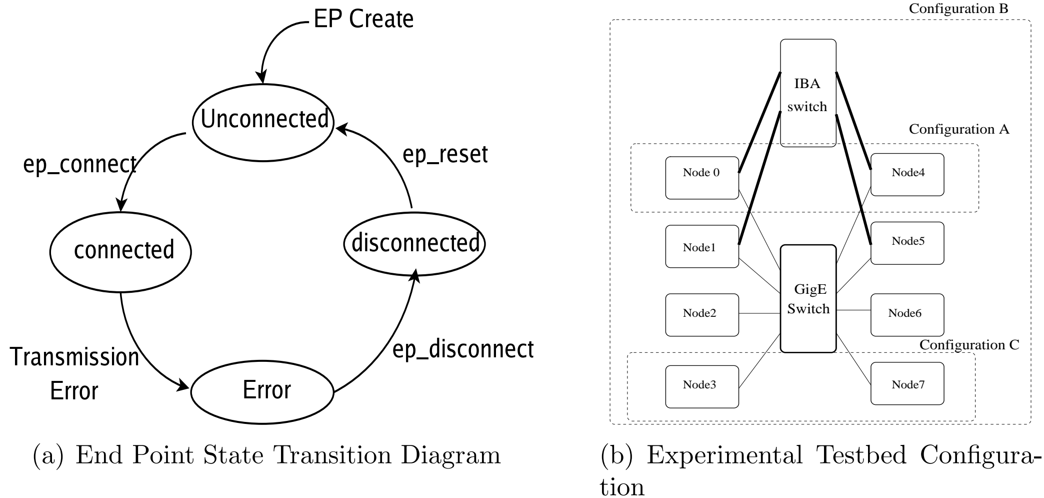
In order to facilitate this capability, the broken end point associated with the failed network path needs to be brought back to the connected state. This is further illustrated in Figure 10.6(a). The DAT specification mentions that an end point in an error state should not be moved to disconnected state at the discovery of first failure, else would result in loss of the previously posted work requests. Hence, in our design, we post a send work request called *marker*. Since Work requests always finish in order on the sender side, after receiving a CQE associated with the marker, the end point can be moved to the disconnected, followed by the unconnected state as shown in Figure 10.6(a).

The communication protocol for recovery from network partition is further illustrated in Figure 10.5. When a process receives the first communication failure, it initiates an asynchronous thread which initiates request(s) for bringing back the end points to the connected state. As mentioned in our previous work [12], each process acts as a server for processe(s) with higher MPI rank and sends connect requests only to processe(s) with lower rank. Since connection requests can possibly arrive at any point of time, the asynchronous server thread remains in *sleep* state during the program execution and wakes up only during connection request(s) from the client(s). Similarly, the client thread initiates request(s), goes to sleep and only activates, when the connection event(s) are generated. Once a client and server have received the connection events, each of their end points are in *connected* state. At this point, each of the processes post receive descriptors, and exchange the credit information by sending a *connect* message. Once the processes receive the message, they are ready for communication. Since these threads are in *sleep* state for most of the time during program execution, they incur little contention to the main thread.

10.4 Performance Evaluation with uDAPL Based MPI

In this section, we evaluate the performance of our design. We call our design *MN-uDAPL* and compare its performance with MVAPICH-0.9.7 for OSU Tests [42] and NAS Parallel Benchmarks [8]. Our Performance Evaluation is further divided into multiple cases:

- No network fault(s) occur during the application execution in the SAN. This evaluation helps us understand the performance improvement which can be



(a) End Point State Transition Diagram

(b) Experimental Testbed Configuration

Figure 10.6

achieved when there are multiple interconnects in the SAN, in homogeneous and heterogeneous environments.

- One or more network fault(s) occur during the application execution in the SAN. We evaluate the cases when a previously failed path recovers during the application execution, to the cases of network partitioning. This helps us understand the overhead incurred by network fault tolerance modules, when such faults occur.

We begin with a brief description of our experimental testbed.

10.4.1 Experimental Testbed

Figure 10.6(b) is a block diagram for our experimental testbed. This cluster consists of eight 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 Dual-Port 4x HCAs from Mellanox. The ServerWorks GC LE chipsets have two separate I/O bridges and three PCI-X 64-bit 133 MHz bus slots. The kernel version we used is Linux 2.6.9smp. The IBGD version is 1.8.2 and HCA firmware version is 3.3.2. The Front Side Bus (FSB) of each node runs at 533MHz. The physical memory is 2 GB of PC2100 DDR-SDRAM.

Four nodes in the cluster comprise of one InfiniBand(InfiniHost MT23108 Dual-Port 4x HCAs from Mellanox) and eight nodes comprise of Ammasso (Ammasso 1100 RDMA-enabled Gigabit-Ethernet Adapter) each. uDAPL libraries provided by Mellanox and Ammasso are used for performance evaluation.

10.4.2 Performance Evaluation on Configuration A

As shown in Figure 10.6(b), this configuration comprises of two nodes which have both IBA and GigE network interface cards.

Figure 10.7 shows the latency of small messages for different devices of MVAPICH, 0.9.7. Since the messages are small, only IBA device is used for communication. Messages above the striping threshold (256K) use adaptive striping for communication. In comparison to MVAPICH-0.9.7, uDAPL device, our MPI incurs negligible overhead. The overhead in latency, when compared to VAPI device is due to the absence of *inline functionality* in uDAPL library. This functionality allows data to be posted along with the descriptor, hence reducing the number of I/O bus transactions.

