Topology-Aware MPI Communication and Scheduling for High Performance Computing Systems

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Hari Subramoni, B.Tech.

Graduate Program in Department of Computer Science and Engineering

The Ohio State University

2013

Dissertation Committee:

Dhabaleswar K. Panda, Advisor

P. Sadayappan

R. Teodorescu

K. Tomko

Abstract

Most of the traditional High End Computing (HEC) applications and current petascale applications are written using the Message Passing Interface (MPI) programming model. Consequently, MPI communication primitives (both point to point and collectives) are extensively used across various scientific and HEC applications. The large-scale HEC systems on which these applications run, by necessity, are designed with multiple layers of switches with different topologies like fat-trees (with different kinds of over-subscription), meshes, torus, etc. Hence, the performance of an MPI library, and in turn the applications, is heavily dependent upon how the MPI library has been designed and optimized to take the system architecture (processor, memory, network interface, and network topology) into account. In addition, parallel jobs are typically submitted to such systems through schedulers (such as PBS and SLURM). Currently, most schedulers do not have the intelligence to allocate compute nodes to MPI tasks based on the underlying topology of the system and the communication requirements of the applications. Thus, the performance and scalability of a parallel application can suffer (even using the best MPI library) if topology-aware scheduling is not employed. Moreover, the placement of logical MPI ranks on a supercomputing system can significantly affect overall application performance. A naive task assignment can result in poor locality of communication. Thus, it is important to design optimal mapping schemes with topology information to improve the overall application performance and scalability. It is also critical for users of High Performance Computing

(HPC) installations to clearly understand the impact IB network topology can have on the performance of HPC applications. However, no currently existing tool allows users of such large scale clusters to analyze and to visualize the communication pattern of their MPI based HPC applications in a network topology-aware manner.

This work addresses several of these critical issues in the MVAPICH2 communication library. It exposes the network topology information to MPI applications, job schedulers and users through an efficient and flexible topology management API called the Topology Information Interface. It proposes the design of network topology aware communication schemes for multiple collective (Scatter, Gather, Broadcast, Alltoall) operations. It proposes a communication model to analyze the communication costs involved in collective operations on large scale supercomputing systems and uses it to model the costs involved in the Gather and Scatter collective operations. It studies the various alternatives available to a middleware designer for designing network topology and speed aware broadcast algorithms. The thesis also studies the challenges involved in designing network topology-aware algorithms for network intensive Alltoall collective operation and propose multiple schemes to design a network-topology-aware Alltoall primitive. It carefully analyzes the performance benefits of these schemes and evaluate their trade-offs across varying application characteristics. The thesis also proposes the design of a network topology-aware MPI communication library capable of leveraging the topology information to improve communication performance of point-to-point operations through network topology-aware placement of processes. It describes the design of network topology-aware plugin for the SLURM job scheduler to make scheduling decisions and allocate compute resources as well as place processes in a network topology-aware manner. It also describes the design

of a network topology-aware, scalable, low-overhead profiler to analyze the impact of our schemes on the communication pattern microbenchmarks and end applications.

The designs proposed in this thesis have been successfully tested at up to 4,096 processes on the Stampede supercomputing system at TACC. We observe up to 14% improvement in the latency of the broadcast operation using our proposed topology-aware scheme over the default scheme at the micro-benchmark level for 1,024 processes. The topology-aware point-to-point communication and process placement scheme is able to improve the performance the *MILC* application up to 6% and 15% improvement in total execution time on 1,024 cores of Hyperion and 2,048 cores of Ranger, respectively. We also observe that our network topology-aware communication schedules for Alltoall is able to significantly reduce the amount of network contention observed during the Alltoall / FFT operations. It is also able to deliver up to 12% improvement in the communication time of P3DFFT at 4,096 processes on Stampede. The proposed network topology-aware plugin for SLURM is able to improve the throughput of a 512 core cluster by up to 8%.

To my family, friends, and mentors.

Acknowledgments

This work was made possible through the love and support of several people who stood by me through the many years of my doctoral work. I would like to take this opportunity to thank all of them.

My extended family - my father, Subramoni, who has always given me complete freedom; my mother, Saraswathy, who has given me the courage to venture forth; my in-laws, Dr. Padmanabhan and Mrs. Girija Padmanabhan, for their invaluable gift and my loving brothers, Krishnan and Vignesh.

My wife, Kripa, for her love, support and understanding. I admire and respect her for the many qualities she possesses, particularly the great courage and presence of mind she has at the most trying of times. She is a joy to be around.

My advisor, Dr. Dhabaleswar K. Panda for his guidance and support throughout my doctoral program. I have been able to grow, both personally and professionally, through my association with him. Even after knowing him for six years, I am still amazed by the energy and commitment he has towards research.

My collaborators. I would like to thank all my collaborators: Adam Moody, Karen Tomko, Jerome Vienne, Devendar Bureddy, Karl Schulz, Bill Barth, Jeff Keasler and Sayantan Sur who have been instrumental in several publications that constitute this thesis.

My friends - Karthikeyan "Maappi" Subramanian, Sudhaa Gnanadesikan, Karthikeyan "Mark" Ramesh, Balaji "Hero" Sampathnarayanan, and Vignesh "Viggu" Ravi, who have

seen and supported me through the highs and lows of my PhD life. I am very happy to have met and become friends with Krishna “Pipps” Kandalla, Sreeram “Merry” Potluri and Jithin “Jitty” Jose. This work would remain incomplete without their support and contribution.

I would also like to thank all my colleagues who have helped me in one way or another throughout my graduate studies.

Above all, I would like to thank the Almighty for making all this possible.

Vita

2000-2004 ..... B.Tech., Computer Science and Engineering, University of Kerala, India

2004-2006 ..... Software Engineer, Integrated SoftTech Solutions, India

2006-2007 ..... Member Technical Staff, Force10 Networks, India

2007-Present ..... Ph.D., Computer Science and Engineering, The Ohio State University, U.S.A

2008-Present ..... Graduate Research Associate, The Ohio State University, U.S.A

2009 ..... Summer Research Intern, IBM T. J. Watson Research Center, U.S.A

2011 ..... Summer Research Intern, Lawrence Livermore National Laboratory, U.S.A

Publications

H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody and D. K. Panda, Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes, In *Int'l Conference on Supercomputing (SC '12)*, November 2012. ***Best Paper and Best Student Paper Finalist***

H. Subramoni, J. Vienne and D. K. Panda, A Scalable InfiniBand Network-Topology-Aware Performance Analysis Tool for MPI, In *Int'l Workshop on Productivity and Performance (Proper '12), held in conjunction with EuroPar, August 2012.* , August 2012.

H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. McLay, K. Schulz and D. K. Panda, Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms for InfiniBand Clusters, In *IEEE Cluster '11*, September 2011.

K. Kandalla, H. Subramoni, A. Vishnu and D. K. Panda, Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters: Case Studies with

Scatter and Gather, In *The 10th Workshop on Communication Architecture for Clusters (CAC '10)*, April 2010.

K. Hamidouche, S. Potluri, H. Subramoni, K. Kandalla and D. K. Panda MIC-RO: Enabling Efficient Remote Offload on Heterogeneous Many Integrated Core (MIC) Clusters with InfiniBand, In *Int'l Conference on Supercomputing (ICS '13)*, June 2013.

S. Potluri, D. Bureddy, H. Wang, H. Subramoni and D. K. Panda Extending OpenSHMEM for GPU Computing, In *Int'l Parallel and Distributed Processing Symposium (IPDPS '13)*, May 2013.

N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy and D. K. Panda, High Performance RDMA-Based Design of HDFS over InfiniBand, In *Int'l Conference on Supercomputing (SC '12)*, November 2012.

R. Rajachandrasekar, J. Jaswani, H. Subramoni and D. K. Panda, Minimizing Network Contention in InfiniBand Clusters with a QoS-Aware Data-Staging Framework, In *IEEE Cluster (Cluster '12)*, September 2012.

K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne and D. K. Panda, Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers, In *Int'l Parallel and Distributed Processing Symposium (IPDPS '12)*, May 2012.

S. P. Raikar, H. Subramoni, K. Kandalla, J. Vienne and D. K. Panda, Designing Network Failover and Recovery in MPI for Multi-Rail InfiniBand Clusters, In *Int'l Workshop on System Management Techniques, Processes, and Services (SMTPS), in conjunction with Int'l Parallel and Distributed Processing Symposium (IPDPS '12)*, May 2012.

J. Jose, H. Subramoni, K. Kandalla, M. W. Rahman, H. Wang, S. Narravula and D. K. Panda, Scalable Memcached design for InfiniBand Clusters using Hybrid Transports, In *Int'l Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2012)*, May 2012.

S. Sur, S. Potluri, K. Kandalla, H. Subramoni, K. Tomko, D. K. Panda, Co-Designing MPI Library and Applications for InfiniBand Clusters, In *Computer, 06 IEEE Computer Society Digital Library* September 2011.

D. K. Panda, S. Sur, H. Subramoni and K. Kandalla Network Support for Collective Communication, In *Encyclopedia of Parallel Computing*, September 2011.

J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. W. Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur and D. K. Panda, Memcached Design on High Performance RDMA Capable Interconnects, In *Int'l Conference on Parallel Processing (ICPP '11),* September 2011.

N. Dandapanthula, H. Subramoni, J. Vienne, K. Kandalla, S. Sur, D. K. Panda, and R. Brightwell, INAM - A Scalable InfiniBand Network Analysis and Monitoring Tool, In *4th Int'l Workshop on Productivity and Performance (PROPER 2011), in conjunction with EuroPar*, August 2011.

H. Subramoni, P. Lai, S. Sur and D. K. Panda, Improving Application Performance and Predictability using Multiple Virtual Lanes in Modern Multi-Core InfiniBand Clusters, In *International Conference on Parallel Processing (ICPP '10)*, September 2010.

H. Subramoni, K. Kandalla, S. Sur and D. K. Panda, Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine, In *Int'l Symposium on Hot Interconnects (HotI)*, August 2010.

H. Subramoni, P. Lai, R. Kettimuthu and D. K. Panda, High Performance Data Transfer in Grid Environment Using GridFTP over InfiniBand, In *Int'l Symposium on Cluster Computing and the Grid (CCGrid '10)*, May 2010.

P. Lai, H. Subramoni, S. Narravula, A. Mamidala and D. K. Panda, Designing Efficient FTP Mechanisms for High Performance Data-Transfer over InfiniBand, In *Int'l Conference on Parallel Processing (ICPP '09)*, September 2009.

H. Subramoni, P. Lai, M. luo, Dhabaleswar K. Panda, RDMA over Ethernet - A Preliminary Study, In *Workshop on High Performance Interconnects for Distributed Computing (HPIDC '09)*, September 2009.

Fields of Study

Major Field: Computer Science and Engineering

Table of Contents

Page

List of Tables

List of Figures

Chapter 1: Introduction

Modern high-end computing (HEC) systems allow scientists and engineers to tackle grand challenge problems in their respective domains and make significant contributions to their fields. Examples of such fields include astro-physics, earthquake analysis, weather prediction, nanoscience modeling, multiscale and multiphysics modeling, biological computations, computational fluid dynamics, etc. The design and deployment of such ultra-scale systems is fueled by the increasing use of multi-core and many-core architectures in multiple environments including commodity cluster systems (e.g., Intel/AMD CMPs [8, 59, 108, 138, 139]), alternate architectures and accelerators (e.g., IBM Cell[69], GPGPUs: general-purpose computation on graphics processing units, MICs: Many Integrated Cores), and large-scale supercomputers (e.g., IBM BlueGene [48], SiCortex [123], Cray [121, 122]). The emergence of commodity networking technologies like InfiniBand [6] is also fueling this growth and allowing high-bandwidth, low-latency petascale systems to be designed with commodity cluster configurations at relatively modest costs. Based on the Jun'13 Top500 ranking there are 205 (41%) InfiniBand systems in the Top500. A notable addition to this list is the 6th ranked TACC Stampede system [132] with 462,462 cores, a non-blocking InfiniBand interconnect, and a peak performance of 5.2 PFlop/s. Additionally, a number of even larger systems are on the horizon for the national community along with numerous HEC research clusters being deployed across university campuses nationwide.

Across scientific domains, application scientists are constantly looking to push the envelope regarding performance by running increasingly larger parallel jobs on supercomputing systems. Supercomputing systems are currently comprised of thousands of compute nodes based on modern multi-core architectures. Interconnection networks have also rapidly evolved to offer low latencies and high bandwidths to meet the communication requirements of parallel applications. As shown in Figure 1.1, large scale supercomputing systems are organized as racks of compute nodes and use complex network architectures ranging from fat-trees to tori. The interconnect fabric is typically comprised of leaf switches and many levels of spine switches. Each traversal (hop) of a switch between two end points increases the message latency, and Table 1 shows the impact of alternate hop counts on the MPI communication latency as measured on the TACC Ranger and Stampede systems under quiescent conditions. Even under such ideal conditions, we observe that the latencies of the 5-hop inter-node exchanges are almost 81% and 131% higher than that of intra-rack exchanges on Ranger and Stampede respectively. Moreover, supercomputing systems typically have several hundred jobs running in parallel and it is common for the network to be congested, further affecting the communication latency. In this context, it is extremely critical to design communication libraries in a network-topology-aware manner.

Most of the traditional HEC applications and current petascale applications are written using the Message Passing Interface (MPI) [86] programming model. To maximize portability, application writers typically use the semantics and API of a given programming model to write their parallel programs. On distributed memory HEC systems, support is thus provided via tuned MPI libraries so that parallel programs can be executed with best performance, scalability, and efficiency. Performance of an MPI program on a given system

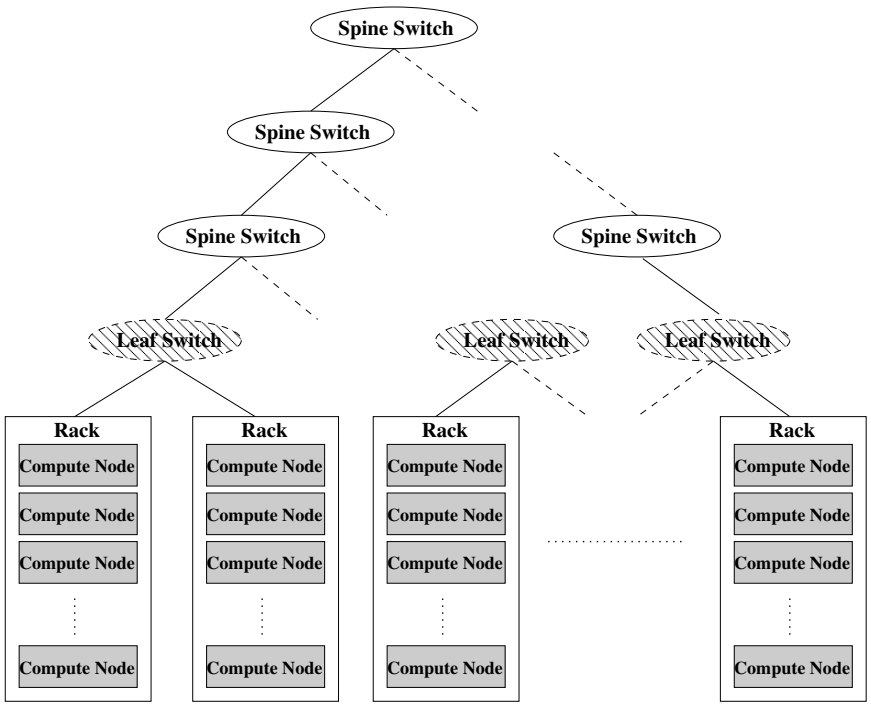


Figure 1.1: A Typical Topology

is heavily dependent upon how the MPI library on that system has been designed and optimized to take the system architecture (processor, memory, network interface, and network topology) into account. Over the years, many MPI libraries (open-source and proprietary) have been designed to deliver optimal performance and scalability for different systems. However, designing MPI libraries that manage the internal communication pipeline for both point-to-point and collectives based on the underlying network topology, is still an open challenge.

Resource allocation on modern supercomputing systems is done through job schedulers such as PBS [105] and Slurm [124]. Currently, most of these schedulers do not have the intelligence to allocate MPI tasks based on the underlying topology of the system and the communication requirements of the applications. Thus, the performance and scalability

of a parallel application can suffer (even using the best MPI library) if topology-aware scheduling is not done. Once an MPI job has been started by the job scheduler, each process in an MPI job is uniquely identified by its rank. This rank assignment is an important part of the overall application performance because communication steps (especially in collectives) use internal algorithms based on the ranks. Many applications use a structured form of communication where certain mappings will be more advantageous than others. A naive task assignment can result in poor locality of communication. Thus, it is important to design optimal mapping schemes with topology information to improve the overall application performance and scalability. All of these require thorough understanding of the underlying network topology.

Process Location	Number of Hops	MPI Latency (*us*) on Ranger	MPI Latency (*us*) on Stampede
Intra-Node	0 Hop in Leaf Switch	0.46	0.19
Intra-Rack	1 Hop in Leaf Switch	1.57	1.07
Inter-Rack	3 Hops Across Spine Switch	2.45	1.76
	5 Hops Across Spine Switch	2.85	2.54

Table 1.1: MPI Communication Performance Across Varying Levels of Switch Topology on TACC Ranger & Stampede

It is also critical for users of HPC installations to clearly understand the impact IB network topology can have on the performance of HPC applications. However, no tool currently exists that allows users of such large scale clusters to analyze and to visualize the communication pattern of their MPI based HPC applications in a network topology-aware

manner. Most contemporary MPI profiling tools for IB clusters also have an overhead attached to them. This prevents users from deploying such profilers on large-scale production runs intent on achieving the best possible performance from their codes.

To summarize, we address the following important challenges in this dissertation:

1. **How can a set of algorithms for frequently-used collectives (broadcast, reduce, all-reduce, gather, scatter, and all-to-all) that are topology-aware be designed?**
2. **Is it possible to design topology-aware communication mechanisms for point to point operations?**
3. **Can a topology-aware task re-ordering framework be designed based on a set of hosts provided by the resource scheduler?**
4. **What are the challenges involved in designing topology-aware scheduling schemes?**
5. **Is it possible to define and design a flexible topology management API so that MPI applications, job schedulers, and users (application developers and system administrators) can take advantage of it?**
6. **How can we design a network topology-aware, scalable, low-overhead profiler for IB clusters that is capable of detecting the communication pattern of high performance MPI applications?**

1.1 Problem Statement

Most of the current petascale applications use the MPI programming model, and spend a lot of time on communication (both point-to-point and collectives). Many of these applications also use structured communication, such as nearest neighbor, row/column-based communication, communication along multiple dimensions, etc. The performance of these applications running on petascale systems is heavily dependent on the following factors: 1)

how the scheduler allocates the required number of cores, 2) how processes of the application get mapped to the cores supplied by the scheduler, and 3) how the underlying MPI library implements point-to-point and collective operations to minimize network contention (in a static and a dynamic manner).

Modern petascale systems with many thousands of cores are designed with multiple levels of hierarchy: sockets consisting of multiple cores, nodes consisting of multiple sockets, blades consisting of multiple nodes, racks consisting of multiple blades, and the entire system consisting of multiple racks. A set of network switches with different topologies (fat-trees with different degrees of over-subscription, 3D meshes, etc.) are typically used to connect all these components. The routing schemes of the underlying network provide connectivity across all the cores. As modern petascale systems are heading towards 100K or more cores, and petascale applications try to achieve the best performance and scalability from the underlying systems with multiple levels of hierarchy, the conventional mode of job scheduling, process mapping, and executing MPI applications without the knowledge of underlying system topology, routing, and contention leads to poor performance and scalability for the applications. Also, different runs of the applications on the same system lead to wide variation in performance. Many of these issues, including network routing, contention, point-to-point / collective communication, and programming models for next generation petascale and exascale systems were discussed in a DOE sponsored Interconnection Network Workshop [54] in July 2008.

This leads to the following broad challenge: ***Can petascale and next generation exascale systems provide topology-aware MPI communication, process mapping and scheduling that can improve performance and scalability for a range of applications?*** If such

schemes can be designed and developed in an efficient manner, it will not only benefit application scientists and system administrators, it will also accelerate the effective use of current and next-generation ultra-scale systems. Figure 1.2 shows the overall scope of this dissertation. We aim to address the following challenges using this research framework:

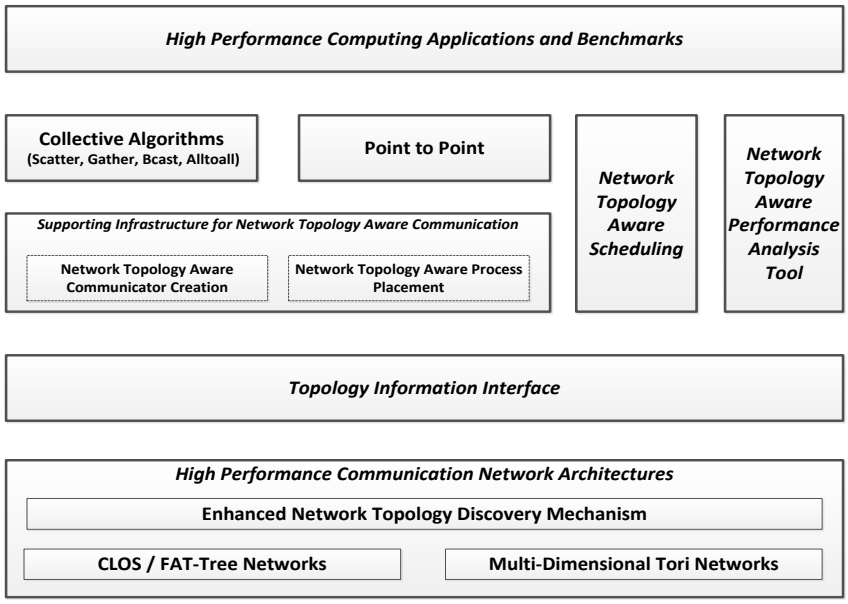


Figure 1.2: The Proposed Research Framework

1. **How can a set of algorithms for frequently-used collectives (broadcast, reduce, all-reduce, gather, scatter, and all-to-all) that are topology-aware be designed?**

   The Message Passing Interface (MPI) [86] has been the dominant programming model for high performance computing applications owing to its ease of use and scalability. The current MPI Standard, MPI-3.0, defines a set of collective operations that may be used to communicate data among a group of participating processors. Because they are easy to use and portable, these collective operations are very

commonly used by application developers. Scientific applications are also known to spend a considerable fraction of their run-times in MPI collective operations. The performance of collective operations is very critical to the overall application performance, particularly at large scales. Most MPI libraries such as MPICH2 [60], Open-MPI [94] and MVAPICH2 [10] use aggressive techniques to optimize the latency of collective operations [72, 109]. Researchers have proposed various multi-core aware, shared-memory based techniques to minimize the communication latency of collective operations. However, these techniques are limited to leveraging the node-level hardware topology and do not consider the network topology. Researchers have demonstrated that [22, 51, 113] the network topology plays a critical role in the latency of collective performance and scientific applications. We believe that it is critical to design MPI libraries in a network topology-aware manner to improve the communication costs of collective operations at large scales to deliver high performance to parallel applications.

2. **Is it possible to design topology-aware communication mechanisms for point-to-point operations?**

   Multiple classes of high performance computing applications use MPI point-to-point communication to transfer large amounts of data between various processes. Multi-dimensional stencil based applications (2D, 3D and 4D) and nearest neighbor applications are among the more popular codes that use MPI point-to-point communication heavily. In these applications, after each time step, each process communicates with its 2-D or 3-D neighbors. These end-of-timestep message exchanges occur at the same time, which can create significant pressure on the network if neighboring processes are not as topologically close together as possible. For such applications, it

is an open challenge whether it is possible to obtain significant performance improvement by optimizing point-to-point messaging with network topology-aware information. Thus, if we re-structure the application to pass the communication pattern to the MPI library using appropriate interfaces, the MPI library should be able to schedule messages on the appropriate routes. In this context, it is imperative that we design network-topology-aware communication mechanisms to optimize the performance of point-to-point operations on large scale supercomputing systems.

3. **Can a topology-aware task re-ordering framework be designed based on a set of hosts provided by the resource scheduler? This will allow the job to be run with topology-aware routines and libraries.**

   On current generation systems, once the scheduler supplies a list of available nodes, MPI processes must be started on these nodes. When each MPI process is started, it is assigned a rank. This rank assignment is an important part of the overall application performance because communication steps (especially in collectives) use internal algorithms based on the ranks. A naive task assignment can result in poor locality of communication. Many applications use a structured form of communication where certain mappings will be more advantageous than others. In general, the default mapping scheme is to allocate in blocks. In this form, sequential ranks are allocated together on the same node. Although block scheduling can improve performance over a random scheduling, there is room for improvement by taking the network topology and application communication pattern into account. For example, in applications that do near-neighbor communication not only numerically-near processes communicate directly, many numerically-far processes also communicate directly. Thus, if we can profile applications, identify their communication patterns

and automatically re-order the ranks based on the communication pattern and the underlying network topology, we will be able to improve the overall application performance and scalability.

4. **What are the challenges involved in designing topology-aware scheduling schemes? This will allow the resource scheduler to use dynamic network information from topology and routing path detection framework to make intelligent scheduling decisions.**

   In order for topology-aware point-to-point communication, collective communication, and task mapping to be effective in a large-scale production HEC environment, the batch scheduling system that controls the assignment of compute nodes to batch jobs must also be topology-aware. As part of proposed work, we extend the scheduling algorithms in the SLURM [124] scheduler on Stampede [132] at TACC to include topology-awareness.

5. **Is it possible to define and design a flexible topology management API such that MPI applications, job schedulers, and users (application developers and system administrators) can take advantage of it?**

   We explore the design of a highly scalable network topology detection mechanism in this thesis. However such a mechanism would be irrelevant if we are unable to get this information to the other components of the supercomputing ecosystem–the MPI library, the job scheduler, and the application in an efficient and scalable manner. This interface should be able to handle requests from potentially tens of thousands of processes. It also needs to provide 'ease of use' so that MPI libraries, mapping schemes and schedulers can utilize this information efficiently and effectively.