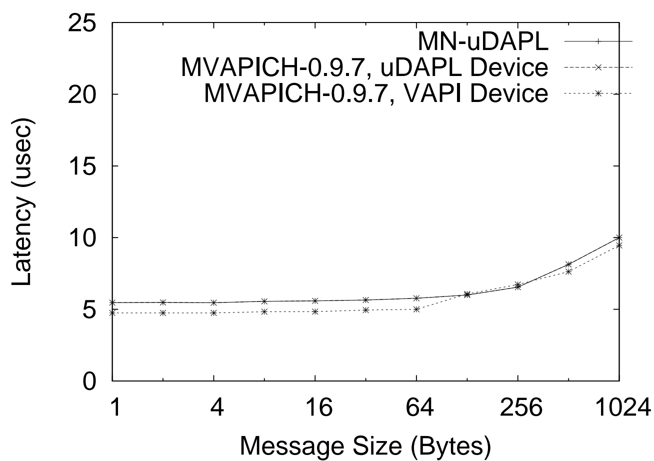


Figure 10.7: Latency Overhead for uDAPL and VAPI based MPI

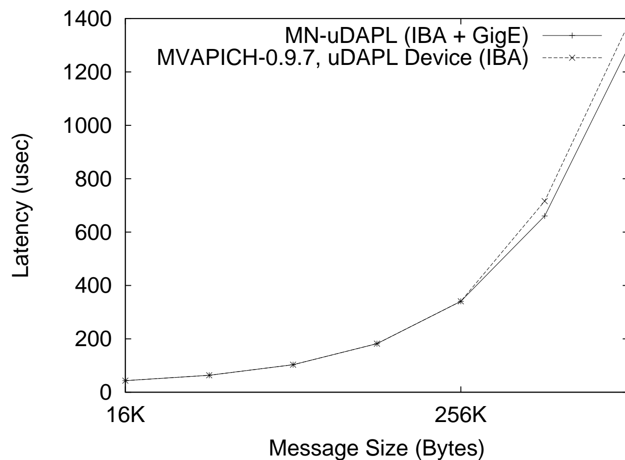


Figure 10.8: Large Message Latency

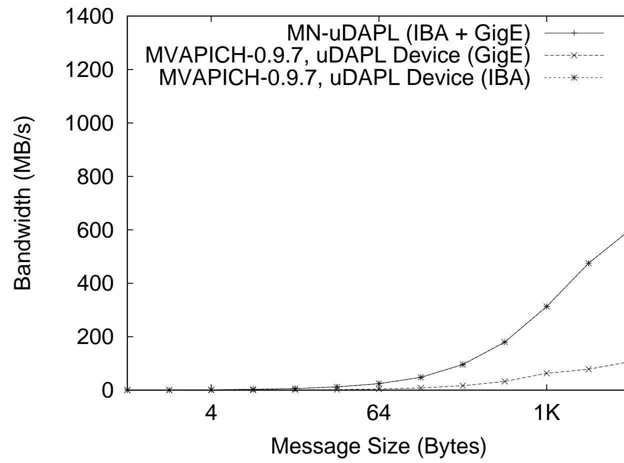


Figure 10.9: Uni-directional Bandwidth Comparison

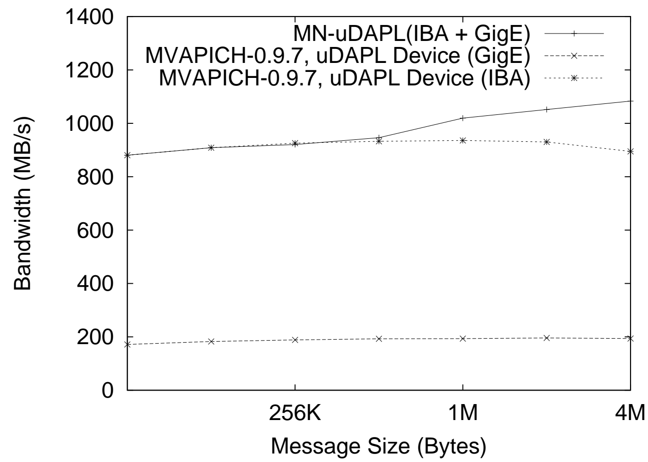


Figure 10.10: Bi-directional Bandwidth Comparison

Figure 10.8 shows the performance of latency for large messages. Messages above the striping threshold are able to be benefited by using the adaptive striping policy. For 512Kbyte message, the latency improves by almost 10%.

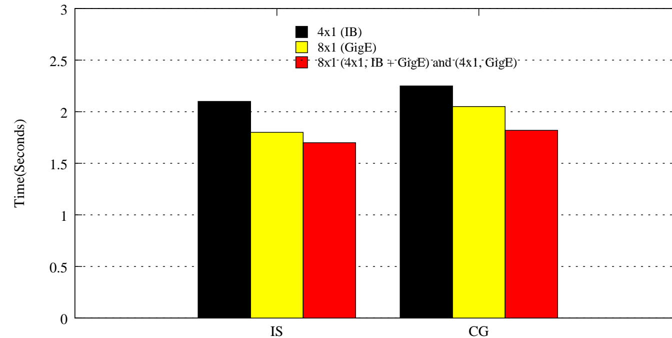


Figure 10.11: Performance Evaluation of IS and CG NAS Parallel Benchmarks, Class A

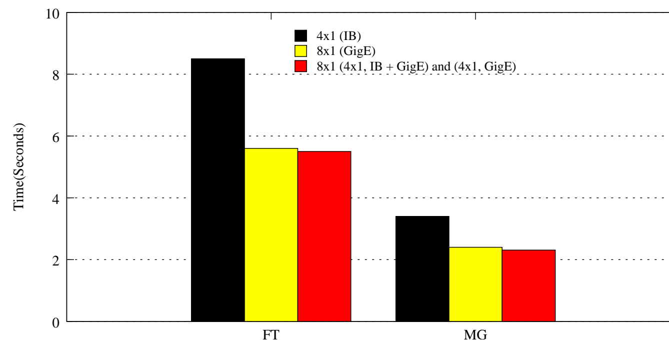


Figure 10.12: Performance Evaluation of FT and MG NAS Parallel Benchmarks, Class A