6. **How can we design a network topology-aware, scalable, low-overhead profiler for IB clusters that is capable of detecting the communication pattern of high performance MPI applications?**

   As IB clusters grow in size and scale, it becomes critical for the users of HPC installations to clearly understand how an HPC application interacts with the underlying IB network and the impact it can have on the performance of the application. However, no tool currently exists that allows users of such large scale clusters to analyze and to visualize the communication pattern of their MPI based HPC applications in a network topology-aware manner. Most contemporary MPI profiling tools for IB clusters also have an overhead attached to them. This prevents users from deploying such profilers on large-scale production runs intent on achieving the best possible performance from their codes.

Chapter 2: Background

2.1 InfiniBand

The InfiniBand Architecture [58] (IBA) defines a switched network fabric for interconnecting compute and I/O nodes. In an InfiniBand network, compute and I/O nodes are connected to the fabric using Channel Adapters (CAs). There are two types of CAs: Host Channel Adapters (HCAs) which connect to the compute nodes and Target Channel Adapters (TCAs) which connect to the I/O nodes. IBA describes the service interface between a host channel adapter and the operating system by a set of semantics called verbs. Verbs describe operations that take place between a CA and the host operating system for submitting work requests to the channel adapter and returning completion status. InfiniBand uses a queue based model. A consumer can queue up a set of instructions that the hardware executes. This facility is referred to as a Work Queue (WQ). Work queues are always created in pairs, called a Queue Pair (QP), one for send operations and one for receive operations. In general, the send work queue holds instructions that cause data to be transferred between the consumer memory and another consumer memory, and the receive work queue holds instructions about where to place data that is received from another consumer. The completion of Work Queue Entries (WQEs) is reported through Completion Queues (CQ). InfiniBand supports four transport modes: Reliable Connection (RC), Reliable Datagram (RD), Unreliable Connection (UC) and Unreliable Datagram (UD). Of these, RC and UD are required to be supported by Host Channel Adapters (HCAs) in the InfiniBand specification.

2.1.1 Communication Semantics in InfiniBand

InfiniBand supports two types of communication semantics: channel and memory semantics. In channel semantics, the sender and the receiver both must post work request entries (WQEs) to their QP. After the sender places the send work request, the hardware transfers the data in the corresponding memory region to the receiver end. It is to be noted that the receive work request needs to be present before the sender initiates the data transfer. This restriction is prevalent in most high-performance networks like Myrinet [26], Quadrics [104] and others. The sender will not complete the work request until a receive request has been posted on the receiver. This allows for no buffering and zero-copy transfers. When using channel semantics, the receive buffer size must be the same or greater than that of the sending side. Receive WQEs are consumed in the same order that they are posted. In the case of reliable transports, if a send operation is sent on a QP where the next receive WQE buffer size is smaller than needed the QPs on both ends of communication are put into the error state. In memory semantics, Remote Direct Memory Access (RDMA) operations are used instead of send/receive operations. These RDMA operations are one-sided and do not require software involvement at the target. The remote host does not have to issue any work request for the data transfer. Both RDMA Write (write to remote memory location) and RDMA Read (read from remote memory location) are supported in InfiniBand, although not all transports support it.

2.1.2 InfiniBand Network Counters

The InfiniBand specification provides a rich set of counters to monitor the health and state of the network fabric. Among these the *PortXmitWait* and *PortXmitDiscard* counters can be used to infer the current state of congestion in the network. According to the InfiniBand specification, the *PortXmitWait* counter represents the number of ticks during which

the network port had data to transmit but no data was sent during the entire tick because of insufficient credits or because of lack of arbitration. The *PortXmitDiscard* counter represents the total number of outbound packets discarded by the port, either because it is down or congested. We use the *PortXmitDiscard* counter to infer the effects of congestion on the performance of the default, pair-wise exchange (XOR) algorithm used to implement the MPI_Alltoall operation. We also use it to quantify the benefits that a contention-aware communication schedule can have on the amount of congestion in the network.

2.2 MPI

Message Passing Interface (MPI) [86], is one of the most popular programming models for writing parallel applications in cluster computing area. MPI libraries provide basic communication support for a parallel computing job. In particular, several convenient point-to-point and collective communication operations are provided. High performance MPI implementations are closely tied to the underlying network dynamics and try to leverage the best communication performance on the given interconnect. We use a modified MVAPICH2 [10] for our evaluations. However, our observations in this context are quite general and they should be applicable to other high performance MPI libraries as well.

2.2.1 Collective Operations in MPI-2

The MPI standard [86] specifies various types of collective operations such as *All-to-All*, *All-to-One*, *One-to-All* and *Other*. Collective operations belonging to the types *All-to-One* and *One-to-All* assign one process as the *root* and involve a message exchange pattern in which the root either acts as the source or the sink. *Personalized* collective operations such as MPI_Scatter, MPI_Gather and MPI_Alltoall involve each process sending and/or receiving a distinct message. Operations of this type can benefit with networks that offer

higher bandwidths. We have studied the performance of MPI_Scatter and MPI_Gather collective operations, in-depth.

2.3 The MVAPICH2 MPI Library

MVAPICH2 [10], is an open-source implementation of the MPI-3.0 specification over modern high-speed networks such as InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). MVAPICH2 delivers best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,055 organizations world-wide in 70 countries and is powering some of the top supercomputing centers in the world, including the 6th ranked Stampede at TACC [132], 19th ranked Pleiades at NASA and the 21st ranked Tsubame 2.0 at Tokyo Institute of Technology.

MPI libraries typically use the *eager* protocol for small messages and the *rendezvous* protocol for large message communication operations. MVAPICH2 uses an RDMA-based eager protocol called RDMA-Fast-Path, along with various optimizations to improve the latency of small message point-to-point communication operations. For large messages MVAPICH2 uses zero-copy designs based on RDMA-Put or RDMA-get operations to achieve excellent communication bandwidth. Further, MVAPICH2 offers good scalability through advanced designs such as eXtended RC (XRC), Shared-Receive Queues (SRQ) and Hybrid (UD/RC) communication modes.

2.3.1 Collective Message Passing Algorithms used in MVAPICH2

In MVAPICH2 [10], we use optimized shared-memory based algorithms to optimize several collective operations. However, these optimizations are limited to identifying and grouping processes that are within the same compute node and we have no knowledge of the topology at the switch-level. Currently, for each communicator, we create an internal

shared-memory communicator to contain all the processes that are within the same compute node and share the same address space. We assign one process per compute node as the node-level leader process and create a node-level-leader communicator object to include all the node-level leader processes. For collectives such as MPI_Bcast, MPI_Reduce, MPI_Allreduce and MPI_Barrier, we schedule message exchange operations across these communicators to achieve lower latencies. These methods have two significant advantages:

- The shared-memory space is leveraged for exchanging messages between processes that are within the same node. This can lead to a higher degree of concurrency, than compared to exchanging messages through point-to-point calls, as shown in [14, 15, 68, 110]

- The intra-node stages of the communication operations can happen without any data movement across the network and can also minimize network contention

2.3.2 Broadcast Algorithms used in MVAPICH2

In MVAPICH2 [10], we use optimized multi-core aware, shared-memory based algorithms to implement the *One-to-All* Broadcast operation. These optimizations are limited to identifying and grouping processes that are within the same compute node and we have no knowledge of the topology at the switch-level. For each communicator, we create an internal shared-memory communicator to contain all the processes that are within the same compute node. We assign one process per compute node as the node-level leader process and create a node-level-leader communicator to include such processes. We schedule the MPI_Bcast operation across these communicators to minimize the communication latency. The intra-node phase of the MPI_Bcast operation can either use the direct shared-memory

approach or a k-nomial algorithm through basic point-to-point operations. For the inter-leader phase, we use the k-nomial algorithm for small and medium sized messages, and a scatter-allgather approach for larger messages.

2.3.3 MPI_Alltoall Algorithms in MVAPICH2

MVAPICH2 library implements the Alltoall personalized exchange collective operation through several popular algorithms. MVAPICH2 uses the Brucks algorithm [30] to implement small message Alltoall operations, scatter-destination algorithm for medium length messages and the pair-wise exchange algorithm for large message MPI_Alltoall operations. For N processes, the pair-wise exchange algorithm involves N iterations. In a given iteration i, a process with id $rank$ computes the peer process id as, $rank \oplus i$.

2.4 Applications and Kernels

In this Section we give some background on the various application-kernels and applications that are used for performance evaluation in this thesis.

2.4.1 MILC

MILC is a set of parallel numerical simulations developed by the MIMD Lattice Computation [134] collaboration for studying quantum chromodynamics (QCD). Codes from the *MILC* set are commonly used for benchmarking [88, 127, 136]. We use the ks_imp_dyn code from *MILC* which employs the conjugate gradient (CG) method on a 4D lattice. Communication is primarily nearest neighbor point-to-point and Allreduce as described in [91]. The influence of the layout on the communication performance is discussed in [49].

2.4.2 WindJammer

Windjammer issues two large broadcast calls in its main loop. Windjammer is a parallel Neighbor-Joining MPI C++ program used to build approximate evolutionary trees, known

as phylogenetic trees, from either DNA or amino acids sequences, known as taxa. The Neighbor-Joining algorithm was first described in 1987 [117]. Windjammer is based on the scalar NINJA program developed by Wheeler [141]. In particular, it relies heavily on the Wheeler optimization described as "d-filtering". The application is used to evaluate our proposed broadcast algorithms.

2.4.3 AWP-ODC

Anelastic Wave Propagation (AWP-ODC) is a community model [35–37, 67, 93] used by researchers at the Southern California Earthquake Center (SCEC) for wave propagation simulations, dynamic fault rupture studies, physics-based seismic hazard analysis, and improvement of structural models. Some of the most detailed simulations to date of earthquakes along the San Andreas fault were carried out using this code, including the well-known TeraShake, SCEC ShakeOut simulations. AWP-ODC solves the 3D velocity-stress wave equation explicitly by a staggered-grid FD method and has predominately nearest-neighbor communication, fourth-order accurate in space and 2nd-order accurate in time. Each processor is responsible for performing stress and velocity calculations for a 3D rectangular sub-grid. Nearest neighbor exchange is used to obtain new values of wave-field parameters from neighboring sub-grids. Recent work by Cui, et al. [36, 37], enhanced the application through single-processor, I/O handling and initialization optimization. An implementation using MPI2 RMA for the nearest neighbor exchange is demonstrated in [106] and the code was a finalist for the Gordon Bell Prize in 2010 [12].

2.4.4 Hypre

Hypre is an open-source, high performance and scalable package of parallel linear solvers and preconditioners. Hypre is designed to leverage the notion of conceptual interfaces, which expose the various solver routines to users in a modular fashion [42]. Such

a design significantly eases the coding burden for application developers and may also be used to provide additional application information to the solver routines. The solvers in Hypre are robust, numerically stable and scalable [17]. Its object model is more generic and flexible when compared to many state-of-the-art solver packages and it may also be used as a framework for algorithm development. In this work, we focus specifically on the PCG solver routine, which uses the diagonal scaling preconditioner.

2.4.5 Parallel 3-D FFT

Many applications in areas including Direct Numerical Simulations of Turbulence, astrophysics, and material science rely on highly scalable, 3D FFTs [38, 114]. The Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) library [98, 102] from the San Diego Supercomputer Center (SDSC) is a portable, high performance and open source implementation based on the MPI programming model. It leverages the fast serial FFT implementations of either IBM's ESSL or FFTW. P3DFFT uses a 2D, or pencil, decomposition and overcomes an important limitation to scalability inherent in FFT libraries by increasing the degree of parallelism up to N^2 where N is the linear size of the data grid. It has been used in various Direct Numerical Simulation (DNS) turbulence applications [38].

2.5 Graph Partitioning Softwares

Jostle [140] and ParMETIS [120] are parallel multilevel graph-partitioning software packages which efficiently partition graphs containing millions of nodes. Jostle was developed by Dr. Walshaw et al. at the University of Greenwich. It uses a multilevel refinement strategy to create perfectly balanced partitions and scales well on graphs containing millions of nodes. ParMETIS is a parallel library for graph partitioning and fill-reducing matrix ordering developed by Dr. Karypis, et. al of the University of Minnesota. We use the

graph partitioning functions of ParMETIS which perform multi-level k-way partitioning using algorithms designed to quickly provide high quality partitions of very large graphs. Multilevel schemes match and coalesce pairs of adjacent vertices to define a new graph and recursively iterate this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (possibly with a crude algorithm) and the partition is successively refined on all the graphs starting with the coarsest and ending with the original. At each change of levels, the final partition of the coarser graph is used to give the initial partition for the next level down.

2.6 Graph Matching

The Alltoall implementation in MVAPICH2 [10] uses a pair-wise exchange algorithm. In each step, the communication schedule can be thought of as a *perfect matching* of a graph of the process ranks. The shortest possible schedule is of length $P - 1$ where P is the number of processes and it is a sequence of perfect matchings, M_i, starting with an initial graph G_0 that is a complete graph of the process ranks. In each step a matching, M_i, is found on the new subgraph, G_i, where $G_i = G_{i-1} - e_j, \forall e \in M_{i-1}$. In this work we present a greedy algorithm to find such a sequence of matchings, with the added constraint that we limit the number of conflicting routes among the edges in a given matching. Due to this constraint, some matchings may not include all ranks and additional communication steps may be required.

2.7 SLURM

SLURM (Simple Linux Utility for Resource Management) is a popular workload manager developed by LLNL (Lawrence Livermore National Laboratory) that has support for network-topology aware scheduling [46]. Due to its successful adoptation into the HPC community the primary developers of SLURM founded SchedMD [13] which should help

reach a wider audience including Big Data users. SLURM provides a specific plugin to support FAT tree networks. The plugin attempts to place the jobs in a *best fit* way on the network and tries to minimize the number leaf switches on which the jobs are allocated. SLURM also provides the flexibility to reduce the spread of the job (the number of switches to which the job is connected) at various levels of the FAT tree.

2.8 Neighbor Joining

The neighbor-joining method (NJ) of Saitou and Nei [118] is a technique in computational biology for constructing phylogenetic trees which represent the evolutionary relationships among biological species based on similarities and differences in their characteristics. The authors' exposure to these techniques inspired the use of agglomerative minimum-distance complete-linkage hierarchical clustering in this work [63]. These algorithms generate binary trees from weighted complete graphs to identify closely-connected or closely-related regions. The algorithm starts with a weighted complete graph and an associated symmetric distance matrix giving the distance (the edge weights) between all pairs of nodes. At each step of the process, an auxiliary symmetric matrix is computed from the distance matrix, and then this matrix is searched for its smallest entry. The pair of nodes corresponding to this entry are "joined" by removing them from the graph and matrices and replacing them by a single new entry. New distances and auxiliary values are computed between the new node and the remaining nodes. The process repeats until only one node remains in the graph. The binary tree is then constructed by connecting all joined nodes to the new node they create.

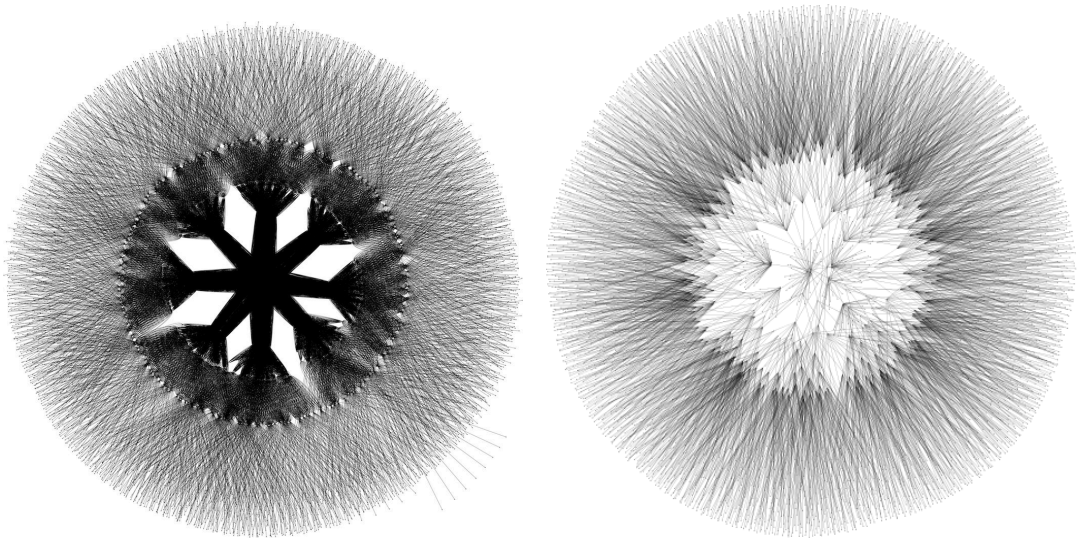
2.8.1 Neighbor Joining Based Abstraction of IB networks

InfiniBand network topology data is not available in a mode easily used by other programs. The required routing data is generally not available to users of IB systems who do

not have administrative privileges. To get IB routing data available to general users of a system, we have taken a two-step approach. We query routing information directly within the OpenSM subnet manager using a plugin interface. Then, we have developed a routing query server that ingests this data at startup and continuously as it is generated and provides a query service for the routes that it describes. We use a simplified form of the NJ algorithm to ingest the topology data. Algorithm 1 shows the version of NJ algorithm we have used. The neighbor-joining algorithm implements agglomerative minimum-distance complete-linkage hierarchical clustering [63] to cluster the graph associated with the routes specified in the Linear Forwarding Tables (LFTs) for an IB fabric [40].

The multi-joining step is natural in this problem since we expect to have many hosts which are connected through a single switch ASIC in the leaves of the physical network. IB fabrics often have an identifiable leaf switch connecting 12 or 18 HCAs into the fabric. Our clustering algorithm will detect these closely-connected hosts because each host connected physically to this ASIC will have a logical route through this ASIC only. As a result, all hosts physically connected to this switch will have unit distances to each other and will be subsequently joined together by the modified NJ algorithm. Figures 2.1(a) and 2.1(b) depict the physical network topology and the output (neighbor joined tree) of the NJ algorithm for the Ranger supercomputer at TACC. Figures 2.2(a) and 2.2(b) show the same results for the Gordon supercomputer at SDSC.

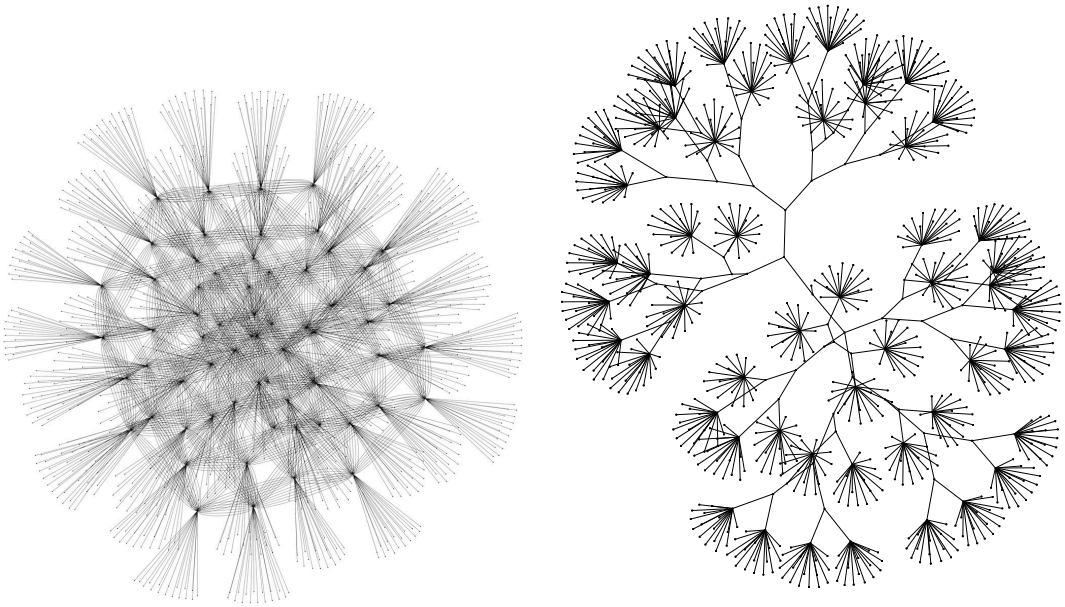
The routing query server is a multi-threaded network program designed to handle large numbers of simultaneous connections on large machines with many jobs simultaneously requesting data. The NJ algorithm requires $\mathcal{O}(N_{\text{hosts}}^2)$ routing queries to populate the initial distance matrix. Our approach only executes this algorithm once at startup or whenever



(a) Physical network topology graph - 4,037 end points (input)

(b) Neighbor joined tree (output)

Figure 2.1: Graphical overview of NJ algorithm created within the topology query service on Ranger



(a) Physical network topology graph - 1,086 end points (input)

(b) Neighbor joined tree (output)

Figure 2.2: Graphical overview of NJ algorithm created within the topology query service on Gordon

changes are detected in the fabric connectivity. Note that we detect routing changes automatically via heavy sweep trigger events with the OpenSM plugin and automatically update the query service with new data after SM remaps have completed. Other approaches, which do not utilize such a centralized service, will have to incur this cost every time an application or MPI library wants to perform network-topology-aware activities.

Clients may connect to the server over TCP/IP sockets and make queries of the form:

```
query hosta:hostb
```

The service returns the GUIDs of the switches traversed between the hosts "hosta" and "hostb". E.g.,

```
query i101-101:i102-201
i101-101 0x00144fa50f240050 0x00144f0000a60369
0x00144f0000a6037a 0x00144f0000a6036a
0x00144fa43db80050 i102-201
```

2.8.2 Performance of Neighbor Joining with OpenMP Constructs

As we saw in Section 2.8.1, the NJ algorithm requires $\mathcal{O}(N_{\text{hosts}}^2)$ routing queries at startup to populate the initial distance matrix. For large fabrics, this step originally took several minutes on a single core and comprised over 80% of the runtime required to initialize the service. To alleviate this bottleneck, we optimized and threaded this portion of the algorithm using OpenMP constructs. Figure 2.3 shows the results of OpenMP parallelization from this effort for the Lonestar and Ranger fabrics using 3,942 and 1,901 endpoint hosts for the NJ algorithm, respectively. In both cases, parallel scaling efficiencies of 85% or higher were obtained when running on a 6-core Westmere processor running at 2.67 GHz. The total time to initialize the service for the large Ranger fabric is approximately 12 seconds using 6 threads.

Input : Weighted Network Topology Graph
Output: NJ tree

```
/* Initialize                                                    */
```
$d(i, j)$ mapping node pairs to distances;
```
/* This is a reverse map of d                                    */
```
$rd(m)$ mapping distance m to a list of node id pairs;
$a(i)$ active node ids;
```
/* A tree with one root node and active nodes as
   children                                                      */
```
t;

repeat
  for $l = 1, 2, \ldots$ **do**
    foreach *pair* $(i, j) \in rd(l)$ **do**
      if $a(i)$ && $a(j)$ **then**
        break;
      else
        remove inactive pairs from rd;
      end
    end
  end
  Add (i, j) to empty list L;
  foreach *row* i **do**
    Find, other $d(i, k)$==$d(i, j)$ and add k to L;
  end
  foreach $i \in L$ **do**
    foreach $j \in L$ **do**
      if $d(i, j)! = \text{minimum}$ **then**
        Remove j from L;
      end
    end
  end
  foreach *i in L* **do**
    $a(i)$=false;
  end
  Create new tree node n and insert into t;
  Connect n to nodes in L as parent and root as child;
  set m=max((x,y) in LxL: d(x,y));
  foreach *a(i)==true* **do**
    $d(i, n)$=m;
    add (i, n) to $rd(m)$;
  end
until *no new entries added to t*;

Algorithm 1: Neighbor Joining Algorithm

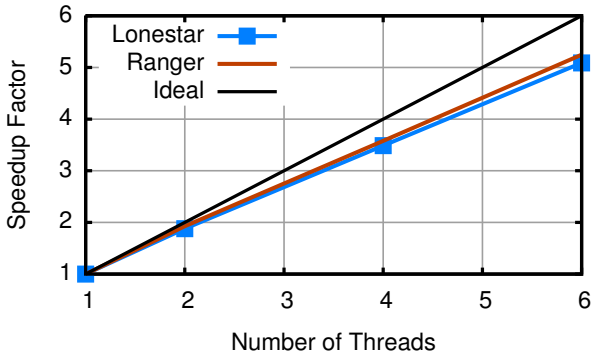


Figure 2.3: OpenMP performance of host-pair queries showing parallel efficiencies of 85% on a single six-core Intel Westmere processor.

Chapter 3: Design and Evaluation Topology-Aware Collective Algorithms for Scatter and Gather

Chapter 1 describes how network topology plays a critical role in the performance of collective operations and consequently of scientific applications that depend on these collectives for their data transfer needs. Thus, it is critical to design MPI libraries in a network topology-aware manner to improve the communication costs of collective operations at large scales to deliver high performance to parallel applications. In this chapter of the thesis, we take on this challenge and attempt to answer the first question posed in Section 1.1. We describe the challenges involved in detecting the topology of large scale InfiniBand clusters and leveraging this knowledge to design efficient topology-aware algorithms for collective operations. We also propose a communication model to analyze the communication costs involved in collective operations on large scale supercomputing systems. We have analyzed the performance characteristics of two collectives, MPI_Gather and MPI_Scatter, on such systems and we have proposed topology-aware algorithms for these operations.

3.1 Designing Topology-Aware Collective Algorithms

In Chapter 1, we illustrated the impact of topology on small message latency between a pair of processes. Collective operations involve many processes exchanging messages and are sensitive to network noise [52]. Since production environments allow several applications to run concurrently, the effective bandwidth available per application also plays a key

role in determining the time consumed for a collective operation. In this chapter, we propose topology-aware algorithms for optimizing the performance of collective operations on large scale computing systems.

In [133], the authors proposed models to predict the costs of various collective algorithms for small scale clusters with compute nodes comprising of a single core. We extend these models to incorporate the communication costs associated with various levels of hierarchy in modern large scale clusters. Let t_s-*intra-node* be the start-up cost associated with an intra-node message exchange operation and t_w-*intra-node* be the cost involved in sending a word of data to a peer process within the same node. Similarly, t_s-*intra-switch* and t_w-*intra-switch* are the costs associated with a message exchange operation between processes that are connected to the same leaf switch and t_s-*inter-switch* and t_w-*inter-switch* are the costs involved for an inter-switch transfer operation. Owing to the hierarchy of the system, the following is true :

$$t_s\text{-}intra\text{-}node < t_s\text{-}intra\text{-}switch < t_s\text{-}inter\text{-}switch$$

$$t_w\text{-}intra\text{-}node < t_w\text{-}inter\text{-}switch < t_w\text{-}inter\text{-}switch$$

It is to be noted that the inter-switch costs depends on the number of switch-hops involved in the message exchange operation. Also, we assume that the contention involved in intra-node and intra-switch operations are insignificant when compared to that of inter-switch exchanges.

3.1.1 MPI_Gather and MPI_Scatter - Default algorithms

As explained in [111], MPI implementations such as MPICH2 [60], Open-MPI [94] and MVAPICH2 [10] use tree-based algorithms to implement these operations. These algorithms were proposed and optimized for conventional single-core systems. However, on large scale production systems, it is necessary to design efficient topology-aware algorithms to lower the communication costs. Since these algorithms are popularly used across various MPI implementations, we have chosen to study the behavior of these algorithms on large scale InfiniBand clusters.

Consider the case in which an MPI_Gather operation is being invoked with a message size of *N* bytes, with a group of *P* processes, that are distributed in some manner across *R* racks. Let *T_binomial* be the cost of performing an MPI_Gather operation using the binomial algorithm on such a system. For simplicity, we assume that the root of the operation is also one of the rack-level leaders. The inter-rack communication can involve more than one inter-switch exchange operation. Since we know that the latency involved in inter-switch message exchange operations is higher when compared to intra-switch and intra-node exchanges, the cost of the entire operation is dominated by the number of inter-switch operations, the cost associated with each inter-switch exchange and the network contention. Consider α to be the overhead introduced by the contention at various levels of switches in the system. The cost involved in binomial tree based approaches with *P* processes on traditional single-core systems could be expressed in terms of *log(P)*. Let us introduce three variables C_1, C_2 and C_3 to account for the hierarchy in the system. These parameters represent the number of intra-node, intra-switch and inter-switch exchanges such that, $C_1 + C_2 + C_3 = log(P)$. Their values also depend on the distribution of processes across different racks. Depending on the processor allocation, each process can potentially

be involved in an inter-switch exchange at some point during the MPI_Gather operation. So, we can assume that the value of C_3 can be $\geq R$. Also, in an MPI_Gather operation, different processes exchange different volumes of data depending on their position in the binomial tree. Since the cost of communication also depends on where the processes are physically located in the cluster, we introduce three positive variables γ, β and δ to indicate the fact that the t_w component of *T_binomial* is obtained by adding up the individual costs of various intra-node, intra-switch and inter-switch operations, respectively. The values of these parameters are chosen such that, C_1*γ, C_2*β and C_3*δ belong to the interval $\{$*0, (P - 1)/P*$\}$. If all the processes are physically connected to just a single switch, there will be no inter-switch exchanges and the value of δ will be 0, and C_1*γ + C_2*β = *(P - 1)/P*. On extending the models presented in [111], we can express the cost of an MPI_Gather operation as

$$\begin{aligned} T_{\textit{binomial}} =& (t_s\textit{-inter-node} * C_1 + t_s\textit{-intra-switch} * C_2 \\ &+ \alpha * t_s\textit{-inter-switch} * C_3) + t_w\textit{-intra-node} \\ &* (C_1) * (N * \gamma) + t_w\textit{-intra-switch} \\ &* (C_2) * (N * \beta) + \alpha * t_w\textit{-inter-switch} \\ &* (C_3) * (N * \delta) \end{aligned}$$

Since the costs involved in inter-switch operations dominate the cost of the entire operation, the following is true:

$$T_{\textit{binomial}} > (\alpha * t_s\textit{-inter-switch} * C_3) + (\alpha * N * t_w\textit{-inter-switch} * C_3 * \delta)$$

3.1.2 Topology-Aware MPI_Gather and MPI_Scatter algorithms

From the above cost model, we can observe that if we schedule the inter-switch exchanges in an optimal manner, we can minimize the costs incurred due to the effects of network contention and also the costs associated with making multiple hops across various levels of the switches.

We create sub-communicators to reflect the topology of the system. This allows us to perform the entire collective operation efficiently in a recursive manner by minimizing the number of inter-switch operations and lowering both the t_s and t_w components of the costs of collective operations which result in better performance for both small and large messages. We can perform the entire collective operation in the following manner:

- The rack-leader processes independently perform an intra-switch gather operation. This phase of the algorithm does not involve any inter-switch exchanges.
- Once the rack-leaders have completed the first phase, the data is gathered at the root through an inter-switch gather operation performed over the *R* rack-leader processes.

However, it is to be noted that the messages exchanged between the rack-leader processes are the aggregated messages of size $M \geq N$, which were gathered at the rack-leader processes at the end of the first phase of the algorithm. Let T_{topo} be the cost of the proposed topology-aware algorithm. Since each rack-leader process is now a part of a binomial tree and has vectors of size *M*, T_{topo} can be expressed in a manner similar to the regular binomial exchange algorithm across *R* processes with a vector size of *M* bytes. Since we established a lower bound for the performance of $T_{binomial}$, in Section 3.1.1, we can express the lower bound of the cost of our proposed topology-aware algorithms in the following manner:

$$T_{topo} > (\alpha * t_s\text{-}inter\text{-}switch * log(R)) + (\alpha * (1 - 1/R))/(M * t_w\text{-}inter\text{-}switch)$$

We indicated in Section 3.1.1, that the number of inter-switch exchange operations is a linear function of *R*. However, with our proposed topology-aware algorithm, the number of inter-switch operations is a function of *(logR)* and this should theoretically improve the latency component of the cost function by a factor of *(*R/logR) and this plays a key-role in minimizing the overall time consumed for small message exchanges. For larger messages, the factor of improvement can be expressed as:

$$T_{binomial}/T_{topo} = N * \delta * C_3/(M * (1 - 1/R))$$

$$T_{binomial}/T_{topo} = f(R)$$

In a general case, the root of the operation might not be physically located to be assigned as a rack-level leader. In such cases, our proposed algorithm will involve an additional communication step to move the aggregated data from the rack-leader process to the root process. Since this happens within a rack, this involves data movement within the leaf switch, but will not lead to any traffic over any other switches in the network. Since MPI_Gather and MPI_Scatter are symmetric operations, we have designed a similar topology-aware algorithm for the MPI_Scatter operation as well.

3.2 Experimental Results

We used three InfiniBand DDR switches, Switches A, B and C to create a tree topology. Switch A is connected to 8 nodes based on the quad-core, quad-socket AMD Barcelona architecture and Switch B is connected to 32 nodes based on the quad-core, dual-socket Intel

Clovertown architecture. Switches A and B are connected to Switch C with two InfiniBand DDR links each. All nodes have ConnectX DDR HCAs. For our experiments, we have used 4 nodes (64 processes) from Switch A and 29 nodes (232 processes) from Switch B. We have also used a simple benchmark code that iterates through various message sizes starting from 0 bytes up to 1M bytes and invokes a collective call several times in a loop. For each message size, the benchmark computes the average time each process takes to complete the collective operation by performing a global reduction operation. We also run a background Alltoall job to study the impact of network congestion on the performance of the collective operations.

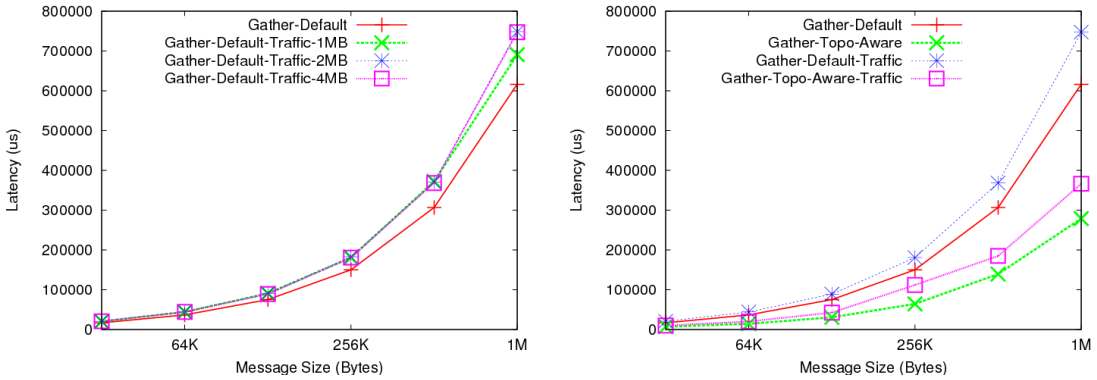


Figure 3.1: Gather latency for 296 processes: (a) Effect of background traffic on MPI_Gather and (b) Default Vs Topology-Aware MPI_Gather algorithms

In Figure 3.1 (a), we compare the performance of the default binomial tree algorithm for MPI_Gather under quiet and busy conditions. We can see that this algorithm is quite sensitive to the presence of background traffic. When the background job is run with buffer sizes larger than 1MB, we see a performance degradation of almost 21% for large messages. When the background job is run with a buffer size of 4MB, the time taken to

complete the MPI_Gather operation with 1MB message size is almost 10% higher than the case when the buffer size of the background job is 1MB. We can infer that the presence of background traffic can significantly affect the performance of the binomial exchange algorithm for MPI_Gather. In Figure 3.1 (b), we compare the performance of the default algorithm with the proposed topology-aware algorithm under both quiet and busy conditions. The background Alltoall job has been executed with a buffer size of 4MB to clearly delineate the performance difference between the default binomial exchange algorithm and the proposed topology-aware algorithm. We can infer that the proposed algorithm delivers an improvement of up to 50% for most messages, even when the network was busy. In Section 3.1.2, we had indicated that for large message MPI_Gather exchanges, we could potentially achieve a speed-up of a factor of R with our proposed topology-aware algorithms. In our experimental setup, we have distributed the processes across $R = 2$ racks and we would expect to see a performance benefit of a factor of 2. Hence, the experimental results that we have obtained match well with the theoretical analysis we had proposed in Section 3.1.2.

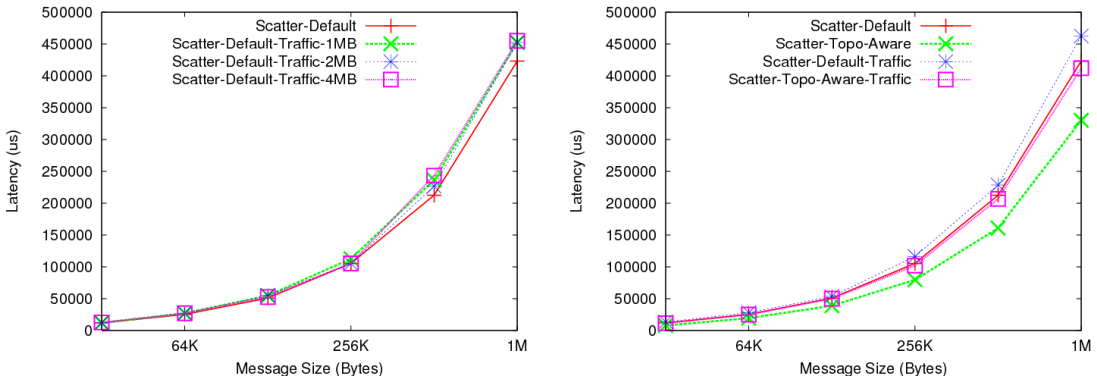


Figure 3.2: Scatter latency for 296 processes: (a) Effect of background traffic on MPI_Scatter and (b) Default Vs Topology-Aware MPI_Scatter algorithms

Similarly, in Figure 3.2(a), we compare the performance of the default binomial tree algorithm for MPI_Scatter under quiet and busy conditions. We observe that the presence of background traffic has resulted in a smaller overhead in this operation, than MPI_Gather. In Figure 3.2(b), we compare the performance of the default algorithm with the proposed topology-aware algorithm under both quiet and busy conditions. We can infer that the proposed topology-aware algorithm performs almost 23% better than the default algorithm when the network is quiet.

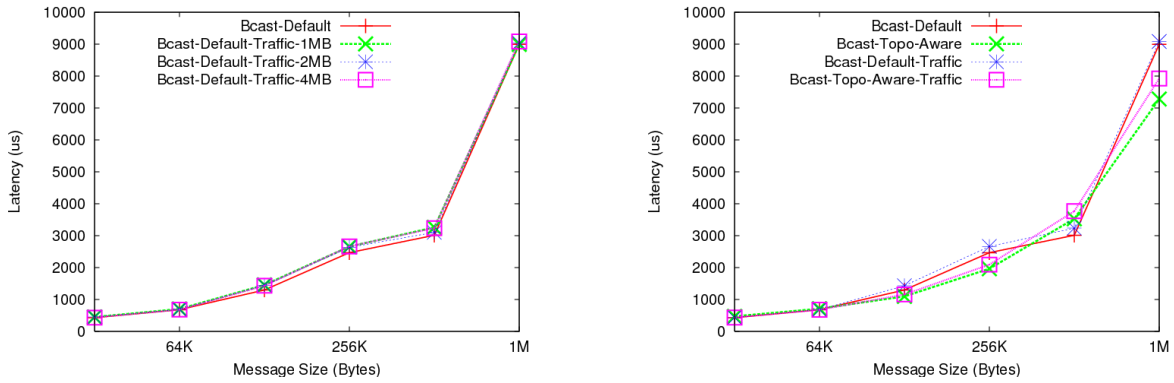


Figure 3.3: Bcast latency for 296 processes: (a) Effect of background traffic on MPI_Bcast and (b) Default Vs Topology Aware MPI_Bcast algorithms

In Figure 3.3 (a), we can see that the MPI_Bcast operation already appears to be quite resilient to the presence of network traffic. For large messages, we use the scatter-allgather algorithm to implement this operation. In our experiments, we have measured the performance with 296 processes. So, the size of the messages exchanged in the scatter phase of the algorithm will be $<=$ (Message_Size)/296. As indicated earlier, at this scale, we did not observe benefits for small and medium sized messages. However, from the cost models we

described in Section 3.1.1, since we have reduced the number of inter-switch exchanges, we expect to see an improvement on clusters involving multiple levels of switches.

3.3 Related Work

The concept of gathering topology information and leveraging this knowledge to design better algorithms has been applied in the past to grid based systems. Several researchers have explored the potential performance benefits of creating a multi-level communication framework to optimize the performance of parallel applications in grid environments [18, 31, 34, 89]. In [77], the authors have proposed a tool to automatically detect the topology of switched clusters for Ethernet based clusters. In [99], the authors have proposed an optimized algorithm for MPI_Allreduce operation that utilizes the network bandwidth in an efficient manner. In [62], the authors have proposed hierarchical algorithms for MPI_Scatter and MPI_Gather operations. However, these studies mostly focused on grid based systems that are loosely coupled through Ethernet or other proprietary networks. In our work, we have proposed topology-aware collective algorithms for large scale supercomputing systems that are tightly coupled through inter-connects such as InfiniBand. Resource management is again a well studied area in grids [96]. Researchers have also explored optimizing collectives on a BlueGene system based on SMP nodes connected in a 3D torus architecture in [119]. However, in our study, we have explored the effects of topology across multiple chassis, racks and across multiple levels of switches. In [43], the authors have proposed an efficient resource manager for cluster based systems. But to the best of our knowledge, there is no resource manager that allocates compute nodes in a contiguous manner, at the same time keeping the overall system utilization high.

3.4 Summary

The scale at which supercomputers are being deployed continues to increase each year. These clusters offer a vast amount of computing resources to application developers. However, it is necessary to design efficient software stacks that can fully leverage the performance benefits offered by such systems to allow applications to achieve good scalability. We have explored the need for detecting the topology of large scale InfiniBand clusters and we have leveraged this knowledge to design efficient topology-aware collective algorithms for Scatter and Gather collective operations. We have proposed a communication model to theoretically analyze the performance of collective operations on large-scale clusters. We have compared the performance of our proposed topology-aware algorithms against the default binomial tree approaches and we have observed benefits of almost 54% with micro-benchmarks.

Chapter 4: Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms

Chapter 3 of this thesis described the challenges involved in designing network topology-aware algorithms for the MPI_Gather and MPI_Scatter collective operations on homogeneous clusters. However, many supercomputing systems are also becoming increasingly heterogeneous, either in terms of processing power or, networking capabilities [1, 3]. Most MPI libraries, including MVAPICH2, operate in a compatibility mode when the networks are detected to be heterogeneous. But, this may degrade the performance of collective operations and parallel applications. Moreover, applications use a wide range of collective operations, requiring the designers of high performance middleware to support network topology-aware designs for multiple collective operations. In this chapter, we follow up on the work done in Chapter 3. We describe a framework to automatically detect the speed of an InfiniBand network and make it available to users through an easy to use interface. We also make design changes inside the MPI library to dynamically query this topology detection service and to form a topology model of the underlying network. We have redesigned the broadcast algorithm to take into account this network topology information and dynamically adapt the communication pattern to best fit the characteristics of the underlying network.

4.1 Motivation

The broadcast algorithms in MVAPICH2 currently use the *k-nomial* exchange for small and medium sized messages and the *scatter-allgather* algorithm for larger messages. Figure 4.1 shows the communication pattern for the *k-nomial* algorithm with 32 processes, with rank 0 as the root. The tree-based algorithms are useful for minimizing the latency for small and medium messages, because they can complete the entire operation in $(O(logN))$ steps, where N is the number of participating processes. However, the tree based algorithms also have an imbalanced communication pattern. As demonstrated in Figure 4.1, Rank 16 receives the data from the root and becomes the co-root of a sub-tree that includes all processes - ranks 17 through 31. Similarly, rank 4 is another co-root that receives data from the root, but is responsible for broadcasting the data for a smaller sub-set of processes - ranks 5 through 7. It is to be noted that if either ranks 16 or 4 get delayed in receiving the data from the root due to poor process placement or slower network link, this delay gets propagated down the tree and affects the latency of the entire broadcast operation. This effect becomes more significant at larger scales as the trees become more imbalanced. The scatter phase of the scatter-allgather algorithm also uses a binomial-tree algorithm. The imbalanced tree affects the latency of large message broadcasts in a similar manner.

In our previous work [71], we proposed designing topology-aware collective algorithms by creating internal communicators that mirror the network topology and scheduling the communication recursively across these communicators. While such an approach was shown to be an effective way to improve the latency of collective operations, as we can see in Table 4.1, it is possible that the nodes are allocated across several racks in the system. In such cases, it might not be beneficial to create rack-level communicators, because

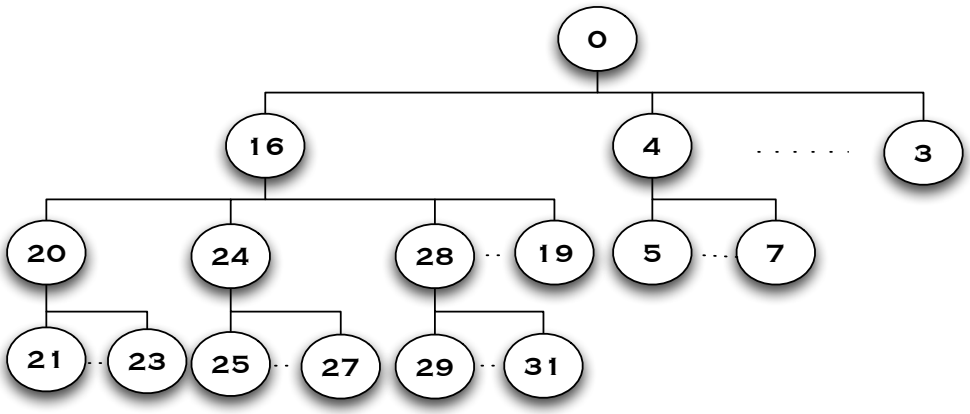


Figure 4.1: K-nomial (K=4) Tree Topology with 32 Processes

the number of racks are of the same order as the number of nodes for a given job. This observation leads us to the following challenge: *Is it possible to leverage the network topology information to re-organize the communication pattern of collective operations to improve their latency?*

Job Size	Average Hosts	Average Racks
<256	2	2
256 - 512	28	22
512 - 1K	58	40
1K - 2K	106	58
2K - 4K	224	74

Table 4.1: Average Job Distribution on Ranger in April 2011

4.2 Designing Network-Topology/-Speed Aware Broadcast Algorithms

As we saw in Section 2.3.1, MVAPICH2 uses some form of tree based pattern as the underlying communication scheme for almost all message sizes while performing the broadcast operation. As discussed in Section 4.1, delaying a message at one of the tree nodes with many children tends to have a greater impact on the performance of the broadcast

operation than delays created at the other nodes of the tree. In this context, we propose certain heuristics which are aimed at reducing the delay in this critical section of the broadcast algorithm.

The central idea is to to *map the logical broadcast tree to the physical network topology in-order to reduce the possible delays in the critical sections of the broadcast operation* described earlier. Note that we are not trying to achieve a perfect mapping of the logical tree to the physical topology. We are only trying to map certain critical regions of the broadcast which have been shown to have most impact on the performance of the broadcast operation as a whole. To this end, we re-order the ranks in the *node-level-leader* communicator described in Section 2.3.1 with information about the network in such a way as to allocate the fastest / shortest network links to the nodes in the critical region thus reducing possible network delays. We choose the classic *Depth First Traversal* (DFT) and *Breadth First Traversal* (BFT) methods as our heuristics to discover the critical sections of the broadcast operation as they lend themselves naturally to the tree based communication pattern used by the broadcast operation.

For each node-level leader in the broadcast, we create a new network aware communicator for it on an *on-demand* basis. When a node-level leader initiates a broadcast exchange operation for the first time, we create a new communication graph with the node level leader as the root and use either the depth first or the breadth first graph traversal method (as requested by the user) to traverse it. Figure 4.2 depicts the flow on control inside the MPI library with the addition of the topology aware collectives. Due to the nature of the traversal algorithms and the underlying communication graph, we will reach the nodes having most children before the other ones. For maintaining correctness of the application, the node on which the root of the broadcast operation resides is fixed. For each child process, we pick

a node that is closest to its parent process. We use an N x N delay matrix (N = number of node level leaders) to define the 'closeness' of two processes. We use the network topology detection service described in Section 2.8.1 to create the delay matrix representing the number of hops in the critical communication path. To compute the delay matrix related to heterogeneity, we use the rates of the IB HCAs on each node.

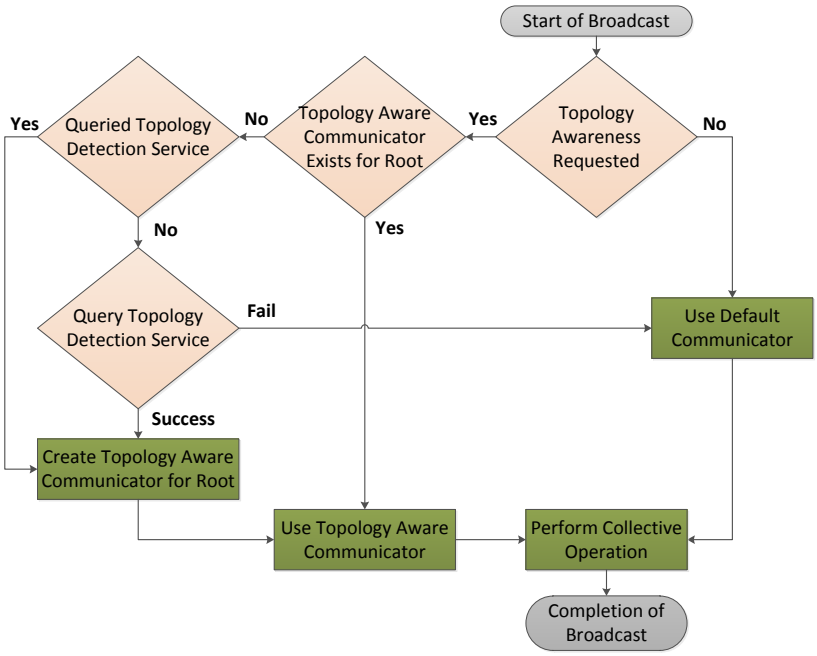


Figure 4.2: Control Flow with Topology Aware Collectives

4.2.1 Delay Matrix for Network Topology Aware Broadcast

It is common for the various processes belonging to an MPI job to get distributed across various switch blades on large networks, like the one used by the *Ranger* supercomputing system at TACC. As the current state of the art collective communication algorithms do not take the network topology into account, it is possible that the messages intended for nodes

in critical sections of the broadcast operation are required to traverse multiple hops before they reach the intended destination resulting in large delays and variable performance. In this context, we query the topology detection service to create the delay matrix based on the number of hops between any pair of node-level leaders. In order to avoid overwhelming the topology detection service, only Rank-0 of the node-level leader communicator queries it. A delay matrix is created locally and broadcasted to all other node-level leaders on the job. This is done only once at job initialization time and cached at each process for future use.

4.2.2 Delay Matrix for Network Speed Aware Broadcast

Heterogeneity adds a different dimension to the problem, requiring us to define the delay matrix in such a way as to take this into consideration. To this end, the node-level leaders perform an *Allgather* operation to obtain the link rate of the IB HCAs on all nodes taking part in the job. Once each node-level leader has the rate of all other nodes on the job, they create the delay matrix by taking the inverse of the of the speed of the HCA with the lower speed among a pair of nodes. This is done only once at job initialization time and cached at each process for future use.