Figure 10.9 and 10.10 show the performance for OSU uni-directional and bi-directional bandwidth test. As explained above, MN-uDAPL uses only InfiniBand device for messages of size lesser than the striping threshold. Adaptive striping provides a peak uni-directional bandwidth of 963 MB/s compared to 880 MB/s for MVAPICH-0.9.7, uDAPL device(IBA) only. The GigE device can only provide

around 100 MB/s. Similarly, a performance improvement of 18% is seen for peak bandwidth in the bi-directional bandwidth test, which improves from 931 MB/s to 1095 MB/s.

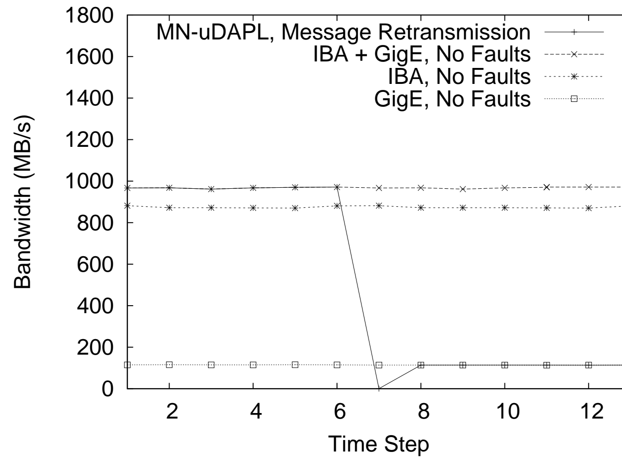


Figure 10.13: Uni-directional Bandwidth Comparison for Fault Tolerant Schemes, IBA Path Fails

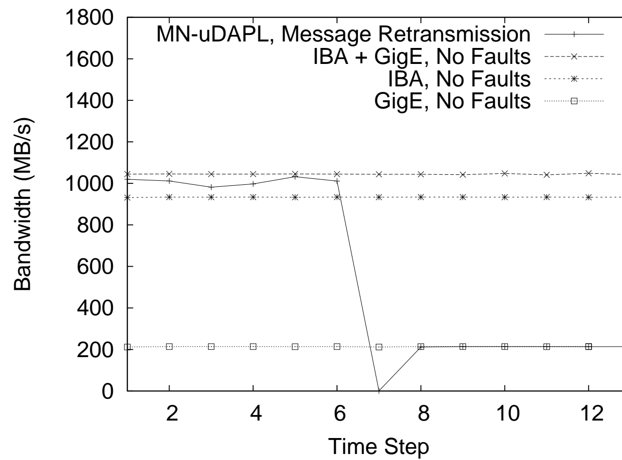


Figure 10.14: Bidirectional Bandwidth Comparison for Fault Tolerant Schemes, IBA Path Fails

10.4.3 Performance Evaluation on Configuration B

In this section, we evaluate the unified performance of the cluster, also the configuration B as shown in the Figure 10.6(b). We use MVAPICH 0.9.7, uDAPL device for evaluation and compare its performance with MN-uDAPL. Since MVAPICH-0.9.7 is capable of utilizing only one interface at a time, we evaluate it under two configurations for our cluster. In one configuration, it is able to utilize nodes with IBA cards only, and the other configuration can utilize nodes with GigE cards only. Figures 10.4.2 and 10.12 compare the performance of these configurations with MN-uDAPL, which is capable of handling this network heterogeneity in a unified manner. In Figure 10.4.2, we use CLASS A, IS and CG benchmarks. For IS, IBA only with 4 nodes takes 2.09 seconds, only GigE takes 1.90 seconds. MN-uDAPL is able to reduce the time taken to 1.75 seconds, which is an improvement of 8% from GigE only and 17% from IBA case only. Respective improvements of 20% and 9% are seen for the CG application kernel. Figure 10.12 shows the performance comparisons for FT and MG benchmarks. We notice that the application time does not improve much with respect to the network. However, a slight improvement in performance is shown by using MN-uDAPL than GigE device only.

10.4.4 Performance Evaluation with Network Faults

Figure 10.13, 10.14, 10.15 and 10.16 show the results for the cases when network fault occurs in the system. The comparisons are being shown for the message re-transmission scheme with the ideal case, when the same test is ran with no network faults. In order to show these results, we let the OSU Latency test report

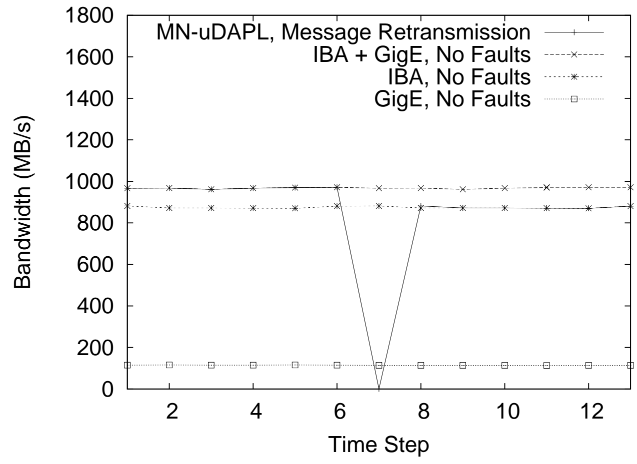


Figure 10.15: Uni-directional Bandwidth Comparison for Fault Tolerant Schemes, GigE Path Fails

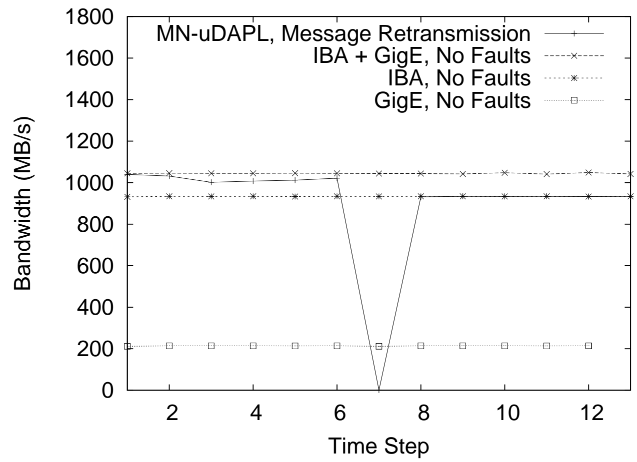


Figure 10.16: Bidirectional Bandwidth Comparison for Fault Tolerant Schemes, GigE Path Fails

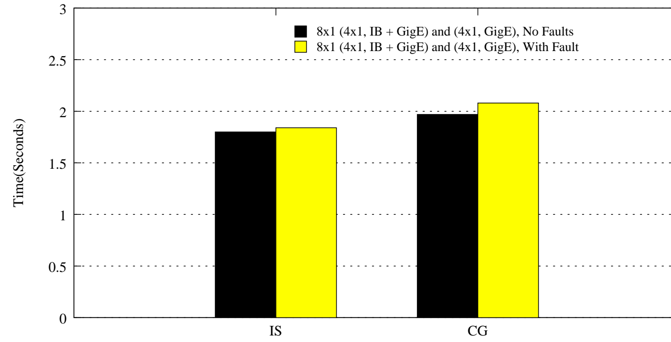


Figure 10.17: Performance Comparison on NAS Benchmarks, When the IBA Path Fails on First Message Transmission

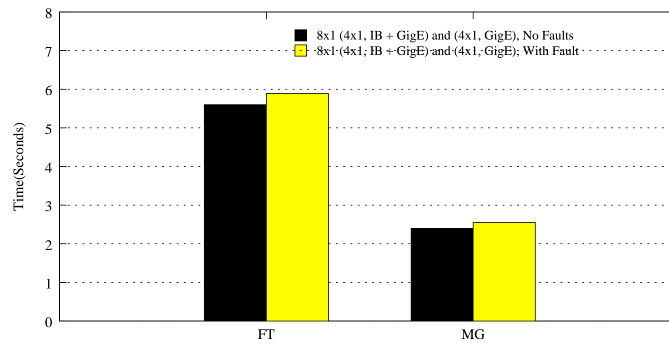


Figure 10.18: Performance Comparison on NAS Benchmarks, When the IBA Path Fails on First Message Transmission

bandwidth at each iteration for a large number of iterations for a message size of 1 MB. The point of failure is reported as the middle point on the x-axis.

Figure 10.13 and 10.14 show the results for uni-directional and bi-directional bandwidth, when IBA path fails during the communication. The message retransmission scheme achieves the peak bandwidth as shown in the previous sections. However, at the point of failure, the re-transmission scheme almost achieves no-bandwidth due to multiple re-transmissions which occur at this point, before the DMA engine concludes the un-reachability of the destination process, and puts a failed CQE into the corresponding interconnect's completion queue. At this point, only GigE path is available. As can be noted from the graphs, our scheme incurs no overhead in providing the peak bandwidth. Figure 10.15 and 10.16 show a similar trend, the difference being the failure of the GigE path.

Figure 10.4.3 and 10.18 present the results, when network paths fail at the beginning of the application itself. We notice from the figures that the performance degradation is negligible in comparison to the case 8x1 case, where only GigE is used for communication. This shows that the overhead of the message re-transmission module, generating an asynchronous thread for communication etc. incurs very low overhead on the communication performance.

Figure 10.19 and 10.20 present the results for uni-directional bandwidth when running experiments in the configuration A. At the point 8 in the graph, both GigE and the IBA path fail and hence a network partition occurs in the system. At this point, the application hangs and waits for one of the connection paths to come up. The path recovery and network partition handling module generates an asynchronous thread and waits for the *connection* events from other process. After

re-connection, the processes are able to achieve the peak uni-directional bandwidth which is achievable with GigE. At a later point, when the IBA path is available, we are able to achieve the peak bandwidth achievable in the presence of no-faults. Figure 10.19 shows a similar trend, however in this case the IBA path comes back earlier than the GigE path. However, in this case also we are able to achieve the peak uni-directional bandwidth achievable, similar to the ideal case.

10.5 Summary

In this chapter, we have designed a network fault tolerant MPI using uDAPL interface, making this design portable for existing and upcoming interconnects. Our design has provided failover to available paths, asynchronous recovery of the previous failed paths and recovery from network partitions without application restart. In addition, the design is able to handle network heterogeneity, making it suitable for the current state of the art clusters. To achieve these goals, we have designed low overhead *completion filter and error-detection*, *message (re)-transmission* and *path recovery and network partition handling* modules which perform completion filter and detection, (re)-transmission and recovery from network partitions respectively. We have implemented our design and evaluated it with micro-benchmarks and applications. Our performance evaluation have shown that the proposed design provides significant performance benefits to both homogeneous and heterogeneous clusters. Experiments also reveal that network fault tolerance modules incur very low overhead and provide optimal performance in wake of network failures for simple MPI micro-benchmarks and applications. In addition, in the absence of such failures, using a heterogeneous 8x1configuration of IBA and Ammasso-GigE, we