4.3 Experimental Results

Our experiments were conducted on the *Ranger* supercomputer at the Texas Advanced Computing Center and on the *Glenn* supercomputer at the Ohio Supercomputing Center. Ranger is a 3,936 node cluster with a total core count of 62,976. Each node is a Sun-Blade x6420 running a 2.6.18.8 Linux kernel with four AMD Opteron Quad-Core 64-bit processors (16 cores in all) on a single board, as an SMP unit. Each node has 32 GB of memory. The nodes are connected with InfiniBand SDR adapters. Ranger also runs the latest version of the topology discovery service that we described in Section 2.8.1. Thus,

Ranger provides us a homogeneous environment to evaluate our network topology-aware broadcast algorithms.

Glenn, on the other hand, consists of multiple generations of InfiniBand adapters - SDR (8 Gbps) and DDR (16 Gbps). It is a 9,532 core (1,624 node) IBM Opteron cluster. The original system consists primarily of 877 dual socket, dual core 2.6 GHz compute nodes with 8 GB of RAM with an SDR two level InfiniBand fat-tree interconnect. The system was expanded to approximately twice its original size with the addition of 650 dual socket, quad core 2.5 Ghz compute nodes with 24 GB of memory connected via a DDR two level InfiniBand fat-tree. The SDR and DDR fat trees are connected via several links between the spine switches of the two trees. Thus, Glenn provides us a heterogeneous environment enabling us to evaluate our network speed-aware broadcast algorithms. Due to technical reasons, we were not able to get the topology discovery service up and running on the Glenn cluster. We use the OSU Benchmark suite and the WindJammer [84] application to evaluate the effectiveness of our algorithms.

4.3.1 Performance on Homogeneous Clusters

We evaluate the performance of the network topology-aware broadcast algorithms here. We ran the benchmark for different core counts and message sizes to evaluate how the proposed scheme scales as the size of the job as well as the message increases. We observed that the impact of network topology on the performance of smaller messages were not that significant. This follows from the fact that most of the time spent in the network for small messages get spent in the startup phase. But as the size of the message increases, we slowly begin to see the impact of the topology-aware schemes on the performance of the broadcast operation.

Figure 4.3 (a) shows this trend for various message sizes when the benchmark was run on 1,024 cores. The same trend exists for jobs of size 256 and 512 as well. We do not include these graphs due to their repetitive nature. Figure 4.3 (b), on the other hand, shows how the performance of the proposed topology-aware scheme scales as the size of the job increases for a fixed message size of 1 MB. Our experimental results show that, for large system sizes and message sizes, we get up to a 14% improvement in the latency of the broadcast operation using our proposed topology-aware scheme over the default one.

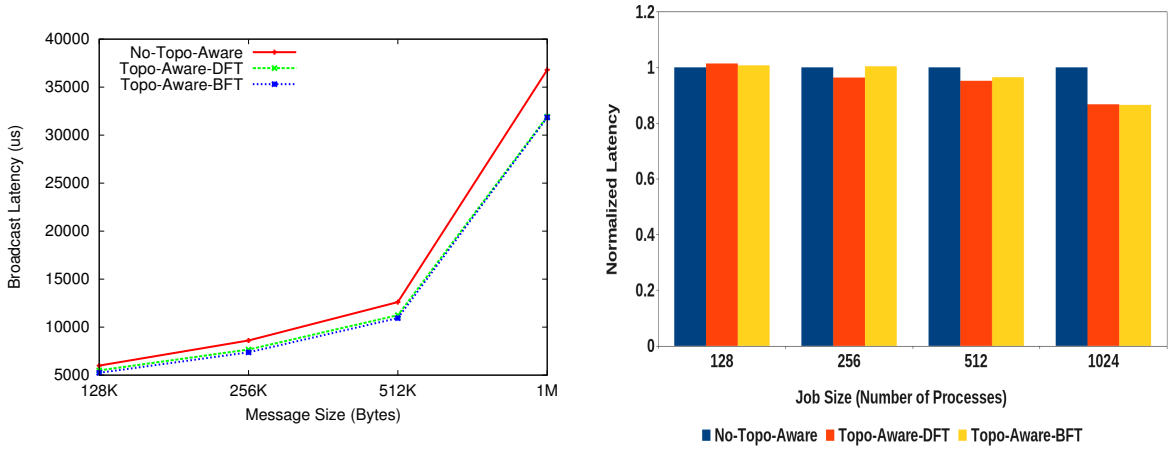


Figure 4.3: Impact of Network-Topology Aware Algorithms on Broadcast Performance for various: (a) Message Sizes at 1K Job Size and (b) Job Sizes

4.3.2 Performance on Heterogeneous Clusters

In this Section, we evaluate the performance of the network speed-aware broadcast algorithms. Figures 4.4 (a) and (b) show the performance comparison of the proposed network speed-aware algorithm and the default network-speed un-aware algorithm on a heterogeneous 256-core allocation on the Glenn cluster consisting of hosts with DDR or SDR

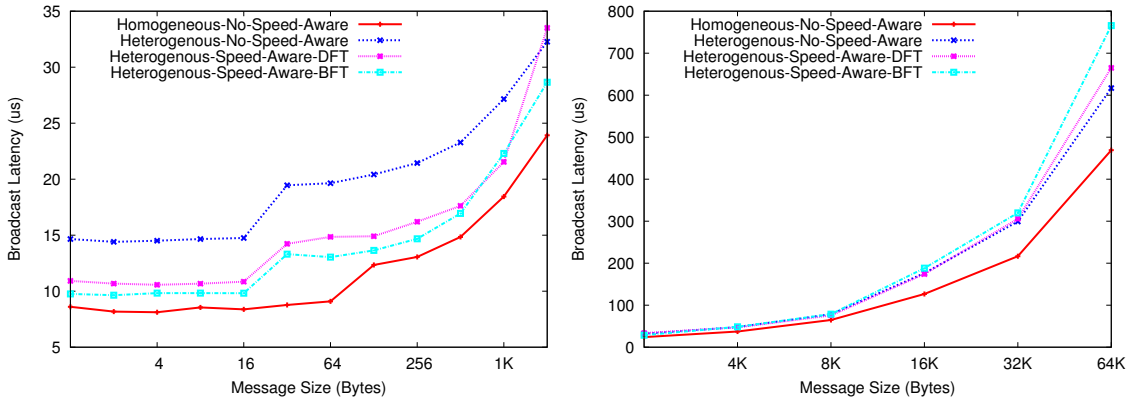


Figure 4.4: Impact of Network-Speed Aware Algorithms on a 256 process Broadcast Performance for: (a) Small Messages and (b) Medium Messages

HCAs. We also show the performance of the default algorithm on a homogeneous 128-core allocation on Glenn just as a point of reference. We see that the network speed-aware broadcast algorithms are able to perform two times better than the network-speed un-aware version on a heterogeneous allocation. At the same time, we also see that its performance is on par with the performance of default algorithm on the homogeneous allocation for small to medium sized messages. But as the size of the message increases, the operation becomes more bandwidth bound and hence more dependent on the raw speed offered by the network. Consequently, the performance of the network speed-aware algorithm on the heterogeneous cluster begins to perform worse than that of the default algorithm on the homogeneous allocation. The slight degradation in performance seen for messages less than 128 bytes is due to difference in size of the messages that can be sent in an inline fashion by the IB HCA. While the newer DDR HCA is able to handle an inline size of 128 bytes, the older SDR HCA is only able to handle a maximum inline size of 64 bytes.

4.3.3 Application Level Results

In this Section, we look at the performance benefits obtained for the WindJammer application described in Chapter 2 with the network topology-aware version of the broadcast algorithm on *Ranger* with 128 and 256 processes. Figure 4.5 depicts the performance of the default network-topology un-aware algorithm as well as the proposed network topology-aware algorithm for various system sizes. As we saw with the micro-benchmark results on homogeneous clusters in Section 4.3.1, the impact of the network topology-aware version of the algorithm is not felt that much at smaller system sizes. But as the system size scales, the improvement seen in performance also increases. We see that, at 256 processes, an improvement of up to 8 % is seen at the application level by using the topology-aware version of the broadcast algorithm over the default one.

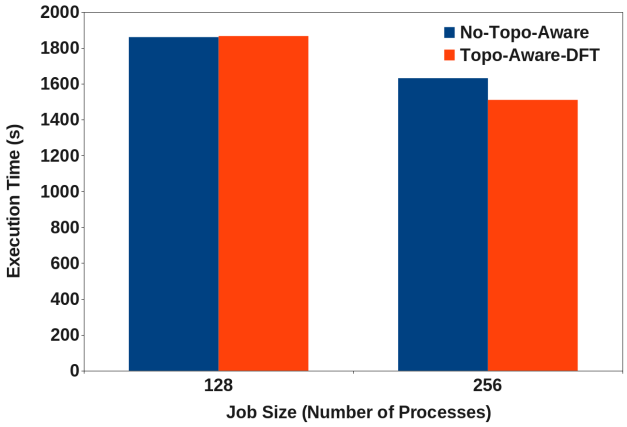


Figure 4.5: Impact of Network-Topology Aware Algorithms on WindJammer Application

4.3.4 Overhead of Creating Topology Aware Communicator

In this Section we analyze the various overheads involved in creating the topology aware communicator. Table 8.1 shows the time spent by the MPI library in various phases of topology discovery for various number of compute nodes. The total job size would be

Number of Nodes	8			16			32		
Phase	Initial Call	Ensuing Call (Same Root)	Ensuing Call (Different Root)	Initial Call	Ensuing Call (Same Root)	Ensuing Call (Different Root)	Initial Call	Ensuing Call (Same Root)	Ensuing Call (Different Root)
Discover Connected Switches	5.1	NA	NA	4.2	NA	NA	6.9	NA	NA
Compute Delay Matrix	5.7	NA	NA	23.7	NA	NA	103.1	NA	NA
Broadcast Delay Matrix	0.5	NA	NA	4.8	NA	NA	5.9	NA	NA
Create Topology Aware Communicator	0.13	NA	0.13	1.5	NA	1.5	2.4	NA	2.4
Total Time	11.6	0	0.13	30.8	0	1.5	115.5	0	2.4

Table 4.2: Overhead of Topology Discovery in MPI Library (Time in ms)

128, 256 and 512 respectively on an 8 core system and 256, 512 and 1,024 respectively on a 16 core system. As we can see, most of the cost involved in topology discovery is a one time cost making the approach scalable.

4.4 Related Work

Extensive research has been conducted around the mapping problems in parallel and distributed computing. The general idea being to map a task graph to a network graph while minimizing the overhead of communication and balancing computational load. This problem can be reduced to a graph-embedding problem which has been proved to be $\mathcal{NP}-Complete$ [27, 41].

From the 70's through the 90's, researchers proposed different solutions to solve the mapping problem based on heuristic algorithms (*local optimization* [74, 97], *greedy* [19, 53], *branch-and-bound* [112]) and physical optimization algorithms (*graph contraction* [20, 82], *simulated annealing* [28, 115], *neural networks* [81], *genetic algorithm* [32]). But most of these works have targeted interconnect topologies such as hypercubes, array processors or shuffle-exchange networks, while modern system topologies are mainly meshes, tori and fat-trees. With Petascale clusters, communications become the bottleneck. Thus, it

is critical to improve the mapping of communication graphs on these interconnect topologies to reduce the impact of contention on large scale parallel applications.

Recent works of Bhatele [22–24] et. al. demonstrated the importance of topology-aware mapping for communication bound applications on tori networks. They have presented a framework for automatic mapping of parallel applications using a suite of heuristics that they developed. Their framework was based on two steps: 1. Obtaining the communication graph using profiling libraries, 2. Based on the communication patterns and topology information, apply heuristics to obtain mapping solution. The topology information was obtained through system calls and their work mainly focused on torus networks.

Here, we have presented our approach to automatically obtain topology information and adapt MPI collective operations. Our work directly focuses on commodity networks and takes into account heterogeneous network speeds.

4.5 Summary

We have re-designed the communication schedule of the Broadcast operation in a network topology and speed-aware manner. Our experimental results show that, for large homogeneous systems and large message sizes, we get up to a 14% improvement in the latency of the broadcast operation using our proposed topology-aware scheme over the default one at the micro-benchmark level. At the application level, we see up to an 8% improvement in total application performance as we scale the job size up. For a heterogeneous SDR-DDR cluster, we see that the proposed network speed-aware algorithms are able to deliver performance on a par with homogeneous DDR clusters for small to medium sized messages. We also see that the network speed-aware algorithms perform 70% to 100% better than the naive algorithms when both are run on the heterogeneous SDR-DDR InfiniBand cluster.

Chapter 5: Design of Network-Topology-Aware Placement of Processes

Chapters 3 and 4 describe the design of network topology-aware solutions for some of the popular collective operations in use by scientific computing applications. However, as we identify in Section 1.1, multiple classes of high performance computing applications also use MPI point-to-point communication to transfer large amounts of data between various processes. For such applications, it is still an open challenge as to whether we can obtain significant performance improvement by scheduling point-to-point messages in a network topology-aware manner. Several such applications also tend to transfer messages in a structured form such as a 2D, 3D or 4D stencil. The assignment of ranks to compute cores also has a significant impact on performance for these applications as certain mappings will be more advantageous than others. In this chapter, we take up both these challenges depicted in Figure 1.2. We propose the design of a network topology-aware MPI communication library capable of leveraging the topology information generated the topology service described in Section 2.8.1 to improve communication performance of point-to-point operations.

5.1 Driving Example

We employ a simple case-study to clearly motivate the need for network-topology-aware process mapping to improve the communication overheads in a 3D stencil (nearest-neighbor) code. In our example, each process communicates with 6 nearest neighbors, across three dimensions. Figure 5.1 presents the network topology of a 4 node allocation on the Hyperion cluster. Figure 5.2(a) shows the default process-to-node mapping of a

32 process MPI job (2x4x4 grid). This figure also denotes the number of switch hops incurred between the communicating processes. For example, communicating peers that are within the same leaf switch will incur 1 hop, whereas those that are mapped across different switches may incur 3 hops. Since the 3 hop latencies are higher than the intra-switch operations, improving the process placement pattern to lower the 3-hop exchanges could potentially improve the communication overheads.

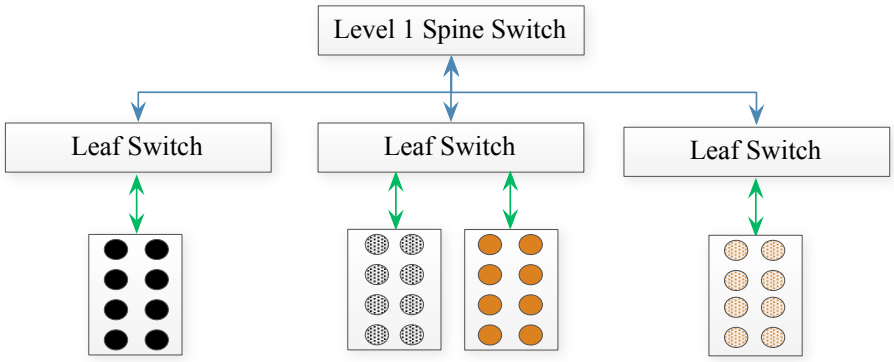
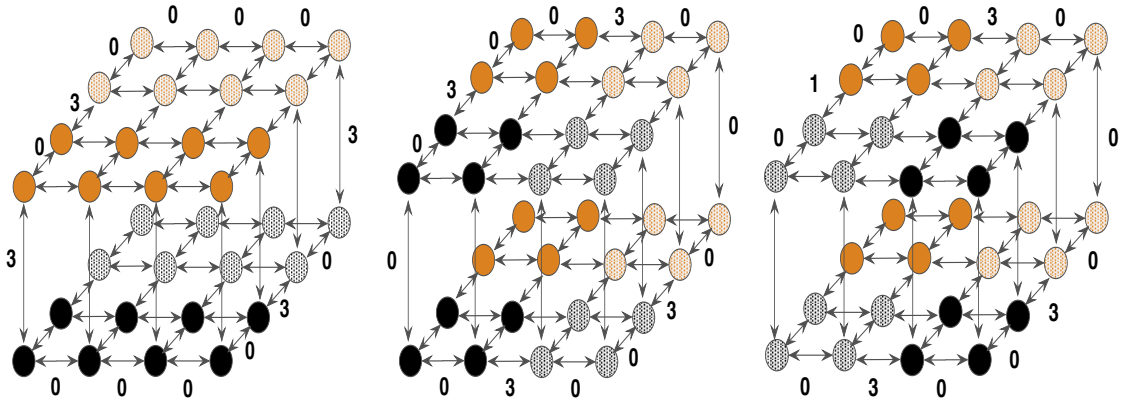


Figure 5.1: Sample network hierarchy



(a) Default process-to-node mapping used by supercomputers (b) Node-aware manual mapping (c) Node and network-aware manual mapping

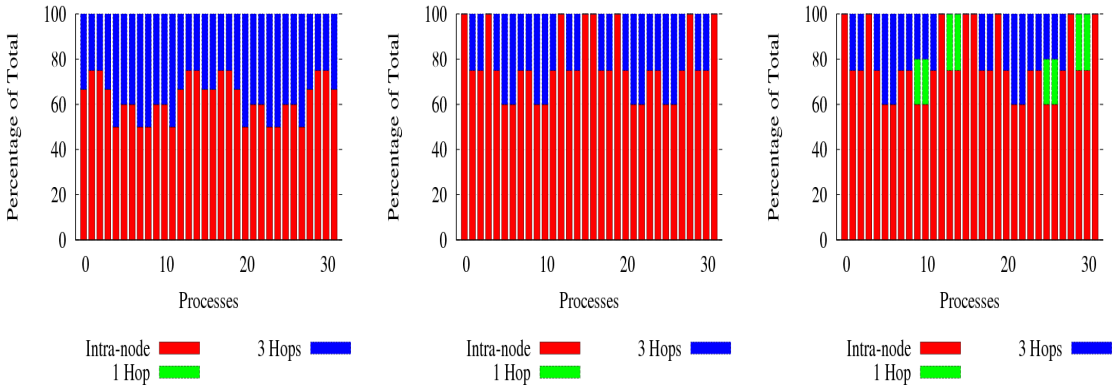
Figure 5.2: Communication pattern with various manually created process to host mappings

	Default Figure 5.2(a)	Optimized for intra-node Figure 5.2(b)	Optimized for intra-node and inter-node Figure 5.2(c)
Intra-node	80	96	96
1 Hop	0	0	8
3 Hops	48	32	24

Table 5.1: Split of messages based on number of hops

In Figures 5.2(b) and 5.2(c), we show that it is possible to manually control the process mapping pattern to achieve this objective. In Figure 5.2(b), we show that we can increase the amount of intra-node communication by mapping 2x2x2 process sub-grids within a node. In Figure 5.2(c), we show that this can be further improved by leveraging network topology information. Table 5.1 shows the split up of the communication exchanges based on the number of network hops. In the default case, our example required as many as 48 3-hop exchanges. We could improve this to 32 3-hop transfers by considering the intra-node communication patterns. Further, we could lower this to 24 3-hop transfers, by mapping the processes in a network-topology-aware manner. In Figures 5.3(a), 5.3(b) and 5.3(c), we graphically compare the number of intra-node, 1-hop and 3-hop transfers across the three process placement patterns. We can observe that the number of 3-hop transfers are much lower in Figures 5.3(b) and 5.3(c), when compared to the default case. We also see that we have fewer 3-hop transfers and more 1-hop transfers in Figure 5.3(c), when compared to Figure 5.3(b). Although it is possible to manually map the processes in a network-topology-aware manner for such small cases, it becomes prohibitively complicated as we go to larger scale real-world application runs. However the benefits from such re-organization also increase as we scale to thousands of nodes. This strongly motivates the need for a scalable

topology service that can support a network-topology-aware MPI library and can be used by applications in a portable fashion.



(a) Default process-to-node mapping used by supercomputers

(b) Node-aware manual mapping

(c) Node and network-aware manual mapping

Figure 5.3: Split-up of physical communication in terms of number of hops with various manually created process to host mappings

5.2 Design of Network-Topology-Aware MPI Library

We describe the various design changes made to the MVAPICH2 [10] MPI library in order to support network-topology-aware mapping of processes. We create multiple communicators to aid us in mapping the communication topology to the network topology. We describe these communicators in Section 5.2.2 and their use in Section 5.2.4. We design an abstraction layer called the Domain Graph Interface (DGI) to isolate the upper level users from the internal implementation details. In a similar fashion, we use a predefined set of interface routines to access external graph partitioners such as ParMETIS and Jostle. This allows us to easily switch between alternate graph partitioners without making major changes to our network-topology-aware process placement techniques.

5.2.1 Design of Topology Information Interface

The Topology Information Interface (TII) is designed to abstract the details of the underlying topology from the MPI library - be it inter-node network topology or intra-node processor topology. All of the APIs exposed by the interface depend on the network topology detection service described in Section 2.8.1. A brief description of the more important APIs is given below:

- `tii_get_network_depth`: Retrieves the depth of NJ tree described in Section 2.8.1
- `tii_num_switch_clusters_at_depth`: Retrieves the number of unique switch clusters the job as a whole is connected to at a given level of the NJ tree
- `tii_connected_switch_cluster_at_depth`: Retrieves the unique ID of the switch cluster a particular rank is connected to at a given depth of the NJ tree

Currently, the TII does not take the intra-node processor topology into account and returns equal communication costs for all processors within a compute node. However, we are working on introducing this support into the TII by leveraging features provided by external libraries such as the Portable Hardware Locality (hwloc) [29].

5.2.2 Communicator Design to Support Graph Mapping

As shown in Figure 5.4, we create multiple sub-communicators to reflect the network topology of the system. We build upon the idea proposed in our earlier work [71]. Network topology information is obtained using the APIs provided by the TII described in Section 5.2.1. To prevent overwhelming the topology discovery service, MPI rank 0 performs the required queries for the entire job and broadcasts it to rest of the processes. Note that in the MVAPICH2 MPI stack, the process with rank 0 will also be a node-level leader.

Rank 0 will query and obtain the depth of the NJ tree, the number of switch clusters at a particular level of the tree and the unique ID of the switch clusters at various levels of the NJ tree each node is connected to. All of these parameters are identified by the network topology detection service using the NJ algorithm described in 2.8.1. If there are multiple switch clusters at a level, we partition the communicator to create separate communicators for switch clusters as depicted in Figure 5.4. We use the unique ID of the switch cluster as the `color` to split the communicator.

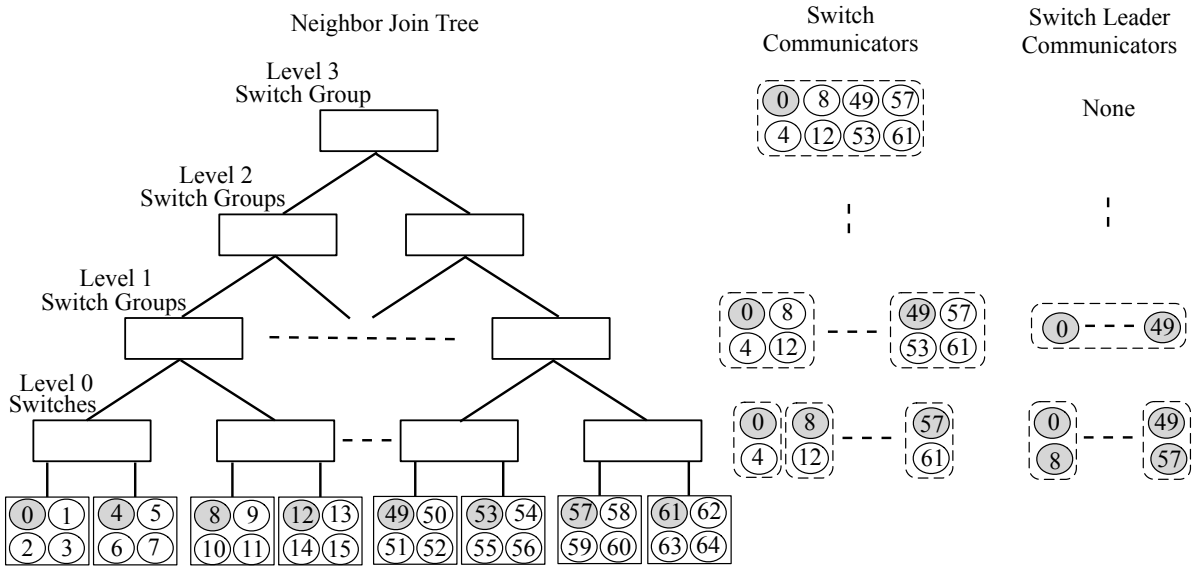


Figure 5.4: Communicator design to support network-topology-aware process placement

5.2.3 Single Step Mapping

The single step mapping module forms the base for all of our designs. Other mapping approaches such as the multi step mapping described in Section 5.2.4 rely on the single

step mapping module to solve the graph partitioning problem. This module accepts a communicator and topology information as inputs and returns the network optimized problem domains embedded in the communicator object as output.

Once the library has obtained the necessary information regarding the communication topology of the application through the DGI, it attempts to minimize the amount of network communication between various processes by considering it as a k-way graph partitioning problem [140]. The MPI library relies on external graph partitioners to map the communication pattern of the application to the topology of the underlying network. We use both Jostle and ParMETIS for this. Most graph partitioners require the user to pass the relative weights of the various edges in the communication graph as input. We explored various metrics like network hops, frequency of communication, volume of communication and frequency / volume of communication weighted using network hops. Initial experimental results showed that In volume of communication is the most suitable metric.

The relative communication volume between peers can either be passed onto the MPI library through the DGI, or it can be obtained by executing the job with some special profiling flags that instruct the MPI library to record the communication frequency between various processes in a file. During subsequent executions, the MPI library automatically uses this file to identify the frequency of communication between peers. If this file is absent or inconsistent with the current job size, and if the application did not provide it through the DGI, the library assumes that all processes communicate equally with their peers. The library then transforms the user input along with the network topology information, gathered through the TII, into a form that is acceptable for the interface to external graph partitioners.