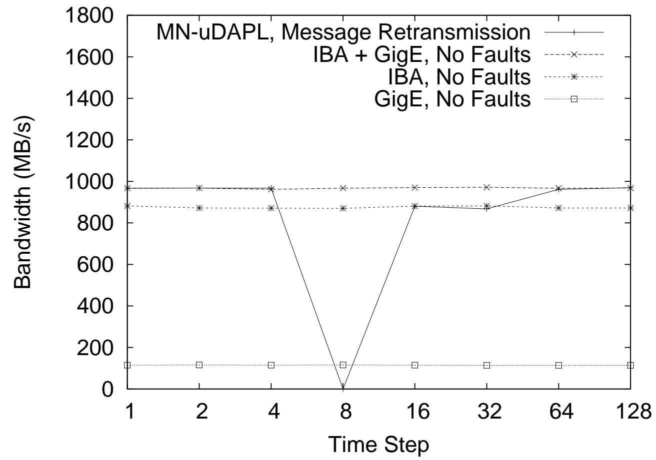


Figure 10.19: Uni-directional Bandwidth Comparison for Fault Tolerant Schemes with Network Partition, IBA Path Recovers First

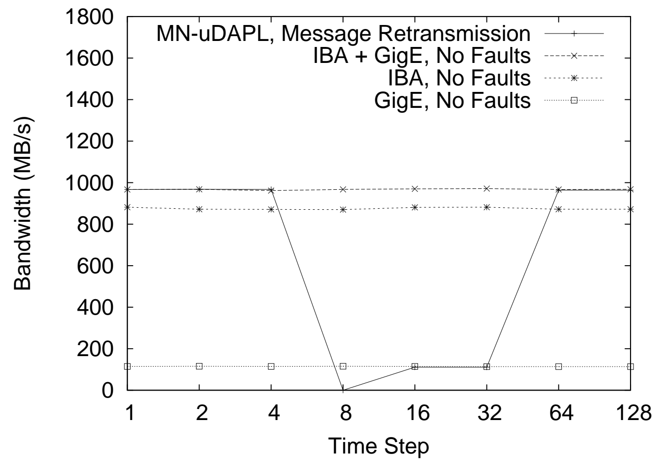


Figure 10.20: Uni-directional Bandwidth Comparison for Fault Tolerant Schemes with Network Partition, GigE Path Recovers First

have been able to improve the performance of NAS Parallel Benchmarks by 10-15% for different benchmarks. For simple micro-benchmarks, we have been able to improve the throughput by 15-20% for uni-directional and bi-directional bandwidth tests.

CHAPTER 11

OPEN SOURCE SOFTWARE RELEASE AND ADOPTION

A majority of the work presented in this dissertation has been incorporated in an open source manner with OSU MPI over InfiniBand, MVAPICH and MVAPICH2, MPI-1 and MPI-2 versions of the MPI libraries. The duration of this work has spanned multiple releases of MVAPICH and MVAPICH2 (0.9.5 - 1.0 for MVAPICH) and (0.9.5 - 1.0 for MVAPICH2). The results presented in this dissertation have enabled the community to design ultra scale InfiniBand clusters with resilience to network faults and hot-spot avoidance.

MVAPICH/MVAPICH2 support many network interfaces including OpenFabrics, uDAPL and VAPI. Much of the initial work has become available with VAPI, the Verbs API from Mellanox. As the community has moved towards Open Source API for InfiniBand, most of the work has become available with OpenFabrics. In addition to InfiniBand, MVAPICH also supports 10GigE-iWARP with OpenFabrics interface and any adapter which supports RDMA with the uDAPL interface. It also supports multiple architectures including popular 32-bit and 64-bit architectures.

Since its release in 2002, more than 580 computing sites and organizations have downloaded this software. In addition, nearly every InfiniBand vendor and the Open Source OpenFabrics stack includes this software in their packages. Our software has been used on some of the most powerful computers, as ranked by Top500 [6]. Examples from the November 2007 rankings include $3rd$ 14336-core Clovertown cluster at New Mexico Computing Applications Center, $22nd$, 5848-core Dell PowerEdge (Intel EM64T) cluster at Texas Advanced Computing Center/Univ. of Texas (TACC), $29th$, 9216-core Appro Quad Opteron dual Core at Lawrence Livermore National Laboratory and $108th$, 2200-processors Apple Xserve 2.3 GHz cluster at Virginia Tech. More information about our software release can be found at MVAPICH Homepage [42].

CHAPTER 12

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

12.1 Summary of Research Contributions

The research in this dissertation aims towards providing high performance and network fault tolerant MPI with multi-pathing over InfiniBand. As discussed in the previous chapter, a majority of these designs are available with MVAPICH and MVAPICH2 software package. These designs are likely to benefit the existing and upcoming ultra-scale InfiniBand clusters.

Although, MPI is the primary programming model for discussion in this dissertation, a majority of the ideas are applicable to other programming models, including X10 and UPC, which are being projected as the programming models for next generation peta-flop systems. Other middlewares including parallel file systems are also likely to benefit from designs presented in this dissertation. Hence, we can conclude that the contribution of this dissertation to this community is significant. Following sections provide a more detailed summary of the research contributions:

12.1.1 Efficient MPI-1 Design for Multi-Rail InfiniBand Clusters

In chapter 4, we presented an in-depth study of designing high performance multi-rail InfiniBand clusters for MPI-1 primitives. We discussed various ways of setting up multi-rail networks with InfiniBand and proposed a unified MPI-1 design that can support all these approaches. By taking advantage of RDMA operations in InfiniBand and integrating the multi-rail design with MPI communication protocols, our design supported multi-rail networks with very low overhead. Our performance results show that the multi-rail MPI can significantly improve MPI communication performance. With a two-rail InfiniBand network, we achieved almost twice the bandwidth and half the latency for large messages compared with the original MPI. The multi-rail MPI design also significantly reduced the communication time as well as the running time for bandwidth-bound applications.

12.1.2 Supporting MPI One-Sided Communication with Multi-Rail InfiniBand Clusters

In chapter 5, we presented the challenges *(Multiple synchronization messages, handling multiple HCAs, scheduling policies, ordering relaxation)* associated with designing MPI-2 one-sided communication over multi-rail InfiniBand networks. We implemented our design and presented the performance evaluation for micro-benchmarks. We observed that multi-rail InfiniBand clusters can significantly improve the performance for one-sided communication. Using a two rail cluster, we achieved almost doubled the throughput and reduced the latency to half with *MPI_Put* and *MPI_Get* operations for large messages. We also observed that

reordering policy can significantly improve the performance for communication patterns with a mix of one-sided operations.

12.1.3 Improving Performance with IBM 12x InfiniBand Architecture

In chapter 6, we focused on designing an MPI substrate for IBM 12x InfiniBand Architecture. We discussed with the introduction of overall design, and presented the limitations of previously proposed designs in achieving the peak performance of IBM 12x InfiniBand architecture. We presented the need for re-visiting the scheduling policies, depending upon the communication pattern in the application. We presented communication marker module, which resides in the ADI layer and differentiates between communication patterns. We achieved a peak unidirectional bandwidth of 2745 MB/s and bidirectional bandwidth of 5362 MB/s. We concluded that none of the previously proposed policies alone provides optimal performance for these communication patterns. Using NAS Parallel Benchmarks, we saw an improvement of 7-13% in execution time along with a signification improvement in collective communication using Intel Benchmark suite.

12.1.4 Hot-Spot Avoidance with Multi-Pathing Using LMC Mechanism

Large scale InfiniBand clusters are becoming increasingly popular, as reflected by the TOP 500 [6] Supercomputer rankings. At the same time, *fat tree* [31] has become a popular interconnection topology for these clusters, since it allows multiple paths to be available in between a pair of nodes. However, even with fat

tree, hot-spots may occur in the network depending upon the route configuration between end nodes and communication patterns in the application.

In chapter 7, we presented the design for an MPI functionality, Hot-Spot Avoidance with MVAPICH (HSAM) which provides hot-spot avoidance for different communication patterns, without apriori knowledge of the pattern. We leveraged LMC (LID Mask Count) mechanism of InfiniBand to create multiple paths in the network, and studied its efficiency in creation of contention free routes. We also presented the design issues (scheduling policies, selecting number of paths, scalability aspects) associated with our MPI functionality. We implemented our design and evaluated it with collective communication and MPI applications. On an InfiniBand cluster with 64 processes, we observed an average improvement of 23% for displaced ring communication pattern among processes. For collective operations like MPI All-to-all Personalized and MPI Reduce Scatter, we observed an improvement of 27% and 19% respectively. Our evaluation with NAS Parallel Benchmarks [8] shows an improvement of 6-9% in execution time for the FT Benchmark, with class B and class C size using 32-64 processes for evaluation. For other NAS Parallel Benchmarks, we did not see a degradation in performance compared to the original design.

12.1.5 Enhanced Design for Avoiding Hot-Spots with Better Network Paths Utilization

In chapter 8, we addressed the limitations of the HSAM design presented in chapter 7. We presented a Hot-Spot Avoidance Layer (HSAL) with InfiniBand to

provide path bandwidth estimation completion filter modules for two-sided communication semantics. To efficiently utilize physically independent paths, we proposed a novel scheduling policy, which performs Batch-based Striping and Sorting (BSS) during the application execution to adaptively eliminate the path(s) with low bandwidth. Using MPI_Alltoall, we achieved an improvement of 27% and 32% in latency with different BSS policy configurations compared to the best configuration of the HSAM scheme on 32 and 64 processes, respectively.

12.1.6 Network Fault Tolerance Using Automatic Path Migration

In chapter 9, we designed a set of modules; *alternate path specification module*, *path loading request module* and *path migration module*, which work together for providing network fault tolerance for user level applications. We integrated these modules for simple micro-benchmarks at the Verbs Layer, the user access layer for InfiniBand, and study the impact of different state transitions associated with APM. We also integrated these modules at the MPI [38, 39] (Message Passing Interface) layer to provide network fault tolerance for MPI applications. Our performance evaluation has shown that APM incurs negligible overhead in the absence of faults in the system. For MPI applications executing for reasonably long time, APM caused negligible overhead in the presence of network faults. For Class B FT and LU NAS Parallel Benchmarks with 8 processes, the degradation was around 5-7% in the presence of network faults.

12.1.7 Software Based Network Fault Tolerance on Clusters with uDAPL Interface

In chapter 10, we designed a network fault tolerant MPI using uDAPL interface, making this design portable for existing and upcoming interconnects. Our design provided failover to available paths, asynchronous recovery of the previous failed paths and recovery from network partitions without application restart. In addition, the design was able to handle network heterogeneity, making it suitable for the current state of the art clusters. To achieve these goals, we designed a set of low overhead modules *completion filter and error-detection*, *message (re)-transmission* and *path recovery and network partition handling* which performed completion filter and detection, (re)-transmission and recovery from network partitions respectively. We implemented our design and evaluated it with micro-benchmarks and applications. Our performance evaluation has shown that the proposed design provides significant performance benefits to both homogeneous and heterogeneous clusters. Experiments also revealed that network fault tolerance modules incur very low overhead and provide optimal performance in the wake of network failures for simple MPI micro-benchmarks and applications. In addition, in the absence of such failures, using a heterogeneous 8x1configuration of IBA and Ammasso-GigE, we were able to improve the performance of NAS Parallel Benchmarks by 10-15% for different benchmarks. For simple micro-benchmarks, we were able to improve the throughput by 15-20% for uni-directional and bi-directional bandwidth tests. The proposed design was generic and capable of supporting any interconnect with uDAPL interface.