As a single execution of the graph mapping library may not yield a balanced mapping of high quality, the single step mapping module calls the external graph mapping library

multiple times until it yields a domain of sufficient quality for the application. The level of quality required can be defined by the user through an environment variable.

5.2.4 Multi Step Mapping

The multi-step mapping module uses the single step mapping module described in Section 5.2.3 as well as the various communicators described in Section 5.2.2 to effectively map the various processes that are part of the MPI job onto the supercomputing system. We designed two different network-topology-aware process mapping schemes. For reference, Figure 5.4 identifies the location of each rank that is mentioned in the following discussion. Note that we assume that level 1 is the top level for purposes of describing the two schemes. In both schemes, the node level leaders (0, 4, 8, 12, 49, 53, 57 and 61) gather data from the other intra-node processes (0 will gather from 1, 2 and 3).

Scheme 1: In this scheme, we follow a top-down approach to the graph partitioning problem, recursively partitioning at each switch level. The goal is to reduce the number of long distance communications over the network and to concentrate communication within individual switch clusters as much as possible. All data pertaining to the application communication pattern like vertices, edges, edge weights, etc. are gathered by the members of the top level (say level 'n') switch leader communicator (0 and 49). The leaders then call the single step mapping module with this aggregated information in order to reduce the number of inter-domain communication. Inter-domain communication in the context of the top level switch leaders correspond to the communication that spans the longest distance over the network for the job in question (5 hops in this example). Once the partitioning is done, the top level leaders scatter the data to the members of switch communicators at that level (0 scatters to 4, 8 and 12; 49 scatters to 53, 57 and 61). The members of switch-leader communicators at level 'n-1' (0 and 8; 49 and 57) aggregate this information through the

switch communicator at that level (0 gathers from 4; 8 gathers from 12; 49 gathers from 53 and 57 gathers from 61). These leader processes then call the single step mapping module again to reduce the inter-domain communication at that level (3 hops in this example). The process is repeated until we reach the set of communicators representing the switches at the lowest level.

Scheme 2: The second scheme is built atop scheme 1, but contains a critical difference in that we partition the data between all node level leaders (0, 4, 8, 12, 49, 53, 57 and 61) first. Once this step is done, we aggregate all the information contained in the vertices inside one node to create a new `super-vertex`. The idea is to not allow the individual vertices to be moved from one node to another independently. We then perform the same steps outlined in scheme 1 with this new `super-vertex`. The goal in this approach is to have as much communication as possible within one node and prevent it from being disturbed by further rounds of partitioning.

5.3 Experimental Results

We used multiple high performance computing systems to obtain the results:

Ranger: Ranger is comprised of 3,936 16-way SMP compute nodes providing a total of 62,976 compute cores. Each core operates at 2.3 GHz and has 32 GB of memory with an independent memory controller per socket. Ranger has two 3,456 port SDR Sun InfiniBand Datacenter switches at its core. The interconnect topology is a 7-stage, full-CLOS fat tree.

Hyperion: This is a 1,400-core testbed where each node has eight Intel Xeon (E5640) cores running at 2.53 Ghz with 12 MB L3 cache. Each node has 12 GB of memory and an MT26428 QDR ConnectX-2 HCA. It has a 171-port Mellanox QDR switch, with 11 leafs, each having 16 ports. Each node is connected to the switch using one QDR link. Although

Number of Nodes (Processes)		16 (128)	32 (256)	64 (512)	128 (1,024)
Splitup of Time for Topology Mapping	Discover Network Topology	0.006	0.007	0.011	0.018
	Communicator Creation	0.006	0.008	0.578	0.773
	Create Topology Aware Mapping	0.180	0.228	0.635	1.474
	Total Time	0.192	0.243	1.224	2.265
Total Execution Time of *MILC*		460.66	462.69	590.66	689.36
Time for Topology Mapping as a Percentage of *MILC* Execution Time (%)		0.04%	0.05%	0.21%	0.33%

Table 5.2: Overhead of Topology Discovery and Graph Mapping (Time in seconds)

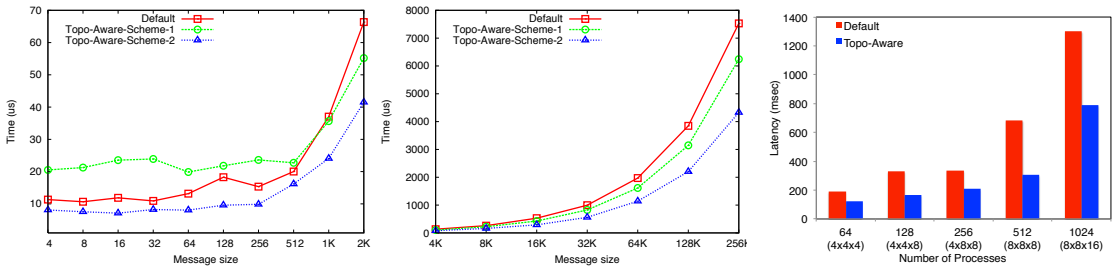
Hyperion does have additional compute cores, they are not homogeneous in nature. Hence we restrict ourselves to the nodes described above.

5.3.1 Overhead of Topology Discovery and Graph Mapping

Table 8.1 shows the time spent by the MPI library in various phases of topology discovery and graph mapping for different number of compute nodes for the *MILC* code. As we can see, the overhead caused due to the network-topology-aware graph mapping scheme only forms a very small percentage of the total time taken by the application for varying job sizes. Another point to note is the amount of time taken for the communicator creation phase. This is orthogonal to our work. Several researchers have already proposed solutions [39, 116] for this which can be easily integrated into our design. We are currently working on enhancing the design of the topology mapping module to make it more robust and scalable.

5.3.2 Performance with 3D stencil benchmark

The processes in the benchmark are mapped onto a 3D grid and each process talks to its neighbors in each dimension (6 neighbors). In every step, each process posts MPI_Irecv operations for all of the messages it expects and then posts all of the MPI_Isend calls. It waits for all of the transfers to complete with a single MPI_Waitall call. We compare both schemes we have outlined in Section 5.2.4 against the case that does not have any network-topology-aware optimizations. Figures 5.5(a) and 5.5(b) compare the performance of the



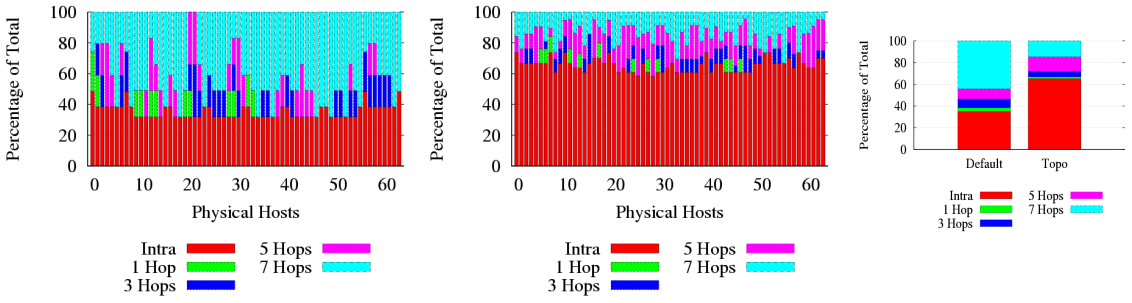
(a) Small Messages on Ranger (b) Large Messages on Ranger (c) Performance at scale on Hyperion

Figure 5.5: Latency Performance of 3D Stencil Communication Benchmark

3D stencil benchmark for small and large messages respectively at 512 processes. We see that for large messages both of the schemes perform better than the default scheme, with scheme-2 giving up to 40% improvements. But for smaller messages we see that scheme-1 performs worse when compared with the default case. Analysis of the performance from a network perspective showed that although we were decreasing the number of long distance network communications with scheme-1, the percentage of intra-node messages were also decreasing leading to poorer overall performance. We do not show the graphs showing the split up of physical communication due to lack of space. Scheme-2 on the other hand was able to increase the percentage of intra-node communication as well as decrease the total number of long distance network communication leading to the best benefits. Hence we use scheme-2 for all of our experiments at the applications level. To verify the scalability of our design, we compare scheme-2 with the default scheme at various system sizes for a fixed message size of 64 KB. As we can see from Figure 5.5(c), the topo-aware scheme consistently performs better at all system sizes. We do not show the trends on Ranger as they are similar.

5.3.3 Impact of Network Topology Aware Task Mapping on the Communication Pattern of AWP-ODC

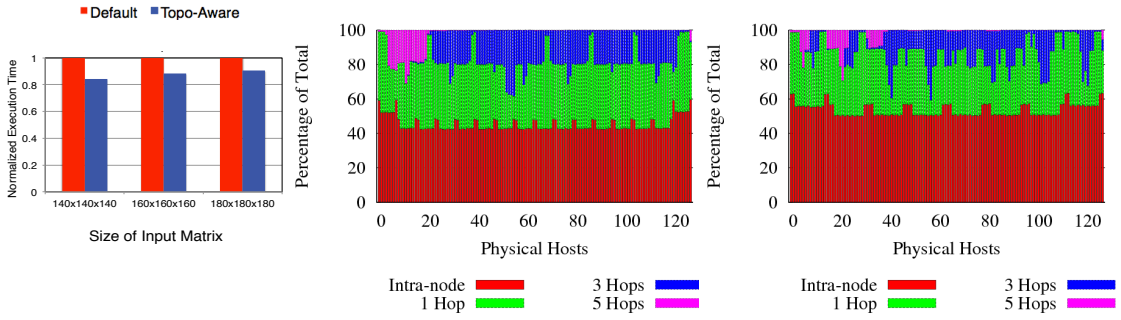
To examine the effectiveness of our topo-aware process placement scheme further, we use AWP-ODC. We analyze the physical communication pattern in terms of number of hops for the default and the automatic topo-aware process placement scheme. Figures 5.6(a) and 5.6(b) depict the communication pattern in terms of number of hops for the default and topo-aware scheme, respectively. As we can clearly see, the topo-aware process placement scheme is able to improve the intra-node communication percentage from 33% to 66%. It is also able to reduce the number of 7, 5, 3 and 1 hop inter-node exchanges. Figure 5.6(c) summarizes the gains obtained in reducing the number of long distance communication. Although the topo-aware scheme does lead to modest gains in the performance of the application, further gains are limited by an inherent imbalance in the application due to boundary condition updates. Nevertheless, this shows the potential benefit that a topo-aware process placement can have.



(a) Default (b) Topo-Aware (c) Summary

Figure 5.6: Overall split up of physical communication for AWP-ODC based on number of hops for a 1,024 core run on Ranger

5.3.4 Impact of network topology on performance of *Hypre*



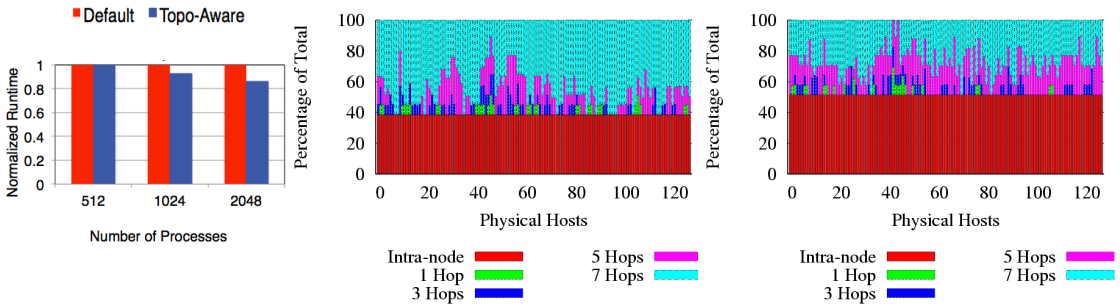
(a) Performance with varying matrix sizes (b) Default for 180x180x180 matrix (c) Topo-Aware for 180x180x180 matrix

Figure 5.7: Overall performance and Split up of physical communication for *Hypre* based on number of hops for a 1,024 core run on Hyperion

Hypre is an open-source, high performance and scalable package of parallel linear solvers and preconditioners. It mostly uses small to medium sized messages for communication. We evaluate the AMG pre-conditioner inside *Hypre* using the *new_ij* benchmark that is available with the *Hypre suite* for various input matrix sizes at a system size of 1,024 processes on Hyperion. We used the following parameters as input to run the benchmark "`-27pt -pmis -interptype 6 -Pmx 4 -amg_max_its 100`". Figure 5.7(a) compares the normalized performance of the topo-aware scheme with the default scheme for different matrix sizes. As we can see, the topo-aware scheme gives between 10% to 15% improvement over the default scheme that is unaware of network topology. We further analyze the physical communication characteristics of the application to gain insights into the performance benefits. Figures 5.7(b) and 5.7(c) depict the split up of physical communication in terms of number of hops for the default and topo-aware cases, respectively.

As we can see, we are able to increase the percentage of intra-node communication (from 45.00% to 53.28%) uniformly across all of the physical hosts involved in the job. We can also see, in the inter-node communication, we are able to reduce the number of long distance 5 and 3 hop transfers in exchange for an increase in the short distance 1 hop transfer. Due to scheduling conflicts, we were unable to run *Hypre* at large scales on Ranger.

5.3.5 Impact of network topology on performance of *MILC*



(a) Performance for varying system sizes

(b) Default for 2048 core run

(c) Topo-Aware for 2048 core run

Figure 5.8: Overall performance and Split up of physical communication for *MILC* on Ranger

We run the *ks_imp_dyn* code which simulates dynamical Kogut-Susskind fermions with the input that is available for it from the NERSC website [135]. For the cases we ran, *MILC* mostly used medium messages in the range for 4 KB - 64 KB. Figure 5.8(a) compares the normalized performance of the topo-aware and default schemes for various system sizes on Ranger. As we can see, the performance improvements seen increases as the size of the system increases, which is a very encouraging result. Overall we see that the topo-aware versions give up to a 10% improvement in performance when compared to the default scheme at at 1,024 cores and up to a 15% improvement at 2,048 cores. We analyze the

communication pattern in terms of number of hops for the topo-aware and default versions. Figures 5.8(b) and 5.8(c) depict the communication split up in terms of number of hops for the default and topo-aware versions, respectively, at 2,048 cores on Ranger. As we can see, the topo-aware version is able to increase the percentage of intra-node communication (from 44.12% to 58.77%). We are also able to significantly reduce the number of long distance (7-hop) communication from 73.10% of inter-node exchanges to 48.67%. The topo-aware scheme brings in most of this communication to inside a node and the remaining to lower hop location by intelligent placement of MPI ranks. Due to sparse nature of the allocation and the communication pattern of *MILC* (128 nodes were allocated on up to 109 different leaf switches, these switches were connected up to 66 different switch clusters at level 1 of the NJ tree, leaving only 2 or 3 nodes per level 2 switch cluster), we were not able uniformly reduce the number of 5 hop and 3 hop exchanges.

5.4 Related Work

Extensive research has been conducted around the mapping problems in parallel and distributed computing. The general idea is to map a task graph to a network graph while minimizing the overhead of communication and balancing computational load. This problem can be reduced to a graph embedding problem which has been proved to be $\mathcal{NP} - Complete$ [27, 41]. Researchers have proposed different solutions to solve the mapping problem based on heuristic algorithms [74, 78, 112] and physical optimization algorithms [20, 32, 81, 115]. But most of these works targeted interconnect topologies such as hypercubes, array processors or shuffle-exchange networks, while modern system topologies are mainly meshes, tori and fat-trees. With petascale clusters, it is critical to improve the mapping of communication graphs to improve communication overheads of large scale parallel applications.

Bhatele et al. [22, 23] presented a framework for automatic mapping of parallel applications using a suite of heuristics. Their framework was based on obtaining the communication graph using profiling libraries and applying heuristics to obtain a mapping solution. The topology information was obtained through system calls. These works demonstrated the importance of topology-aware mapping for communication bound applications on tori networks and proprietary systems like BlueGene.

Mercier and Jeannot proposed the *TreeMatch* algorithm [61] which provides a near-optimal placement of MPI processes on NUMA architectures. They evaluated this algorithm using two different metrics [85] to assign weights to the edges: the communication volume and frequency. Nevertheless, this work does not take into account the physical topology of the network.

Rashti et al. [113] proposed a network discovery module based on *ibtracert* and used *Scotch* [103] to match the communication graph of the application to the physical topology. Hoefler et al. [51] proposed different topology mapping algorithms available for different network topologies. However, these approaches either relied on *ibtracert* and *ibnetdiscover*, which are privileged operations or, as discussed in Chapter 1, were not scalable.

Here, we propose a scalable network discovery service for InfiniBand available at the user level. With this service, we design a network-topology-aware MPI library to provide a topology aware mapping of the MPI processes.

5.5 Summary

We design a network-topology-aware MPI library, and we have shown that its use can improve performance of real-world applications and libraries at scale. Micro-benchmark level evaluations showed that the proposed topo-aware MPI can enhance the performance of all message sizes by up to 40%. With *MILC*, we see up to 6% and 15% improvement in

total execution time on 1,024 cores of Hyperion and 2,048 cores of Ranger, respectively. On Hyperion we were able to increase the percentage of intra-node communication from 29.04% to 43.52%. On Ranger the percentage of intra-node communication went up from 43.51% to 57.95%. In both cases, we reduced the number of 7, 5, 3 and 1 hop exchanges. We also got up to a 15% improvement in the runtime of the *Hypre* linear system solver using 1,024 processes for varying sizes of the input matrix on Hyperion.

Chapter 6: Designing Topology Aware Communication Schedules for Alltoall Operations in Large InfiniBand Clusters

In this chapter, we follow up on our work done in Chapters 3 and 4. We focus on the important MPI_Alltoall collective operation. We consider the network topology of the systems and propose multiple schemes to design a network-topology-aware Alltoall primitive. We carefully analyze the performance benefits of these schemes and evaluate their trade-offs across varying application characteristics.

Evaluating the network congestion and making use of this information to optimize the performance of communication libraries is an active area of research. Prisacari et al. have proposed a congestion-aware Alltoall schedule [107]. In this study, the authors have assumed that the allocation of compute nodes in a supercomputing system is perfectly balanced. However, experience on large systems with a diverse job size mix dictates that this is not necessarily the case, particularly on busy systems where the need to maintain high utilization allowing backfill opportunities can be counter to topology-based scheduling considerations. To verify our claim, we conducted a post-mortem analysis on job scheduling distributions on a recently deployed 6,400-node cluster named Stampede [132] at TACC to ascertain potential load imbalance at the leaf-switch level. Stampede is currently using the SLURM multi-factor priority scheduler to allocate compute resources. We calculate the switch balance factor as the ratio between the number of hosts on the leaf switch with maximum variance from ideal and the average number of hosts per leaf switch. The experimental results presented in Figure 6.1 cover a range of jobs over a three month period and

demonstrate that typical parallel jobs at these sizes (1K, 2K, and 4K) tend to have fairly imbalanced node allocation patterns. Further, the model presented in [107] was based on a Fat-Tree network with one root switch. However, typical large-scale Fat-Tree networks are likely to have many switches at the top-most level (e.g. Stampede has eight, 648-port core switches [132]). Moreover, the authors in [107] analyzed the benefits of their approach through simulations. In contrast, the goal of our work is to demonstrate deployable algorithms with concrete analysis of the benefits with real scientific applications or kernels on large scale supercomputing systems.

Performing network-topology-aware scheduling of Alltoall operations on large-scale systems requires in-depth knowledge about the paths that various messages will take in the network. This requires the designer to have access to the network topology information. Researchers have explored the design space of detecting the network topology of large scale InfiniBand networks and using this information to study the impact of network contention in large scale supercomputing systems [51, 113]. Previously, we proposed a network topology discovery service [130, 131] that can be used by the MPI library to dynamically query the InfiniBand network's topology and explore the benefits of designing topology-aware algorithms for the Alltoall operation. Some of the existing approaches have a time and space complexity of $\mathcal{O}(N_{\text{hosts}}^2)$ [113, 130] while others require administrative privileges to get the InfiniBand topology information that normal users of a supercomputing system will not have [51]. As supercomputing systems aim to scale to millions of processes, such techniques impose serious scalability and usability issues. These issues lead us to the following broad challenge: *Can we design scalable topology aware communication schedules for Alltoall communication in large InfiniBand clusters?*

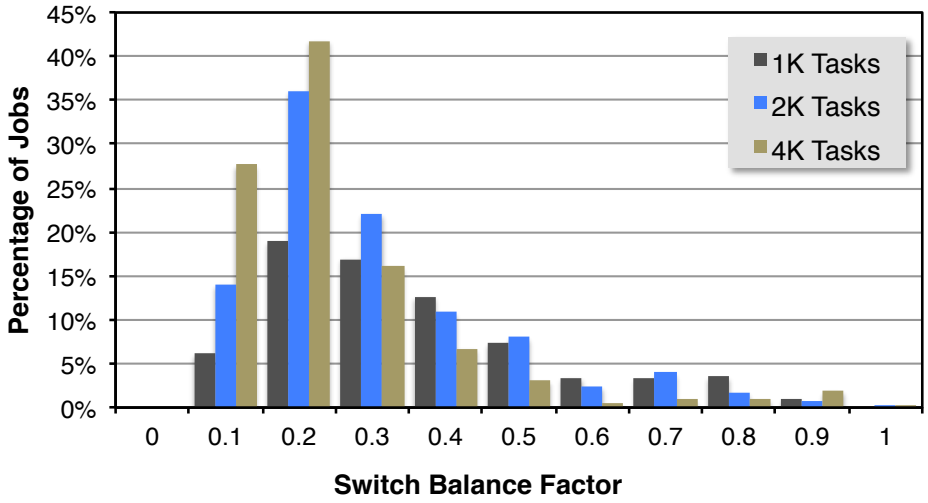


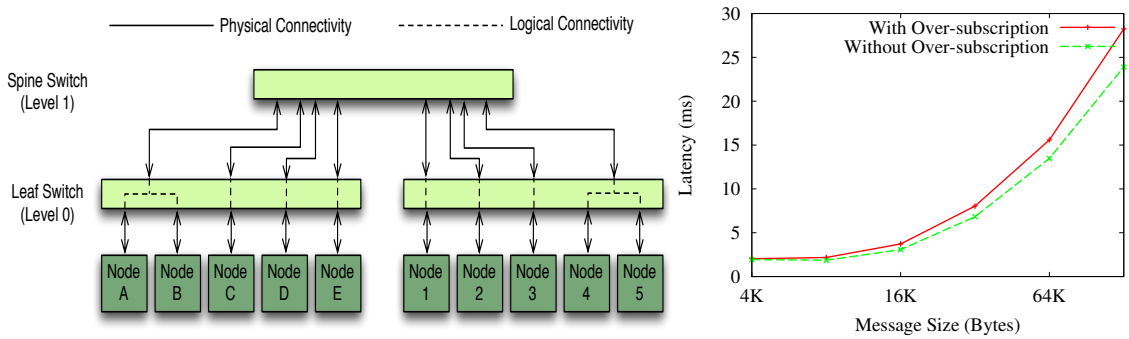
Figure 6.1: Imbalance in allocation of hosts on Stampede

6.1 Motivation: Congestion in Large InfiniBand Networks

In this Section, we study the various factors that can lead to congestion in large InfiniBand networks. The topology of the network often determines the utilization of network links and hence impacts the level of congestion observed in the network. The multi-dimensional torus and Fat-Tree are two of the most popular interconnects adopted across modern supercomputing systems. Of these, the Fat-Tree network architecture is richer in terms of available network resources. A full Fat-Tree network has the potential to provide a conflict free path between two disjointed host pairs in the system. However, a full Fat-Tree can be prohibitive in terms of total cost of ownership as well as being "over-kill" in terms of the performance it offers to end applications. Consequently, many supercomputing centers tend to be designed based on a partial Fat-Tree network topology with varying over-subscription factors across different levels of the tree.

In Figure 6.2(a), we demonstrate a typical communication network that has a 5:4 over-subscription on the links connecting the leaf switch and the spine switch. The Stampede supercomputing system at TACC [132] has been designed with a similar level of over-subscription. We observe that such over-subscription patterns can lead to multiple hosts sharing the same network link for certain communication operations. In our example, suppose nodes A,B and 4,5 are involved in a communication operation (Figure 6.2(a)). This communication pattern results in the same communication link servicing multiple transfers across the two leaf switches, leading to contention of network links. To evaluate the impact of such over-subscription, we select two other host-pairs that do not share the same network link on Stampede. We rely on our IB network topology discovery service (Section 2.8.1) to identify suitable host-pairs for the experiment. We evaluate the communication performance for both of these configurations with the `osu_alltoall` benchmark in the OSU Microbenchmark Suite (OMB) [11] with 64 processes (four nodes, 16 processes per node). Figure 6.2(b) compares the Alltoall performance averaged across multiple runs for different message sizes across these two allocations. As we can see, the performance of the Alltoall operation degrades by up to 15% for the case where the same network link is shared across multiple transfers. This clearly identifies the need for a network topology-aware Alltoall message exchange algorithm that is capable of reducing network contention.

The amount of network bandwidth that is available to a job is another factor that determines the performance. The fraction of network bandwidth available to a job depends on the number of network links that are concurrently serving communication requests from multiple jobs. This depends entirely on how the InfiniBand Subnet Manager (eg: OpenSM) configures the switching tables. If the hosts are densely allocated, i.e. restricted to a few



(a) Example network with 5:4 over-subscription on links (b) Performance of 64 process alltoall

Figure 6.2: Impact of link over-subscription on Alltoall performance

leaf switches, then they will only have access to a lesser number of network links. Such scenarios lead to contention for network resources between processes belonging to the same job — *intra-job* contention. Such contention can be mitigated through cleverly scheduled messages, so that they do not conflict. However, if the hosts are spread across many leaf switches, then the processes in the job will have access to many more network links. Such scenarios lead to contention for resources between jobs — *inter-job* contention. Mitigating these contentions requires a global view of the system, and hence can only be achieved through scheduler-level designs. We note that addressing the challenges associated with inter-job contention is outside the scope of this work.

6.2 Design

Our design consists of two major design components: (a) scalable network path discovery using network topology discovery service and (b) topology aware Alltoall communication schedule generation. We will go into the details of each in the following Sections.

6.2.1 Enhancements to the Topology Information Interface

We use an enhanced version of the topology detection service described in Section 2.8.1 that is capable of providing the required information in a scalable and efficient fashion. We also enhance the Topology Information Interface (TII) proposed in [131] with new APIs that enable end users to access this information with ease and scalability. A brief description of the new APIs is given below:

- `tii_get_network_path_matrix`: Retrieves the communication path in the network taken by the hosts in the input file
- `tii_get_num_hosts_sharing_links`: Retrieves the number of hosts in the input host file that shares a network link just above the leaf level
- `tii_get_avg_hosts_per_leaf_switch`: Retrieves the average number of hosts per leaf switch in the allocation

6.2.2 Dynamic Communication Scheme Selection

As demonstrated in Section 6.1, whether we will see benefits by attempting to reduce the number of over-subscribed links, or not, depends on the kind of hosts allocated by the scheduler. It is possible that certain allocations are already free of intra-job contention. This could be either because the hosts are spread across many leaf switches or because the hosts do not share the same links (no over-subscription). In such cases, spending the extra time to compute a topology aware communication schedule will not lead to any significant improvement in communication performance. To handle these scenarios, we add a dynamic communication scheme selection (DCSS) component in our design that has the intelligence to select the best possible algorithm for a given allocation. DCSS gets activated if the user

(through environment variables) requested the MPI library to identify and use the best possible Alltoall algorithm for the given allocation.

Figure 6.3 depicts the flow of control in the dynamic communication schedule selection module. DCSS uses the topology information interface APIs described in Section 6.2.1 to identify all network parameters. The `tii_get_avg_hosts_per_leaf_switch` API is used first to identify if the allocation is sparse. An allocation is defined as sparse if each leaf switch in the allocation does not have more than 5% of the allocated hosts. We define this as the *sparsity-threshold*. If the allocation is sparse, then it is possible that the job will suffer more from inter-job contention rather than intra-job contention. Creating a network-topology aware communication schedule in such a context will not lead to predictable improvements in performance. Consequently, we fall to using the default algorithm. If the allocation is not sparse, then the DCSS checks if the allocation has hosts that share links using the `tii_get_num_hosts_sharing_links` API. If the allocation does not have such hosts, it means that the job will not suffer performance degradation due to link sharing and we fall back to using the default algorithm. However, if the allocation is not sparse and it has hosts that share links, then DCSS invokes the topology-aware communication schedule generation algorithm.

6.2.3 Design of Topology Aware Alltoall Communication Schedule

In this Section, we describe the various schemes to generate a topology aware communication schedule for Alltoall communication. As we observed in Section 6.1, over-subscription of links can lead to a significant degradation in communication performance for the Alltoall communication operation. Our idea is to examine the trade-offs of reducing the over-subscription of links, by reducing the number of shared links per step. The tradeoff here is between the number of over-subscribed links being used in a step and the

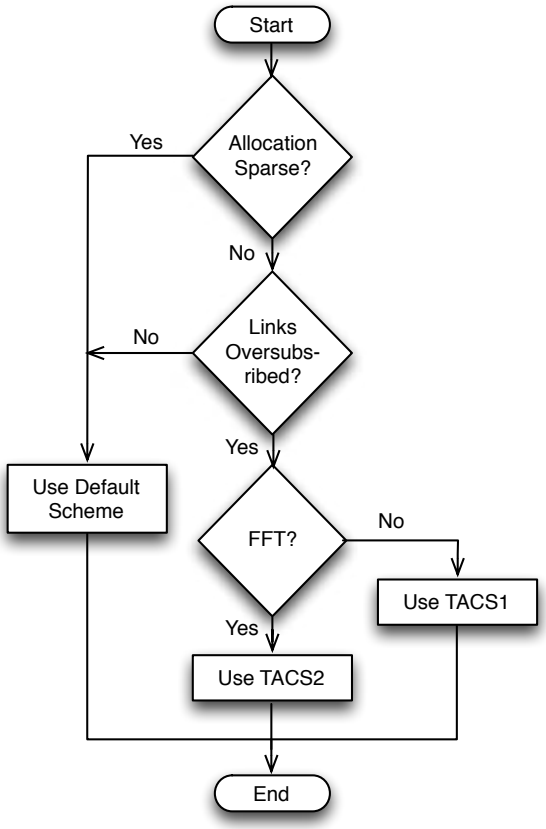


Figure 6.3: Flow of control in dynamic communication schedule selection module

total number of communication steps in the Alltoall operation. If we attempt to make the communication schedule completely free of over-subscribed links, it can require very large number of steps to finish the entire communication operation. On the other hand, if we attempt to fit the entire communication in *number_of_processes* number of steps, we may end up paying a high cost for communication due to over-subscription of links, as demonstrated in Section 6.1. The challenge is to design an algorithm where the benefits gained by making the communication steps free of over-subscribed links are not overshadowed by the degradation incurred by increasing the number of steps in the Alltoall operation.

Algorithm 2 describes the algorithm we use to generate a topology-aware communication schedule. We rely on the scalable network path discovery framework we proposed in the previous Section to get the relevant network topology information required by the algorithm. We use the proposed additions to the topology information interface to retrieve this data from the service. The algorithm is based on the principle of graph matching, described in Section 2.6. As the total exchange communication scheduling problem is *NP-Complete* [21], we use a *greedy* approximation to generate the communication schedule. As researchers have shown in the past [51], the higher levels of a Fat-Tree-like InfiniBand network is extremely susceptible to congestion. Consequently, in our scheme, each node attempts to schedule the communication with the nodes that are farthest from it (in terms of network hops), first in a manner that avoids shared links. The idea is to reduce the load on the already burdened higher levels of the Fat-Tree network. Once it has gone over all nodes, it cycles around and attempts to schedule the communication with the number of shared links that is allowed by the user. In our experiments, we found that an over-subscription ratio of *one* leads to the best tradeoff between the number of over subscribed links per step and total number of steps in the communication operation. A link is said to

Input : num_nodes, allowed_conflicts_per_edge, sparsity_threshold

Output: Topology Aware Communication Schedule

num_steps = num_nodes

$tii\_get\_network\_depth$
$tii\_get\_network\_path\_matrix$
$tii\_get\_avg\_hosts\_per\_leaf\_switch$
$tii\_get\_num\_hosts\_sharing\_links$

if *(($num\_hosts\_sharing\_links$ == 0) —— ($avg\_hosts\_per\_leaf\_switch$ ¡ $sparsity\_threshold$))* **then**
    use default algorithm
    exit
end

for $i = max\_tree\_depth \ldots 1$ **do**
    for $j = 1, 2, \ldots$ *num_steps* **do**
        foreach $node$ **do**
            identify $partner\_node$ more than i hops away that does not conflict with communication of previously selected $node\_pairs$ in step
        end
        foreach $node$ **do**
            identify $partner\_node$ more than i hops away that does not cause more than $num\_conflicts$ conflicts per edge with communication of previously selected $node\_pairs$ in step
        end
    end
end

num_conflicts = allowed_conflicts_per_edge

repeat
    add additional communication step
    foreach $node$ **do**
        identify $partner\_node$ that does not cause more than $num\_conflicts$ conflicts per edge with communication of previously selected $node\_pairs$ in step
    end
until *all exchanges are finished*;

expand node level schedule to process level granularity

Algorithm 2: Topology Aware Communication Schedule Generation

be over-subscribed by a factor of one in a step if two nodes share the link for communication in that step. If no peer could be identified for a step, then the process remains idle for that step. The algorithm progresses until it reaches the end of the normal schedule (*number_of_processes*). Any communication step that gets added after this is an extra step that has to be introduced for reducing the number of over-subscribed links per step. We call this communication scheme as **T**opology **A**ware **C**ommunication **S**chedule-1 (TACS1). The results of our initial experimental evaluation show that this scheme is able to deliver good performance for global Alltoall communication. This is because the benefits gained by making the communication steps free of over-subscribed links is greater than the degradation caused by increasing the number of steps in the communication operation. Hence we use the TACS1 communication schedule for all topology aware global Alltoall operations.

Optimization for FFT Communication: As described above, the TACS1 communication schedule is able to deliver good performance for the global Alltoall operation. However, initial evaluations of this scheme with FFT benchmarks indicate that TACS1 is not able to deliver similar improvement in performance for FFT based operations.

Typical FFT based communication happens along different row and column sub communicators consisting of a subset of processes. Consequently, this communication pattern does not generate the same level of traffic or congestion in the network as a global Alltoall where all the processes in the job are involved in the Alltoall operation. Hence, benefits gained by making the communication steps free of over subscribed links is not greater than the degradation caused by increasing the number of steps in the communication operation. To address this we come up with an optimization on top of TACS1 to ensure that the communication operation finish in *number_of_processes*. In TACS1, if a process were unable to find a peer to communicate with during a step, it would remain idle in that step. However,

in TACS2, the process will select the first process that it would normally send a message to after *number_of_processes* steps in TACS1 and schedule non-blocking sends and receives for that process. While doing this for a regular global Alltoall operation will lead to severe contention, it does not affect the performance of the FFT operation as it not as network intensive. In fact, initial evaluations indicate that the performance of FFT operations can be enhanced by up to 15% by using TACS2.

6.3 Experimental Results

In this Section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results. All numbers reported here are averages of multiple runs conducted over the course of several days. For the rest of this Section, the *default* scheme refers to the default XOR algorithm (discussed in Section 2.3.3) used to generate the communication schedule for large message MPI_Alltoall operations in the MVAPICH2 [10] MPI stack; and, the *topology-aware* scheme refers to our proposed network-topology-aware communication schedule for the MPI_Alltoall operation.

All experiments were performed on the Stampede supercomputing system at TACC [132]. Each compute node is equipped with an Intel SandyBridge series of processors using Xeon dual eight-core sockets, operating at 2.70 GHz with 32 GB RAM. Each node is equipped with MT4099 FDR ConnectX HCAs (54 Gbps data rate) with PCI-Ex Gen3 interfaces. The operating system used is CentOS release 6.3, with kernel version 2.6.32-279.el6, and OpenFabrics version 1.5.4.1. The network is a five-stage partial Fat-Tree with 5:4 over subscription on the links.

6.3.1 Overhead of Network Path Discovery

We compare the old pair-wise query scheme that required the MPI library to make $\mathcal{O}(N_{\text{hosts}}^2)$ routing queries at startup to populate the initial path matrix, with the proposed

scheme that only requires the MPI library to make $\mathcal{O}(N_{\text{hosts}})$ queries. Figure 6.4 shows the time taken in milliseconds for the older scheme and the newer scheme. As we can see, the newer scheme is far more scalable as the system size increases. As we go forward to exascale systems, such scaling performance will be extremely critical.

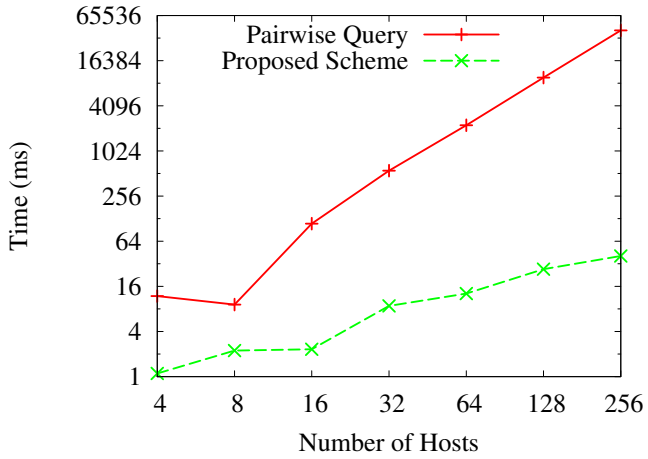


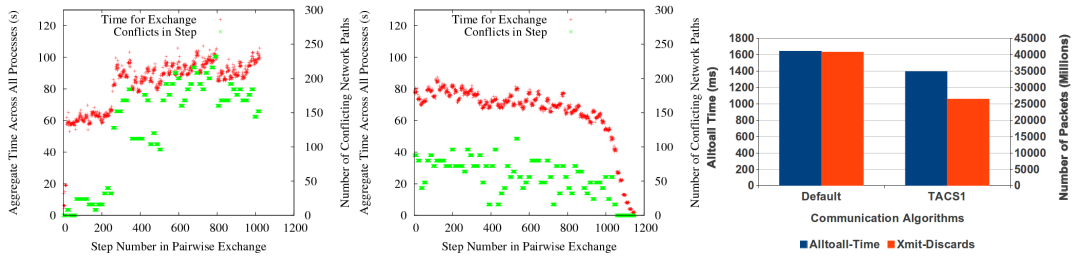
Figure 6.4: Comparison of query times incurred in old and new schemes

6.3.2 Analysis of Time Taken per Step of Global Alltoall

The Alltoall algorithm consists of several communication steps. The contention encountered by various processes in each step of the Alltoall operation has a direct impact on the completion time for that step and hence the entire Alltoall operation. As we saw in Section 6.1, over-subscription of network links is one of the major causes of contention on modern supercomputing systems. In this Section, we attempt to find a correlation between the time taken by each step of the Alltoall operation and number links that are over-subscribed in that step. We also analyze the impact of our *TACS1* communication schedule on the number of over-subscribed links and consequently the performance of the Alltoall

operation. In order to accurately measure the time taken by each step of the Alltoall operation, we insert barriers before each step and measure the amount of time spent by the MPI library in the Alltoall operation. We then aggregate this time across all processes to get an understanding of the overall time spent by the MPI library in the Alltoall operation. As described in Section 6.2, we use the network topology detection service to compute the number of over subscribed links encountered by the Alltoall operation in each step.

Figure 6.5(a) compares the time taken by the default XOR algorithm per step and the number of over-subscribed links in use in that step for a 1,024 process Alltoall operation. The overall time taken for a step is directly related to the number of over-subscribed links in use for that step. As the first few steps take place completely inside the node they take the least amount of time and do not have any over-subscribed links in them. However, as the algorithm progresses, each process talks to peers that are subsequently farther away in the network from them. Consequently, the possibility of encountering an over-subscribed link also increases towards the later parts of the algorithm, as observed in Figure 6.5(a). The proposed *TACS1* communication schedule on the other hand, as depicted in Figure 6.5(b), is able to significantly reduce the number of over-subscribed links per step and hence is able to achieve much better communication performance. In order to gain further insights into the reasons for the performance improvement, we monitored the *PortXmitDiscard* counter (described in Section 2.1.2) for signs of network congestion. Figure 6.5(c) compares the overall execution time of the Alltoall operation for the default and topology-aware cases, as well as the value of the *PortXmitDiscard* counter aggregated across all nodes in the job. As we can see, the default case has 35% larger value for the counter indicating a severe level of congestion.



(a) Default schedule (b) TACS1 schedule (c) Comparison of HCA counters and Alltoall time for the entire job

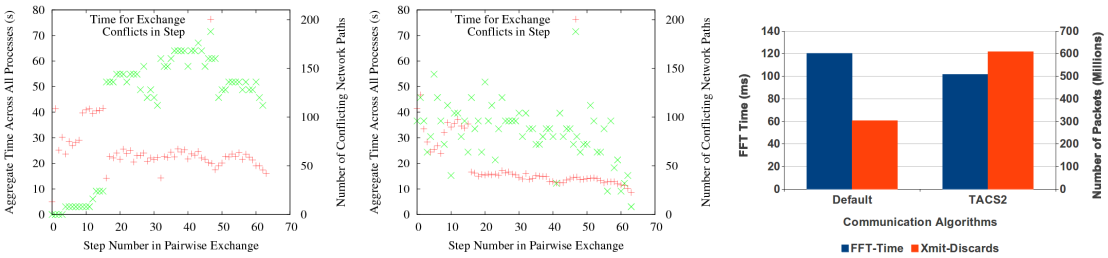
Figure 6.5: Step wise comparison of time taken and conflicts encountered for a 1,024 process MPI_Alltoall operation

6.3.3 Analysis of Time Taken per Step of FFT

In this Section, we describe our efforts to understand the impact and gain insights into the network behavior of the *TACS2* communication scheme on the performance of FFT operations using a simple 3D FFT benchmark. From the perspective of one process, the benchmark performs two Alltoalls in each iteration a row (inter-node) Alltoall that includes one process per node and spans all nodes and a column (intra-node) Alltoall that includes all processes inside a node. In our case, the column Alltoall occurs between 16 processes (the number of cores available per node on Stampede) and the row Alltoall occurs between the `num_hosts` number of processes. For 1,024 processes, the row Alltoall will contain 64 processes. As we can see, although FFT operations are implemented using Alltoall exchanges, they are significantly less communication intensive when compared to the global Alltoall exchange. As with the previous experiment, we insert barriers before each step to accurately measure the amount of time spent by the MPI library in the Alltoall operation in a step. These values are then aggregated across all processes to get a global picture of the time spent per step of the operation. Figures 6.6(a) and 6.6(b) show the time taken per step and the number of conflicts encountered per step for the default and topology-aware *TACS2*

Alltoall communication schedules for 1,024 processes. As we can see, the proposed *TACS2* communication schedule is able to uniformly reduce the time taken per step of FFT based Alltoall exchange. One interesting factor we note with both communication schedules is that the first 16 steps of the FFT seem to take more time than the others. We believe that this is due to the skew introduced by the intra-node (column) Alltoall.

In order to gain further insights into the reasons for the performance improvement, we monitored the *PortXmitDiscard* counter (described in Section 2.1.2) and evaluated network congestion. Figure 6.6(c) compares the overall execution time of the FFT operation for the default and topology-aware cases, as well as the value of the *PortXmitDiscard* counter aggregated across the entire job. As described in Section 6.2.3, the *TACS2* communication scheme trades off on network congestion for certain steps to keep a fixed upper bound on the number of steps. Hence, we expect to see an increased value for the congestion counters overall. This is mostly because in certain steps, a node can end up catering to multiple receivers. However, this does not impact the communication performance. This is because the level of congestion observed is almost 100 times less than what we saw for the Alltoall operation in Figure 6.5(c).



(a) Default schedule (b) TACS2 schedule (c) Comparison of HCA counters and FFT time for the entire job

Figure 6.6: Step wise comparison of time taken and conflicts encountered for a 1,024 process FFT operation

6.3.4 Overhead of Topology Discovery and Schedule Creation

Table 8.1 shows the *one-time* overhead encountered by the MPI library in various phases of topology discovery and schedule creation for different number of compute nodes code. For small systems sizes, the overhead caused by the network-topology-aware schedule creation is negligible. However, as the system size increases to 4,096 processes, the time taken for schedule generation also increases. We are in the process of optimizing this further by processing the various nodes at a switch level granularity instead of processing them individually. Through this optimization, we believe that we can significantly reduce the time taken for schedule generation at scale. Moreover, compared to long running times of all large applications and kernels that use Alltoalls/FFTs (P3DFFT, PSDNS, and ddcMD [129]), we believe that this one-time overhead would be negligible.

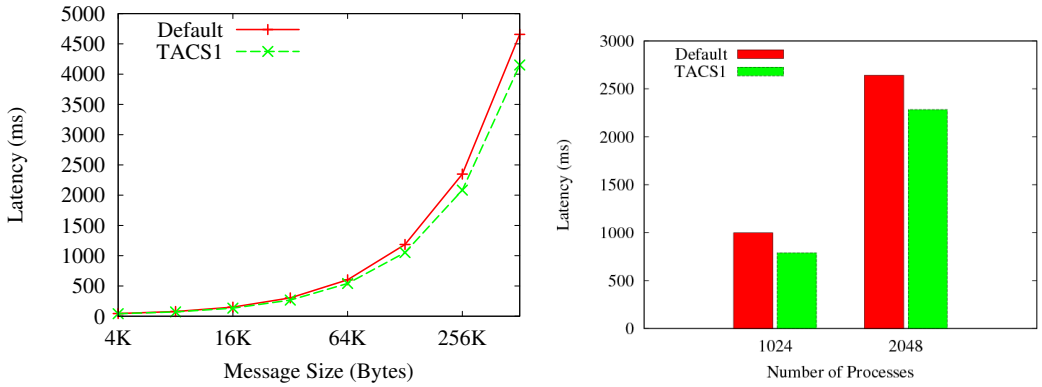
Number of Nodes (Processes)	8 (128)	16 (256)	32 (512)	64 (1,024)	128 (2,048)	256 (4,096)
Time to discover topology	0.003000	0.005180	0.004716	0.017542	0.034315	0.123710
Time for Schedule Generation	0.002023	0.002318	0.005787	0.113244	0.396456	11.6709

Table 6.1: Overhead of Topology Discovery and Schedule Creation (Time in seconds)

6.3.5 Performance with Alltoall Microbenchmark

In this Section, we look at the performance improvement obtained with the `osu_alltoall` microbenchmark (OSU Microbenchmark Suite [11]) for the global Alltoall operation using the *TACS1* communication scheme for different system sizes and message sizes. Figure 6.7(a) depicts the performance of the 1,024 process Alltoall operation for varying message sizes. As we can see, the *TACS1* communication scheme consistently outperforms the default communication scheme by up to 15%. Figure 6.7(b) shows the performance of a 128 KB Alltoall operation for varying system sizes. We observe that *TACS1*

communication scheme is able to deliver up to 15% better performance for both 1,024 and 2,048 processes.

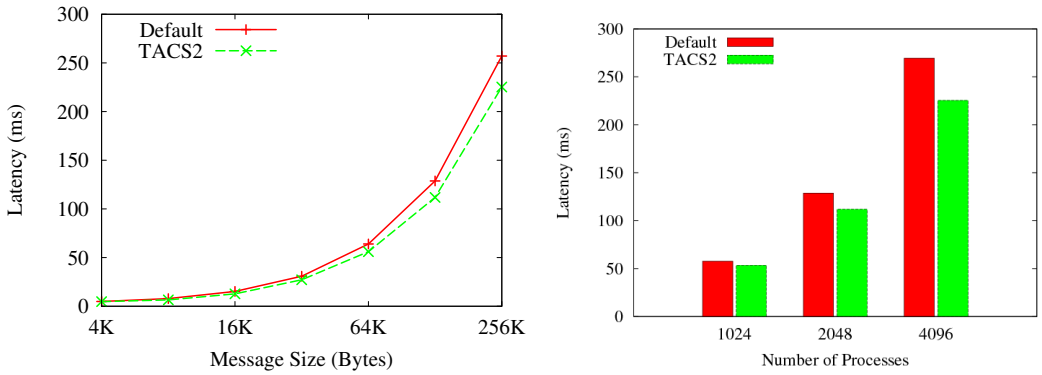


(a) Performance of 1,024 process MPI_Alltoall (b) Performance of 128 KB MPI_Alltoall

Figure 6.7: Performance with osu_alltoall microbenchmark

6.3.6 Performance with FFT Microbenchmark

We analyze the performance improvements obtained using the *TACS2* communication scheme for a simple FFT microbenchmark (described in Section 6.3.3) for various systems and message sizes. Figure 6.8(a) compares the performance of the default algorithm and the proposed *TACS2* communication schedule, for a 1,024 process run of the FFT benchmark for various message sizes. As we can see, the *TACS2* communication schedule is able to consistently outperform the default algorithm by up to 15%. In Figure 6.8(b), we compare and contrast the performance delivered by the FFT benchmark using the default and *TACS2* communication schedule for an inter-node message transfer size of 128 KB. We observe that the proposed *TACS2* communication scheme is able to consistently outperform the default version by around 15% for various system sizes.



(a) Performance of 1,024 process FFT microbenchmark

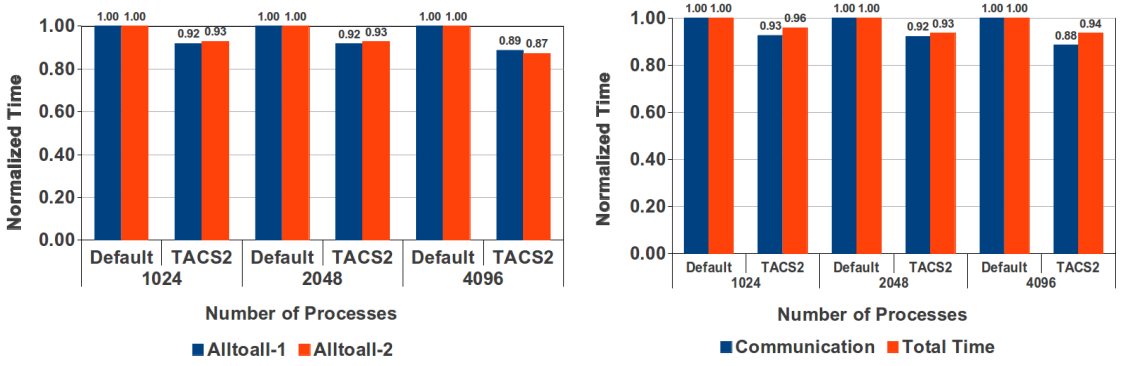
(b) Performance of 128 KB FFT microbenchmark

Figure 6.8: Performance with FFT microbenchmark

6.3.7 Performance with P3DFFT Kernel

We analyze the performance improvements obtained with the P3DFFT kernel in this Section. We used the "driver_sine" benchmark that is part of the P3DFFT installation for the evaluation. We use grid sizes of 2400x2400x2400, 3000x3000x3000, and 3600x3600x3600, respectively for 1,024, 2,048, and 4,096 processes. P3DFFT performs four Alltoalls in each iteration — two intra-node row Alltoalls and two inter-node column Alltoalls. The inter-node Alltoall dominates the total communication time. It accounts for up to 45% of total run time of the kernel. In Figure 6.9(a), we compare the performance improvements obtained using the *TACS2* communication schedule for the two inter-node Alltoalls for 1,024, 2,048, and 4,096 processes. As we can see, the *TACS2* communication scheme is able to consistently outperform the default scheme. We observe up to a 13% reduction in FFT time at 4,096 processes. Significantly the improvement increases as the system size increases. This is an encouraging trend as many applications spend a significant amount of their time in Alltoall operations at scale. Figure 6.9(b) compares the default and the *TACS2* communication scheme in terms of total execution time and total communication time. The

percentage improvement obtained for to total execution time is limited by the amount of time taken by the FFT being optimized. As the optimized FFT accounts for only 45% of the total execution time, we observe up to 6% reduction in total runtime at 4,096 processes. An important point to note is that the percentage of improvement observed increases as system size scales. We believe that at larger system sizes, the benefits see increase even further.