12.2 Future Research Directions

The high performance and rich features of InfiniBand has made it an attractive solution for designing various programming models, including MPI for existing and upcoming large scale InfiniBand clusters. In this dissertation, we have demonstrated that it is possible to design and implement an efficient communications and network fault tolerance layer for MPI over InfiniBand. We have discussed various design issues including scheduling policies, efficient data transfer for one-sided communication with multi-pathing support, network fault detection and recovery from failures, including network partitions. However, there are still many research topics, which can be pursued further. In the upcoming sections, we describe them in brief.

- **Designing Efficient MPI with Congestion Control** In the previous chapters, we have studied various designs for avoiding hot-spot avoidance with path bandwidth estimation [56]. However, our design does not provide solutions for congestion control. InfiniBand has proposed mechanisms for congestion notification at the source, sink and in the network for marking packets with congestion. Various researchers have focused on providing efficient response functions in presence of congestion for high speed interconnects like InfiniBand. However, these mechanisms have not become available in current generation InfiniBand products. Studying the impact of congestion control mechanisms with hot-spot avoidance mechanisms can be beneficial for upcoming ultra-scale InfiniBand clusters.

- **Designing Software Based Network Fault Tolerance Layer with APM** In previous chapters, we presented the solution for designing network fault detection and recovery with uDAPL support and user-transparent failover with Automatic Path Migration. However, there are mechanisms provided by InfiniBand, which can simplify the recovery protocol. As an example, the shared receive queue mechanism can be used with APM for a combination of hardware-software network fault tolerance. The software layer is triggered only in cases the APM does not complete successfully.

- **Leveraging Adaptive Routing Mechanisms with InfiniBand Architecture** Recently, Mellanox [3] has announced the support for adaptive routing with InfiniBand architecture and a 36-port crossbar silicon. Leveraging the ConnectX architecture [3] with support for adaptive routing would alleviate hot-spots in the network, in a user-transparent fashion. Studying the impact of user-level hot-spot avoidance mechanisms with adaptive routing would provide key insights with respect to efficacy of different mechanisms, along with the congestion control mechanisms, proposed recently by InfiniBand community.

BIBLIOGRAPHY

[1] Chelsio Communications. http://www.chelsio.com/.

[2] Intel MPI Benchmark. http://www.intel.com/cd/software/products/.

[3] Mellanox Technologies. http://www.mellanox.com/.

[4] Parallel Spectral Transform Shallow Water Model. http://www.csm.ornl.gov/chammp/pstswm/.

[5] The Linux Kernel Archives. http://www.kernel.org/.

[6] TOP 500 Supercomputer Sites. http://www.top500.org.

[7] H. Adiseshu, G. M. Parulkar, and G. Varghese. A Reliable and Scalable Striping Protocol. In *SIGCOMM*, pages 131–141, 1996.

[8] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, D. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks. In *The International Journal of Supercomputer Applications*, number 3, pages 63–73, 1991.

[9] M. Banikazemi, V. Moorthy, L. Herger, D. K. Panda, and B. Abali. Efficient Virtual Interface Architecture Support for the IBM SP Switch-Connected NT Clusters. In *Int'l Parallel and Distributed Processing Symposium (IPDPS '00)*, pages 33–42, May 2000.

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

[11] R. K. Brunner, J. C. Phillips, and L. V. Kale. Scalable Molecular Dynamics for Large Biomolecular Systems. In *SuperComputing (SC'2000)*, pages 67–85, 2000.

[12] L. Chai, R. Noronha, P. Gupta, G. Brown, and D. K. Panda. Designing a Portable MPI-2 over Modern Interconnects Using uDAPL Interface. In *EuroPVM/MPI*, pages 200–208, 2005.

[13] L. Chai, R. Noronha, and D. K. Panda. MPI over uDAPL: Can High Performance and Portability Exist Across Architectures?. In *Cluster Computing and Grid*, pages 19–26, 2006.

[14] S. Coll, E. Frachtenberg, F. Petrini, A. Hoisie, and L. Gurvits. Using Multirail Networks in High-Performance Clusters. In *Cluster*, pages 16–24, 2001.

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

[16] DAT Collaborative. uDAPL: User Direct Access Programming Library Version 1.2. http://www.datcollaborative.org/udapl.html, July 2004.

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

[18] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[19] W. Gropp and E. Lusk. A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.

[20] M. Gusat, D. Craddock, W. Denzel, T. Engbersen, N. Ni, G. Pfister, W. Rooney, and José Duato. Congestion Control in InfiniBand Networks. In *Hot Interconnects*, pages 158–159, 2005.

[21] A. Hari, G. Varghese, and G. Parulkar. An Architecture for Packet-Striping Protocols. *ACM Transactions on Computer Systems*, 17(4):249–287, 1999.

[22] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of High Performance MVAPICH2: MPI2 over InfiniBand. In *Cluster Computing and Grid*, pages 43–48, 2006.

[23] W. Huang, G. Santhanaraman, H.-W. Jin, and D. K. Panda. Scheduling of MPI-2 One Sided Operations over InfiniBand. In *International Symposium on Parallel and Distributed Processing*, 2005.

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

[25] Ammasso Incorporation. http://www.ammasso.com/.