(a) Performance of individual FFT operations

(b) Comparison of total execution time with communication time

Figure 6.9: Performance with P3DFFT kernel

6.4 Related Work

Tuple representations of Fat-Trees for real systems (XGFTs) are proposed by Öhring [92] and extended (PGFTs) by Zahavi [143]. These representations are commonly used in discussing bandwidth requirements and contention in collective communication on InfiniBand networks. Such representations do not accurately model the sparse subtree structures that correspond to sets of nodes that are allocated to any given job on a heavily used shared system.

Hoefler et. al. demonstrates that network fabric congestion and routing-based hot-spots can have a significant impact on communication performance [50]. Zahavi confirms this result and provides additional simulation-based analysis of link contention in [143]. Our greedy algorithm attempts to find a schedule that keeps link congestion low, utilizing routing-based topology information for an allocated set of nodes. Our experiments are carried out on a large production cluster.

Many researchers have proposed routing algorithms to address fabric congestion such as random [47], adaptive [45, 83], pattern-aware [75], and D-mod-K [143, 144]. Our solution assumes a static routing method and our experimental system uses the Fat-tree Unicast routing algorithm in OpenSM 3.3.13 with OFED version 1.5.4.1.

A congestion-aware Alltoall schedule is presented by Prisacari et al. for balanced, single-rooted Fat-Trees in [107]. Additionally, the authors derive bandwidth requirements for large message Alltoall for their schedule and other common schedules. Extension for balanced XGFTs are discussed. Congestion-aware rank placement and routing for shift based collective algorithms are presented in [143]. We propose a greedy algorithm to generate schedules for arbitrary subtrees in a Fat-Tree InfiniBand network with static routing. We do not assume a specific routing algorithm, instead we utilize the OpenSM plugin infrastructure to directly query the routes assigned by the SM [131].

6.5 Summary

We proposed multiple schemes to create communication schedules for Alltoall / FFT operations that improve communication performance. Through careful study and analysis of communication performance we derived critical factors that result in network contention in large scale InfiniBand clusters. We proposed novel network-topology-aware communication schedules for the MPI_Alltoall operation. Through our techniques, we are able to

significantly reduce the amount of network contention observed during the Alltoall / FFT operations. The results of our experimental evaluation indicate that our proposed technique is able to deliver up to 12% improvement in the communication time of P3DFFT at 4,096 processes.

Chapter 7: A Scalable InfiniBand Network Topology-Aware Performance Analysis Tool for MPI

The previous chapters describe the design of network topology-aware designs for point-to-point and collective communication operations in MPI. However, as depicted in Figure 1.2, end users require network topology-aware performance analysis tools to clearly understand how an HPC application interacts with the underlying IB network and the impact network topology can have on the performance of the application. No tool currently exists that allows users of large IB clusters to analyze and to visualize the communication pattern of their MPI based HPC applications in a network topology-aware manner. We take up this challenge and design *INTAP-MPI* - a scalable, low-overhead InfiniBand Network Topology-Aware Performance Analysis Tool for MPI. INTAP-MPI allows users to analyze and visualize the communication pattern of HPC applications on any IB network (FAT Tree, Tori, or Mesh). We integrate INTAP-MPI into the MVAPICH2 MPI library, allowing users of HPC clusters to seamlessly use it for analyzing their applications. Our experimental analysis shows that the INTAP-MPI is able to profile and visualize the communication pattern of applications with very low memory and performance overhead at scale.

7.1 Design of Topology-Aware Analysis Module

The Topology-Aware Analysis Module (TAAM) forms the core of INTAP-MPI. TAAM initiates and coordinates all profiling and analysis in INTAP-MPI. The user is responsible for initiating all profiling activities. This can be done either for the entire duration of application execution (by means of environment variables) or for specific sections of the

application (by means of unix signals). TAAM performs the following activities on receiving the user request to initialize the profiling: (1) informs the Light Weight Profiling interface (LWP) (described in Section 7.2) to initiate logging the intra-node and inter-node communication inside the MPI library and, (2) queries the InfiniBand Network Topology Detection service and identifies the layout of the various processes on the InfiniBand network. Depending on the users choice (through environment variables), TAAM can either request LWP to profile either the number of messages or the volume of messages.

Once TAAM performs these activities, it remains idle until either the application terminates (calls MPI_Finalize) or receives a signal from the user requesting INTAP-MPI to finalize the profiling. On receiving the request from the user to finalize the profiling, TAAM informs the LWP to stop logging the messages. Once the LWP has stopped logging messages and handed it over to the TAAM on the local process, TAAM in *rank 0* of the MPI job gathers the communication profile from each rank. It then uses the IB network topology information obtained earlier to classify the messages logged by the LWP based on the number of hops they had to traverse. TAAM depends on the InfiniBand Network Topology Detection Service described in Section 2.8.1 to perform this classification. TAAM can perform this classification various granularities - process, compute node and switch blade, based on the users choice. This information is stored in a file which is later parsed by the data visualization module described in the next Section.

7.1.1 Data Visualization Module

As the name suggests, the Data Visualization Module (DVM) visualizes the network topology-aware communication matrix generated by TAAM using standard unix tools like *gnuplot* [4]. It generates two kinds of graphs - (1) a stacked histogram showing the splitup

of physical communication based on the number of network hops and (2) a heatmap depicting the relative volume of various types of message transfers (intra-node, inter-node-1-hop, inter-node-3-hops, inter-node-5-hops, etc). Depending on the granularity of the data generated by TAAM (process, compute node or switch blade level), the graphs generated by the DVM will also depict different patterns. This information can then be used by HPC application developers, MPI library developers as well as system administrators to decide upon the best rank / task layout for a given job. DVM also provides scripts that allows users to perform post-run comparison and analysis of the communication pattern of multiple jobs. The scripts summarize the overall communication pattern of the jobs into a single graph allowing users to reason about the performance of various jobs from a network perspective.

7.2 Design of Light Weight Profiler

The Light Weight Profiler (LWP) is responsible for logging all intra-node and inter-node communication performed by the MPI library. We integrate the LWP into the lowest communication layers of the MVAPICH2 MPI library allowing it to capture the actual communication behavior, including any fragmentation done by the MPI library for load balancing purposes. On receiving the signal from TAAM to start message logging, LWP at each process allocates an array whose size is determined by the level of profiling granularity chosen by the user. If the user wants to view the communication pattern between each pair of processes, LWP allocates an array of size $\mathcal{O}(N_{\text{processes}})$ to log the various messages issued by the process to its peers in the MPI job. If the user desires to view the communication pattern at a compute node level granularity, LWP allocates an array of size $\mathcal{O}(N_{\text{nodes}})$. LWP continues to log the messages until it receives the signal from TAAM to stop profiling. On receiving this message, it stops profiling and transfers the collected data

to the TAAM on the local process which will then be gathered by the TAAM on *rank 0* of the MPI job, as described in Section 7.1.

7.3 Experimental Results

Multiple high performance computing systems were used to obtain the results:

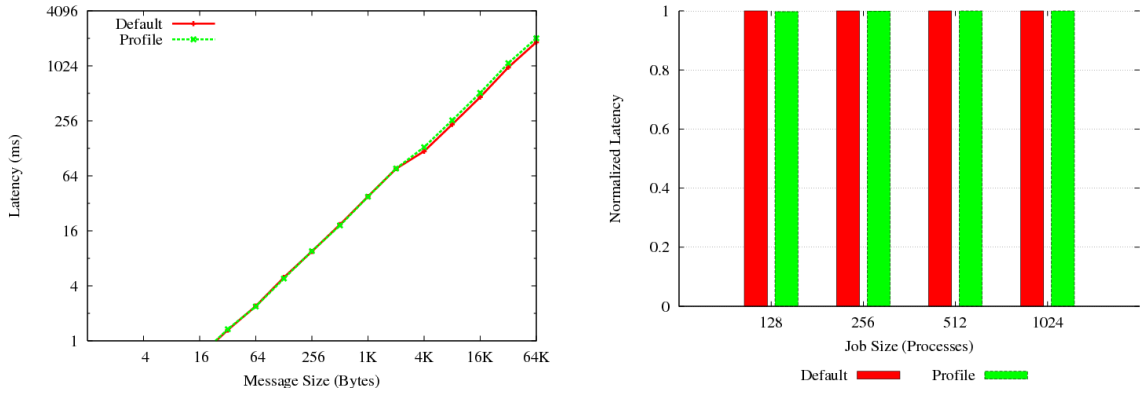
Ranger: Ranger is comprised of 3,936 16-way SMP compute nodes providing a total of 62,976 compute cores. Each core operates at 2.3 GHz and has 32 GB of memory with an independent memory controller per socket. Ranger has two 3,456 port SDR Sun InfiniBand Datacenter switches at its core. The interconnect topology is a 7-stage, full-CLOS fat tree.

Hyperion: This is a 1,400-core testbed where each node has eight Intel Xeon (E5640) cores (Nehalem) running at 2.53 Ghz with 12 MB L3 cache. Each node has 12 GB of memory and an MT26428 QDR ConnectX-2 HCA. It has a 171-port Mellanox QDR switch, with 11 leafs, each having 16 ports. The switches form a partial FAT tree. Each node is connected to the switch using one QDR link. Although Hyperion does have additional compute cores, they use the Harpertown range of CPU's. Hence we restrict ourselves to the nodes described above.

The experiments described in Section 7.3.2 were obtained on the Ranger supercomputing system. However, due to lack of system resources, we ran the larger scale jobs studying the scalability of INTAP-MPI described in Section 7.3.1 on Hyperion.

In the following visualizations of communication patterns, *Red* color represents intra-node communication, *Green* color represents 1-hop communication, *Blue* color represents 3-hop communication, *Pink* color represents 5-hop communication and *Cyan* color represents 7-hop communication. In the various heat maps to follow, the intensities of these colors represent the amount of communication between the respective entities with respect to the maximum amount of communication that any two entities in the MPI job perform.

7.3.1 Impact of INTAP-MPI on Performance of MPI Jobs



(a) Performance on 1,024 processes

(b) Performance for 64 KB message size

Figure 7.1: Impact of INTAP-MPI on performance MPI_Alltoall on Hyperion

Figure 7.1 depicts the performance of the MPI_Alltoall collective operation for various message sizes and system sizes. Figure 7.1(a) compares the communication performance of the collective operation with and without profiling at a system size of 1,024 processes. As we can see, there is very little impact on the communication performance due to INTAP-MPI. Figure 7.1(b) compares the performance of the MPI_Alltoall collective operation at various system sizes for a message size of 64 KB. As seen in Figure 7.1(a), INTAP-MPI has very little overhead on the communication performance.

7.3.2 Visualizing Network Characteristics of Collective Communication

In this Section, we analyze the performance of two different runs of MPI_Alltoall operation using INTAP-MPI. Figure 7.2(a) compares the average performance of the MPI_Alltoall operation over the two runs. Inside each run, the MPI_Alltoall operation was executed multiple times to remove possible variations in performance due to system noise. We can see

Job Size (#Processes)	64	128	256	512	1,024
Memory Overhead	0.04	0.16	0.58	2.19	8.61

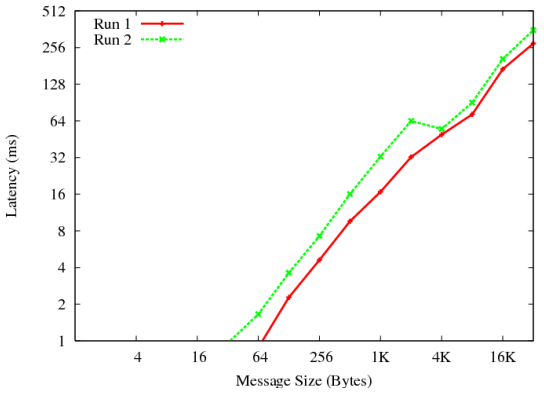
Table 7.1: Overhead (in MegaBytes) of INTAP-MPI on memory consumption of MPI_Alltoall on Hyperion

that, on average, run #1 is seen to perform better than run #2. Below, we describe how an user of INTAP-MPI can use network topology-aware communication information to reason about the difference in performance of the MPI_Alltoall operation seen between the two runs.

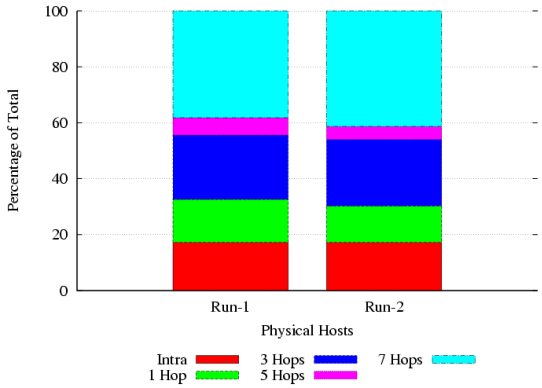
Figure 7.2(b) shows the summary of the network topology-aware communication patterns of both jobs. As we can see, the number of long distance network communication (7-hops and 5-hops) is lesser in run #1 than run #2. Figures 7.2(c) and 7.2(d) depict the network topology-aware communication pattern of the MPI_Alltoall operations as stacked histograms at node level granularity. Figures 7.2(e) and 7.2(f) illustrate the same communication pattern as heat maps at node level granularity.

7.3.3 Impact of INTAP-MPI on Memory Consumption

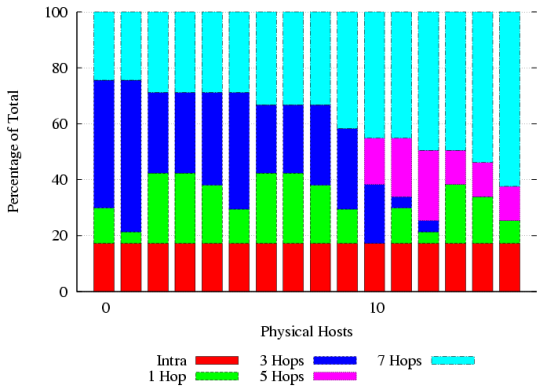
Table 7.3.3 shows the impact INTAP-MPI has on the total memory consumption of the MPI job. As we saw in Section 7.2, only *rank 0* of the MPI job needs to allocate large memory data structures like the $\mathcal{O}(N_{\text{nodes}}^2)$ matrix required to store the communication matrix of the job at various granularities (process, node, switch blade). The memory consumption at each individual rank is only of the order of $\mathcal{O}(N_{\text{nodes}})$ bytes. As we can see, the overhead imposed by INTAP-MPI on the memory consumption of the entire application is very minimal at scale. As number of nodes reaches $\mathcal{O}(10^5)$, we can start looking at switch blade level granularity to save memory.



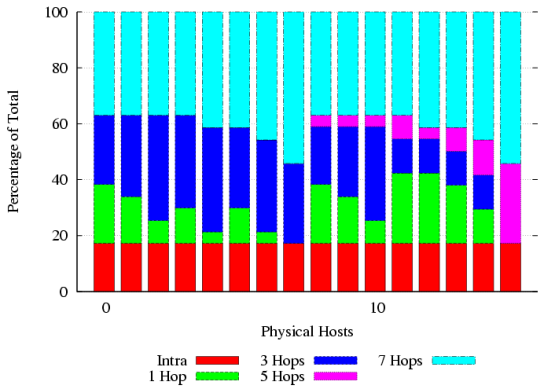
(a) Performance comparison of MPI_Alltoall between run #1 and run #2.



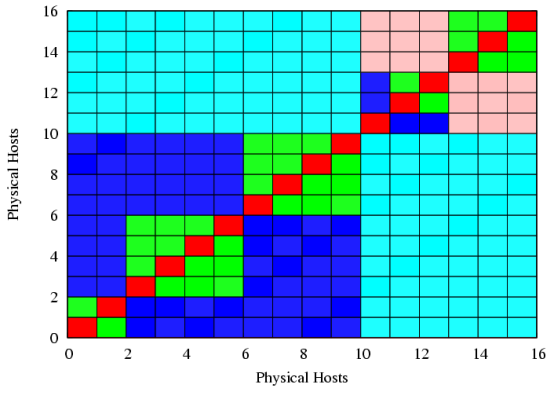
(b) Summary graph comparing communication patterns of run #1 and run #2.



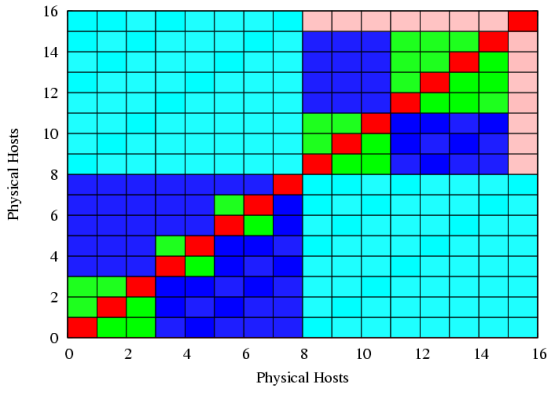
(c) Stacked histogram depicting network topology-aware communication pattern for run #1 at node level granularity.



(d) Stacked histogram depicting network topology-aware communication pattern for run #2 at node level granularity.



(e) Heatmap depicting network topology-aware communication pattern for run #1 at node level granularity.



(f) Heatmap depicting network topology-aware communication pattern for run #2 at node level granularity.

Figure 7.2: Analysis of 256 process MPI_Alltoall operation on Ranger using INTAP-MPI

7.4 Related Tools

The benefit of data visualization and performance analysis tools have been demonstrated since Prism [125]. A classical MPI profiling tool, mpiP [9], is able to display details of the performance of MPI calls through MpiPView but cannot provide any topology information. Some tools like IPM [7] or Intel Trace Analyzer and Collector (ITAC), based on Vampir [90], are able to generate the communication matrix of the messages sent and received between each task, but this information is independent of the process mapping.

For IBM Blue Gene and Cray clusters, Bhathele [22] developed a Topology Manager API allowing an easy access to the topology information. This knowledge of the topology was possible through system calls. It allowed the development of pattern language for optimizing communication [25] or heuristics [22] included inside CHARM++ [70] to provide a topology-aware process mapping. Using similar techniques, some profiling tools, like TAU [80] or the Scalasca performance toolkit [44], are able to display the topology mapping of the processes on these systems. Nevertheless, any current tool is able to profile and display the mapping information of MPI processes on InfiniBand clusters.

7.5 Summary

We presented the design, evaluation and use case driven analysis of INTAP-MPI - a network topology-aware, scalable, low-overhead MPI profiler that allows users to analyze and visualize the communication pattern of MPI based HPC applications for any IB network architecture (FAT Tree, Tori, or Mesh). INTAP-MPI gives the flexibility to profile the MPI application either for the entire duration of application execution (by means of environment variables) or for specific sections of the application (by means of Unix signals). We integrated INTAP-MPI into the MVAPICH2 MPI library, allowing users to seamlessly use it for analyzing their applications. Our experimental analysis showed that INTAP-MPI

is able to profile and visualize the communication pattern of applications with very low memory and performance overhead at scale.

Chapter 8: Design of Network Topology Aware Scheduling Services for Large InfiniBand Clusters

This thesis has so far focused on MPI level solutions for making the communication network topology-aware. However, as we identify in Section 1.1, in order for topology-aware point-to-point communication, collective communication, and task mapping to be effective in a large-scale production HEC environment, the batch scheduling system that controls the assignment of compute nodes to batch jobs must also be topology-aware. In this chapter, we propose the design of a network-topology-aware plugin for the SLURM job scheduler to make scheduling decisions and allocate compute resources in a topology-aware manner. We introduce *relaxed* network-topology-aware scheduling constraints to ensure effective utilization of network links and minimize congestion. Further, we also introduce a topology-aware task distribution plugin to map the tasks of a parallel job in a topology-aware manner by considering the communication pattern of parallel applications.

8.1 Motivation

The goal of any scheduler is to satisfy users demands for computation and achieve a good performance in overall system utilization by efficiently assigning jobs to resources. However, network congestion can occur if the various processes belonging to a job are scheduled without regard to the underlying network topology and this affects the performance of parallel applications. The interconnect fabrics of large scale supercomputing systems are typically comprised of leaf switches and many levels of spine switches. As

we saw in Table 1 described in Chapter 1, each traversal (hop) of a switch between two end points increases the message latency. Even under ideal conditions, we observe that the latencies of the 5-hop inter-node exchanges are almost 130% higher than that of 1-hop intra-rack exchanges. However, we also observe more than a factor of five increase in latency between intra-node exchange and 1-hop inter-node exchange.

Many supercomputing centers also tend to have over-subscribed interconnection networks in order to reduce the total cost of ownership. In Figure 8.1, we demonstrate a typical communication network that has a 5:4 over-subscription on the links connecting the leaf switch and the spine switch. The Stampede supercomputing system at TACC [132] has been designed with a similar level of over-subscription. We observe that such over-subscription patterns can lead to multiple hosts sharing the same network link for certain communication operations. In our example, suppose nodes A,B and 4,5 are involved in a communication operation (Figure 8.1). This communication pattern results in the same communication link servicing multiple transfers across the two leaf switches, leading to contention of network links. Moreover, researchers have also shown that network congestion becomes a significant factor as we go up the various levels of a FAT-tree network [50]. To complicate matters further, supercomputing systems typically have several hundreds of jobs running in parallel and it is common for the network to be congested, further affecting the communication latency. As the job scheduler has knowledge of both workloads and computational resources, it ideally is the responsibility of the scheduler to place jobs so they will ultimately reduce the overall contention in the network. These issues lead us to the following broad challenge - *Can we design a scheduler that can utilize dynamic topology information from the network to schedule jobs in a network-topology-aware manner?*

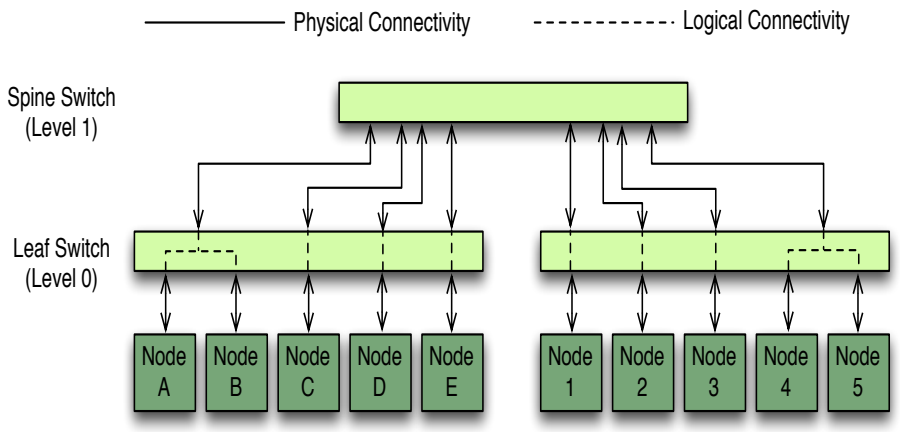


Figure 8.1: Example network with 5:4 over-subscription on links

Several researchers have shown that reducing network congestion by making communication network-topology-aware can lead to significant benefits in performance [50, 51, 113, 130, 131]. There are two ways that one can reduce network congestion through topology awareness. One is to reduce the volume of inter-node communication and the other is to make inter-node communication take separate paths through the network, i.e. make inter-node communication non-conflicting. Previously, we showed that one can significantly reduce the total volume of inter-node communication through network-topology-aware placement of processes [131]. However, as it was an MPI level solution, it was unable to fully eliminate the inter-node contention that arises due to over-subscription and link sharing that is common in large supercomputing systems. Moreover, its benefits were restricted to users of that particular MPI stack. Thus, one needs scheduler level topology awareness to be beneficial to the maximum number of users.

In order to study the effect of over-subscription of links on communication performance, we ran the osu_alltoall on a hand-picked set of nodes on the Stampede supercomputing system at TACC [132]. One set of nodes shares links with each other while the other

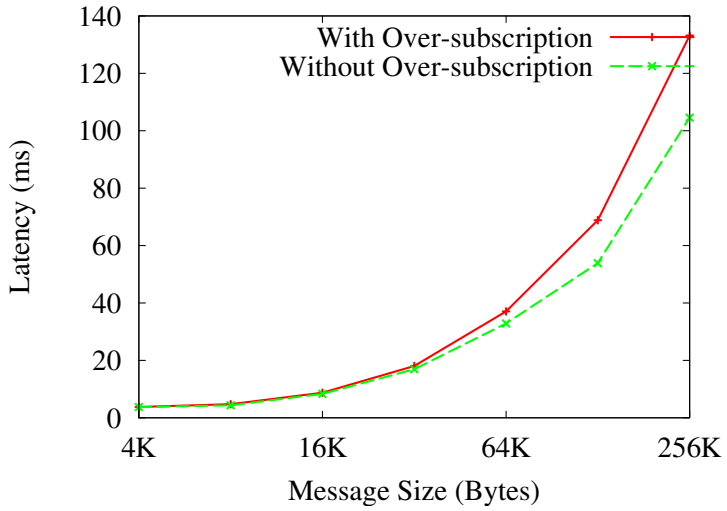


Figure 8.2: Performance of 128 process alltoall

set does not. Figure 8.2 compares the Alltoall performance averaged across multiple runs for different message sizes across these two allocations. As we can see, the performance of the Alltoall operation degrades by up to 21% when the same network link is shared across multiple transfers. While one can design communication schemes that can alleviate the level of performance degradation [107], they are all limited by the kind of nodes allocated by the scheduler. Thus, to fully eliminate the inter-node contention that arises due to link sharing, the job scheduler needs to be made network-topology-aware.