[26] InfiniBand Trade Association. InfiniBand Architecture Specification, Release 1.2, October 2004.

[27] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[28] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. In *SIGCOMM*, pages 259–268, 1993.

[29] E. Kim, K. H. Yum, C. R. Das, M. S. Yousif, and J. Duato. Exploring IBA Design Space for Improved Performance. *IEEE Transactions on Parallel and Distributed Systems*, 18(4):498–510, 2007.

[30] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Redwood City, CA, USA, 1994.

[31] C. E. Leiserson. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers*, 34(10):892–901, 1985.

[32] J. Liu, B. Chandrasekaran, J. Wu, W. Jiang, S. P. Kini, W. Yu, D. Buntinas, P. Wyckoff, and D. K. Panda. Performance Comparison of MPI Implementations over InfiniBand, Myrinet and Quadrics. In *SuperComputing*, pages 58–69, 2003.

[33] J. Liu, A. Vishnu, and D. K. Panda. Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation. In *SuperComputing*, pages 33–44, 2004.

[34] J. Liu, J. Wu, S. P. Kini, P. Wyckoff, and D. K. Panda. High Performance RDMA-based MPI Implementation over InfiniBand. In *International Conference on SuperComputing*, pages 295–304, 2003.

[35] P. Lopez, J. Flich, and J. Duato. Deadlock-Free Routing in InfiniBand through Destination Renaming. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 427–436, 2001.

[36] A. R. Mamidala, J. Liu, and D. K. Panda. Efficient Barrier and Allreduce on Infiniband Clusters using Multicast and Adaptive Algorithms. In *Cluster*, pages 135–144, 2004.

[37] J. C. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato. Supporting Fully Adaptive Routing in InfiniBand Networks. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

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

[39] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, Jul 1997.

[40] T. Nachiondo, J. Flich, and J. Duato. Destination-Based HoL Blocking Elimination. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 213–222. IEEE Computer Society, 2006.

[41] S. Narravula, A. Mamidala, A. Vishnu, G. Santhanaraman, and D. K. Panda. High Performance MPI over iWARP: Early Experiences. In *International Conference on Parallel Processing*, 2007.

[42] Network-Based Computing Laboratory. MVAPICH/MVAPICH2: MPI-1/MPI-2 for InfiniBand and iWARP with OpenFabrics. http://mvapich.cse.ohio-state.edu/.

[43] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, and W. Su. Myrinet: A Gigabit-per-second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[44] OpenFabrics Organization. http://www.openib.org/.

[45] OpenMPI. Open Source High Performance Computing. http://www.open-mpi.org/.

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

[47] D. A. Patterson, D. E. Culler, and T. E. Anderson. A Case for NOW (Networks of Workstations). In *Principles of Distributed Computing*, pages 17–28, 1995.

[48] 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.

[49] J. C. Sancho and A. Robles. Improving the Up*/Down* Routing Scheme for Networks of Workstations. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 882–889, 2000.

[50] J. C. Sancho, A. Robles, and J. Duato. Effective Strategy to Compute Forwarding Tables for InfiniBand Networks. In *ICPP '02: Proceedings of the 2001 International Conference on Parallel Processing*, pages 48–60, 2001.

[51] J. C. Sancho, A. Robles, J. Flich, P. Lopez, and J. Duato. Effective Methodology for Deadlock-Free Minimal Routing in InfiniBand Networks. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, page 409, 2002.

[52] J. R. Santos, Y. Turner, and G. J. Janakiraman. End-to-End Congestion Control for InfiniBand. In *InfoComm*, pages 1123–1133, 2003.

[53] L. C. Stewart, D. Gingold, J. Leonard, and P. Watkins. RDMA in the SiCortex Cluster Systems. In *EuroPVM/MPI*, pages 260–271, 2007.

[54] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-Down Cache: A Virtual Memory Management Technique for Zero-Copy Communication. In *IPPS/SPDP*, pages 308–314, 1998.

[55] A. Vishnu, B. Benton, and D. K. Panda. High Performance MPI on IBM 12x InfiniBand Architecture. In *International Workshop on High-Level Parallel Programming Models and Supportive Environments, held in conjunction with IPDPS '07 (HIPS'07)*, 2007.

[56] A. Vishnu, M. J. Koop, A. Moody, A. R. Mamidala, S. Narravula, and D. K. Panda. Hot-Spot Avoidance With Multi-Pathing Over InfiniBand: An MPI Perspective. In *Cluster Computing and Grid*, pages 479–486, 2007.

[57] A. Vishnu, A. Mamidala, S. Narravula, and D. K. Panda. Automatic Path Migration over InfiniBand: Early Experiences. In *Proceedings of Third International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, March 2007.

[58] A. Vishnu, A. R. Mamidala, H.-W. Jin, and D. K. Panda. Performance Modeling of Subnet Management on Fat Tree InfiniBand Networks using OpenSM. In *Proceedings of First International Workshop on System Management Techniques, Processes, and Services, held in conjunction with IPDPS'07*, 2005.

[59] A. Vishnu, G. Santhanaraman, W. Huang, H.-W. Jin, and D. K. Panda. Supporting MPI-2 One Sided Communication on Multi-rail InfiniBand Clusters: Design Challenges and Performance Benefits. In *International Conference on High Performance Computing*, pages 137–147, 2005.

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

[61] H. Wang and K. Shin. Refined Design of Random Early Detection Gateways. *Proceedings of IEEE GLOBECOM*, 1999.

[62] S. Yan, G. Min, and I. Awan. An Enhanced Congestion Control Mechanism in InfiniBand Networks for High Performance Computing Systems. *Advanced Information Networking and Applications*, 1:845–850, 2006.