8.2 Design of Network-Topology-Aware Scheduler

In this Section, we describe the various design changes made to the SLURM job scheduler in order to support network-topology-aware selection of nodes, as well as, scheduling of processes on those nodes. Our approach is consistent with the architecture of SLURM and we introduce two new network-topology-aware plugins: (a) *OSU-Topo* topology plugin and (b) *OSU-TaskDist* task distribution plugin.

8.2.1 Description of New Command Line Parameters

Figure 8.3 shows the new command-line options we introduce to enable users to take advantage of the proposed topology-aware changes in SLURM. To enhance the performance of point-to-point based applications, we introduce a new type of task distribution - *pt2pt* to SLURM. Users are expected to provide the communication pattern of the application through a file in the metis format [73] through the option *–metis-file* along with this option. If the user specifies the *–avoid-path-conflict* option, our proposed scheduler plugin will select nodes that do not use over-subscribed links.

```
--distribution=pt2p2            distribution method for processes
                                to nodes
--metis-file=<filepath>         metis input file for pt2pt
                                distribution
--avoid-path-conflict={[0]/1}   avoid hosts that share over
                                subscribed links
```

Figure 8.3: New command line arguments

8.2.2 Design of Topology Plugin (OSU-Topo)

The SLURM scheduler plugin uses many selector plugins to choose the nodes on which to schedule a job. These selector plugins use other helper plugins to refine the node list. We intend OSU-Topo to be a helper plugin that aides in fine grained network-topology-aware refinement of the node list. Our initial experiments with the FTree plugin available in SLURM showed that the throughput of the system degrades significantly even with small number of jobs in the queue. We believe that this could be due to the selection criteria in use for this plugin. Consequently we propose the following relaxed node selection criteria for OSU-Topo to enhance the throughput of the system. In order

to efficiently support the proposed node selection criteria, we also introduce a new API - `tii_get_hosts_sharing_links` - to the Topology Information Interface (TII) proposed in [131] to enable end users to access this information with ease and scalability. This API returns the hosts in the input host file that shares a network link just above the leaf level. It is to be noted that if the user requests for a particular node or a set of nodes, then we fall back to the default node selection policy.

8.2.2.1 Proposed Relaxed Node Selection Constraints

As discussed in Section 8.1, the performance of a simple Alltoall operation degrades significantly if the nodes in question share network links. To avoid this, OSU-Topo identifies the list of nodes that share links and creates a *conflict list* when the plugin gets loaded for the first time. OSU-Topo uses the new TII API `tii_get_hosts_sharing_links` to obtain the list of nodes that shares a network link just above the leaf level. When the selector plugin calls OSU-Topo, it uses this "conflict list" to refine the list of nodes that may be used for this job. In essence, OSU-Topo attempts to ensure that no two nodes that share a link are allocated for the same job. To account for network events (switches / links going down), OSU-Topo interacts periodically with the network topology detection service to update the "conflict list".

8.2.3 Design of Task Distribution Plugin (OSU-TaskDist)

The last task performed by SLURM is that of placing the tasks on the nodes allocated by the selector plugin. We design OSU-TaskDist to schedule processes in a network-topology-aware manner on the allocated nodes. OSU-TaskDist takes two inputs: (a) the application communication pattern passed by the user through the newly introduced command line parameter described in Section 8.2.1 and (b) the list of nodes selected by OSU-Topo described in the previous Section. OSU-TaskDist then uses the TII APIs described in Section 5.2.1 to

query the relevant network information for the list of nodes selected by OSU-Topo. In particular, OSU-TaskDist uses `tii_get_network_depth`, `tii_num_switch_clusters_at_depth`, and `tii_connected_switch_cluster_at_depth`. Once OSU-TaskDist has the relevant network information, it attempts to minimize the amount of network communication between various processes by considering it as a k-way graph partitioning problem [140]. We rely on external graph partitioners to map the communication pattern of the application to the topology of the underlying network. We use both Jostle and ParMETIS for this. Most graph partitioners require the user to pass the relative weights of the various edges in the communication graph as input. If the user has either not passed this file through the new command line interface, or if it is inconsistent with the current job size, OSU-TaskDist assumes that all processes communicate equally with their peers. Next, OSU-TaskDist transforms the user input along with the network topology information, gathered through the TII, into a form that is acceptable for the interface to external graph partitioners. As we had proposed in [131], we give more importance to maximize intra-node communication.

8.3 Experimental Results

In this Section, we describe the experimental setup, provide the results of our experiments, and give an in-depth analysis of these results. All numbers reported here are averages of multiple runs conducted of the course of several days. For the rest of this Section, the *default* scheme refers to the default backfill scheduling technique used by SLURM (discussed in Section 2.7); and the *topology-aware* scheme refers to our proposed network-topology-aware scheduling technique.

Our experimental cluster comprised of 64 compute nodes with Intel Westmere series of processors using Xeon dual socket, quad-core processors (512 cores in total) operating at 2.67 GHz with 12 GB RAM. Each node is equipped with MT26428 QDR ConnectX-2 HCAs (32 Gbps data rate) with PCI-Ex Gen2 interfaces. The operating system used is Red Hat Enterprise Linux Server release 6.3 (Santiago), with kernel version 2.6.32-71.el6 and Mellanox OFED version 1.5.3-3. SLURM v2.5.6 with our proposed topology-aware changes was used as the scheduler for the experimental cluster.

8.3.1 Overhead of Topology Discovery

Table 8.1 shows the *one-time* and recurring overhead encountered by the scheduler in various phases of topology discovery and graph mapping for various number of compute nodes. As we can see, the overhead caused by the network-topology-aware designs are very small.

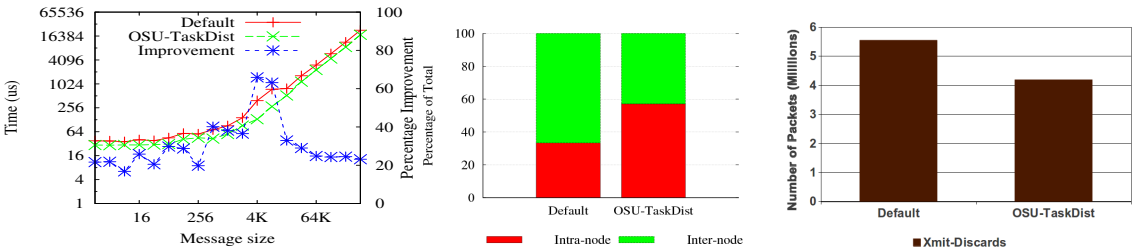
Number of Nodes (Processes)	8 (64)	16 (128)	32 (256)	64 (512)	128 (1,024)
Time to discover topology	0.0030	0.0052	0.0047	0.0175	0.0343
Time for topology aware mapping	0.0030	0.0040	0.0090	0.0200	0.0400
Total time	0.0060	0.0092	0.0137	0.0375	0.0743

Table 8.1: Overhead of Topology Discovery and Graph Mapping (Time in seconds)

8.3.2 Performance with 3D stencil benchmark

The processes in the benchmark are mapped onto a 3D grid and each process talks to its neighbors in each dimension (six neighbors). In every step, each process posts MPI_Irecv operations for all of the messages it expects and then posts all of the MPI_Isend calls. It waits for all of the transfers to complete with a single MPI_Waitall call. Figure 8.4(a) compares the performance of the 3D stencil benchmark for different messages at 512 processes.

As we can see, the topology-aware process scheduling done by *OSU-TaskDist* performs up to 40% better for all message sizes. Figure 8.4(b) depicts the summary of communication pattern in terms of number of hops for the default and topo-aware scheme, respectively. As we can clearly see, topology aware process scheduling is able to improve the intra-node communication percentage from 33% to 57%. In order to gain further insights into the reasons for the performance improvement, we monitored the *PortXmitDiscard* counter (described in Section 2.1.2) for signs of network congestion. Figure 8.4(c) compares the value of the *PortXmitDiscard* counter aggregated across all nodes in the job for the default and topology-aware runs. We observe that the default case has 25% larger value for the counter indicating a higher level of congestion.



(a) Performance comparison (b) Summary of communication pattern (c) Comparison of HCA counters

Figure 8.4: Performance analysis of a 512 process 3D stencil benchmark

8.3.3 Impact of Network Topology Aware Task Mapping on the Communication Pattern of AWP-ODC

To examine the effectiveness of our topology aware process scheduling scheme further, we use AWP-ODC. We analyze the physical communication pattern in terms of number of hops for the default and the topology-aware process scheduling scheme. Figure 8.5 summarizes the gains obtained in reducing the number of long distance communication

through topology aware process scheduling. As we can see, the topology-aware process scheduling done by *OSU-TaskDist* is able to improve the intra-node communication percentage from 33% to 57%. Although the topology aware scheme does lead to modest gains in the performance of the application, further gains are limited by an inherent imbalance in the application due to boundary condition updates. Nevertheless, this shows the potential benefit that a topology aware process scheduling can have on the communication pattern of applications. Moreover, as seen in [131], this can lead to significant improvement in performance of several applications at larger scale. Due to our limited system size, we are unable to perform large scale experiments at present. It is to be noted that one has to update the scheduler on a cluster to the proposed network-topology-aware scheduler to get such numbers. We have been able to achieve that by updating the scheduler of one cluster.

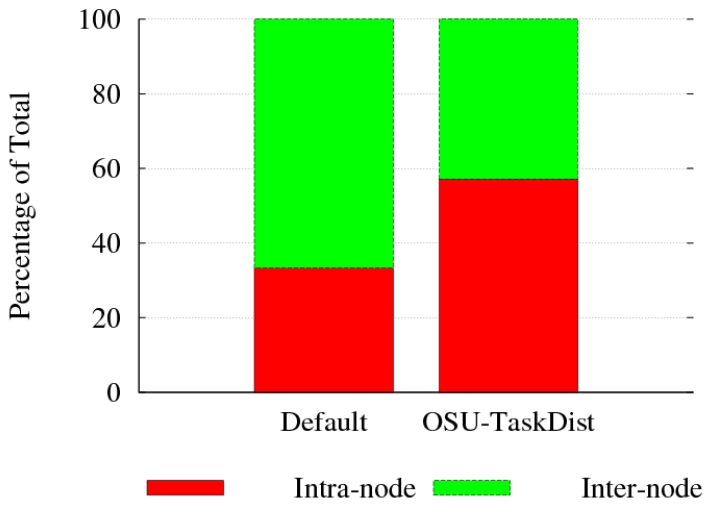
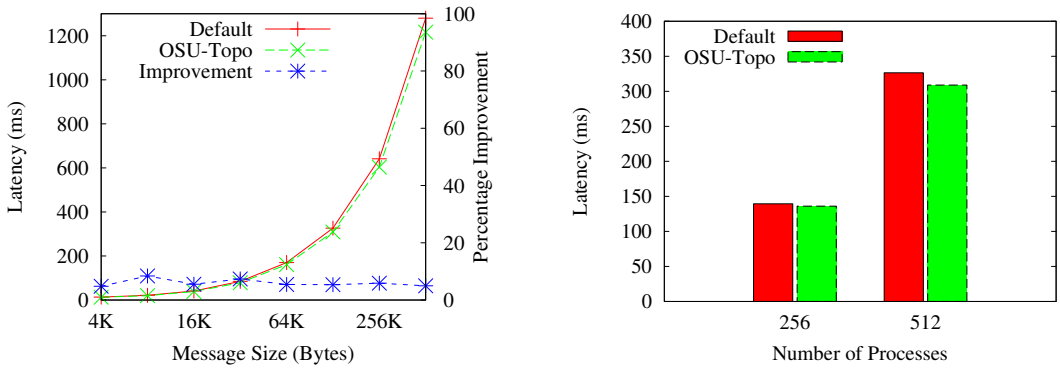


Figure 8.5: Summary of splitup of communication pattern in terms of number of hops for a 512 process run of AWP-ODC

8.3.4 Performance with Alltoallv Microbenchmark

In this Section, we look at the performance improvement obtained with the `osu_alltoallv` microbenchmark (OSU Microbenchmark Suite [11]) for the global Alltoallv operation. Figure 8.6(a) compares the performance of a non topology-aware run of the benchmark against a network-topology-aware run of the benchmark for 512 processes. We observe that the *OSU-Topo* consistently outperforms the default communication scheme by up to 6%. Figure 8.6(b) shows the performance of a 128 KB Alltoallv operation for varying system sizes. We observe that the network-topology-aware version is able to deliver up to 6% better performance for different system sizes. It is to be noted that the performance gains observed increases as the system size increases, which is a very good trend for very large clusters, such as, Stampede.



(a) Performance of 512 process MPI_Alltoallv

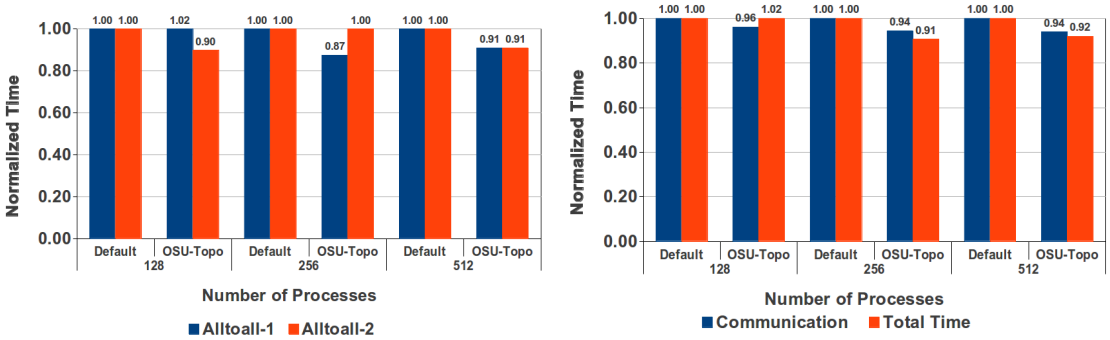
(b) Performance of 128 KB MPI_Alltoallv

Figure 8.6: Analysis of performance of osu_alltoallv microbenchmark

8.3.5 Performance with P3DFFT Kernel

We analyze the performance improvements obtained with the P3DFFT kernel in this Section. We used the "driver_sine" benchmark that is part of the P3DFFT installation for the

evaluation. We use grid sizes of 1000x1000x1000, 1400x1400x1400 and 1800x1800x1800. For each of these grid sizes, we also vary the number of processes between 128 and 512. P3DFFT performs four Alltoalls in each iteration — two intra-node row Alltoalls and two inter-node column Alltoalls. The inter-node Alltoall dominates the total communication time. It accounts for up to 45% of total run time of the kernel. In Figure 8.7(a), we compare the performance improvements obtained using the network-topology-aware scheduler for the two inter-node Alltoalls for different system sizes. As we can see, the runs with the *OSU-Topo* leads to considerably lower execution times. Further, we observe that our proposed designs improve the communication time by up to 9%. It is to be noted that the improvement in performance improves, with increase in the job size.



(a) Performance of individual FFT operations

(b) Comparison of total execution time with communication time

Figure 8.7: Performance with P3DFFT kernel

8.3.6 Impact of Network Topology Aware Task Scheduling on System Throughput

In this Section, we analyze the impact a network topology aware scheduler can have on the throughput of a high performance computing system. We define throughput as the rate at which an HPC system can execute computing jobs that have been submitted to it. In

Figure 8.8, we compare the cumulative execution time for the various jobs submitted to the experimental cluster described in Section 8.3. The size of the jobs varied from 128 to 512 processes. The jobs consisted of microbenchmarks and application kernels representing different application classes from point-to-point to dense collective operations. As we can see, the proposed network topology aware scheduler utilizing *OSU-Topo* and *OSU-TaskDist* is able to execute the same set of jobs faster when compared to the default scheduling technique used by SLURM. The cumulative improvement line in Figure 8.8 depicts this trend. We are able to observe a 8% improvement in cumulative job execution time. We believe that this number will go up as the size of the cluster increases. It is to be noted that the cumulative time indicated includes the time spent by the job waiting in the scheduler queue to get scheduled.

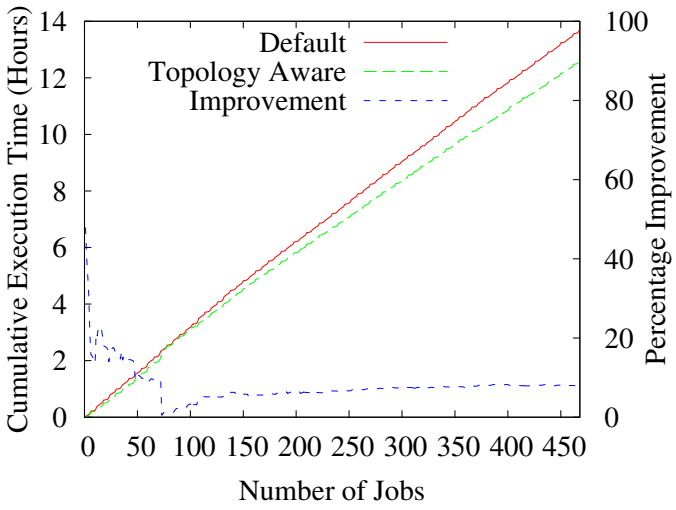


Figure 8.8: Analysis of system throughput

8.4 Related Work

Several current generation job schedulers attempt to take network topology into account while scheduling jobs on large supercomputing systems. Researchers have proposed to introduce support for topology aware scheduling [100] in the Maui [16] job scheduler. However public releases are not yet available. PBS has a range of job schedulers aimed for the supercomputing domain, some of which are capable of placing the jobs in a network topology aware manner on the system [101]. PBS builds its view of the network using basic building blocks called a virtual node or *vnode*. A virtual node is an abstract object representing a set of resources that form a usable part of a machine. The system administrator creates groups of closely connected *vnodes* called *placement sets*. The users can request for such *placement sets* while submitting jobs. While the *placement sets* provides a flexible method to define the topology, applying it across various network topologies is still a challenge that is left to the administrators.

Several researchers have studied the problem of job management on the IBM BlueGene range of systems [76]. Several solutions specifically aimed at the IBM BlueGene [55] systems are also available. Further, some of the state-of-the-art schedulers, such as, LSF [57], only consider the intra-node topology and not the network topology [126]. Researchers have also looked at the problem of effectively scheduling jobs on toroidal-interconnected systems [33]. Loadleveller [56] from IBM is another job scheduler that is capable of taking advantage of network topology information while scheduling jobs. However, like PBS, it needs to be enhanced with the ability to automatically detect multiple network topologies (FAT Trees, 3D/5D Tori, etc.). Yiannis et al., discussed the scalability of state-of-the-art job management systems on HPC cluster [46]. However, they do not take dynamic topology information into account and do not account for over-subscription of links. Previously,

we proposed a scalable network topology-aware service for InfiniBand networks [131] and demonstrated the performance improvements across different classes of parallel applications on large-scale InfiniBand clusters. We note that these designs specifically addressed the scope of topology-aware optimizations within the MPI library. Here, we address the challenges in designing the SLURM scheduler [142] to schedule jobs in a topology-aware manner, by leveraging the topology information offered by this service. We also take the level of over-subscription into account while selecting nodes.

8.5 Summary

We proposed the design of a network-topology-aware plugin for the SLURM job scheduler that uses dynamic topology information obtained through the network topology detection service to make scheduling decisions. We introduced *relaxed* network-topology-aware scheduling constraints that enhanced the performance obtained by MPI jobs launched by our proposed scheduler without affecting the total throughput of the system. Micro-benchmark level evaluations showed that the proposed network-topology-aware scheduler can improve the latency for all message sizes of a 3D-stencil benchmark by up to 40%. Through our techniques, we were able to observe up to a 9% improvement in overall communication time for P3DFFT. We also showed that our techniques have the capability to improve the throughput of a 512 core cluster by up to 8%.

Chapter 9: Future Research Directions

This chapter describes the possible future research directions that can be explored as a follow up of the work done as part of this thesis.

9.1 Study Impact of Proposed Schemes at Application Level

Applications like AWP-ODC and MPCUGLES use near neighbor or a "Halo" exchange communication pattern. For these applications, efficient ghost cell updates are necessary to ensure performance. The communication pattern of these application is also dominated by nearest neighbor type communication. These applications would also benefit from the optimizations proposed for nearest neighbor communication. Applications and kernels such as P3DFFT, PSDNS and WindJammer that rely on collective communication patterns like Alltoall and Broadcast can also benefit from the proposed optimizations for these collectives that is part of this thesis. As future work, one could evaluate the impact our proposed schemes at the MPI library and scheduler levels have on the performance of these applications on production HPC clusters such as Stampede. We have also modified benchmarks like IRS [5] and end applications such as ALE3D [2] to directly use the DGI described in Section 5.2 rather than rely on the profiling information obtained from the MPI library. One can also run these modified benchmarks and applications at scale on various supercomputing systems.

9.2 Understanding Impact of Topology-Aware communication on PGAS programming models

Topology-aware process placement as well as topology-aware point-to-point and collective communication can be relevant to PGAS programming models (like UPC and OpenSHMEM). While OpenSHMEM [95] is a library-based implementation of the PGAS model, UPC [137] presents a compiler directives based representation. Although PGAS models can offer better programmability when compared to MPI based approaches for certain applications, they still need a high performance and efficient runtime to extract the best performance for end applications on current generation distributed memory supercomputing systems. Several researchers have already studied the need for a high performance runtime implementation to extract the best possible performance for point-to-point as well as collective operations in applications written using pure PGAS as well as Hybrid MPI + PGAS (MPI + UPC, MPI + OpenSHMEM) programming models on modern high performance networks such as InfiniBand [64–66, 79, 87, 128].

The major communication operations defined in OpenSHMEM Specification v.1.0 are data-transfer routines (One-Sided Put/Get), synchronization routines (Fence, Quiet, Barrier), collective operations (Broadcast, Collection, Reduction) and Atomic memory operations (Swap, Conditional Swap, Add and Increment). In data-transfer routines (equivalent to point-to-point operations in MPI), the source/destination address of data transfer operations can either be in symmetric heap or symmetric static memory, as defined in the OpenSHMEM specification. However, as authors explain in [64, 128], although at a programming model level, each process has direct access to remote process's symmetric heap, one needs to perform network operations (One-Sided Put/Get) to bring in the data on the remote process to a memory location which is accessible by the local process. Let us

take the example of a 3D Stencil benchmark described in [131], which uses mainly point-to-point operations to perform data transfers. One way of implementing this using the OpenSHMEM programming model will be to allocate symmetric heaps to exchange the data in each dimension between the processes. However, as we saw above, such operations will get translated to One-Sided Put/Get operations by the runtime implementation. Once this happens, techniques that were applied to enhance the performance of the 3D Stencil benchmark in [131] will become directly applicable here.

Collective routines defined in Open-SHMEM specification are Broadcast, Reduction and Collection. These routines are implemented using point-to-point communication operations such as put, get, and atomic memory update operations. For example, in case of a Broadcast (linear algorithm), all the non-root processes issue a get operation and then all the processes wait on a barrier. This is similar to how collectives are implemented in MPI. However, as we saw in [130] and [71], one can significantly enhance the performance of collectives in MPI by taking network topology into account for communication. Hence, one can apply similar techniques here to enhance the performance of collectives in Open-SHMEM as well.

Applications based on the UPC programming model will also have similar performance tradeoffs. Our topology-aware techniques will be able to enhance the performance of high performance UPC runtimes similar to the ones proposed by researchers in [65]. Researchers can explore these designs and their evaluations in the future.

9.3 Understanding Impact of Topology-Aware communication on Energy Consumption

We have seen that the techniques described in Chapters 5 and 8 are able to improve the intra-node communication percentage of applications significantly. They are also able to

reduce the number of 3 and 1 hop inter-node exchanges. If, through methods similar to the ones described above, one is able to change the communication pattern of applications such that some of the switches in the network are unused, then one can selectively power down switches in the network to save power. Such designs will require tight integration with the job scheduler and the Network Manager entity. However, such coarse grained methods aimed at saving power based on network topology aware scheduling can disappear as system utilization increases. Such scenarios will require fine grained techniques to save power.

The communication performance of modern high performance interconnects such as InfiniBand depends on two factors 1) the signaling rate of the InfiniBand (IB) Host Channel Adapter (HCA) and 2) the number of physical lanes or wires used to interconnect the HCAs. Table 9.1 shows how the effective unidirectional throughput of an InfiniBand HCA varies depending on these two factors. Thus, the bandwidth of various HCAs is a product of the signaling rate and the number of physical lanes available for communication in hardware. Hence, theoretically, one can keep increasing the bandwidth of an InfiniBand HCA by adding more number of physical lanes in hardware. However, as more lanes are added, the power needed to keep these lanes operational increases as well. The InfiniBand standard [58] also offers the flexibility to selectively power up and power down physical lanes dynamically based on communication requirements.

	Single Data Rate (SDR)	Double Data Rate (DDR)	Quad Data Rate (QDR)	Fourteen Data Rate (FDR)	Enhanced Data Rate (EDR)
1 Lane	2	4	8	13.64	25
4 Lanes	8	16	32	54.54	100
12 Lanes	24	48	96	163.64	300

Table 9.1: Variance of Effective Unidirectional Throughput (in Gbps) for Different Generations of InfiniBand HCAs

As seen above, one can significantly cut down on the percentage inter-node communication through various network topology-aware techniques. Such dramatic cuts in the total percentage of inter-node communication will reduce the network bandwidth requirements placed by the applications on the InfiniBand HCAs. In this context, if one can dynamically power up and power down physical lanes in a selective fashion, it can lead to considerable savings in power without affecting the performance of end applications.

Chapter 10: Open Source Software Release and its Impact

MVAPICH2 [10], is an open-source implementation of the MPI-3.0 specification over modern high-speed networks such as InfiniBand, 10GigE/iWARP and RDMA over Converged Ethernet (RoCE). MVAPICH2 delivers best performance, scalability and fault tolerance for high-end computing systems and servers using InfiniBand, 10GigE/iWARP and RoCE networking technologies. This software is being used by more than 2,055 organizations world-wide in 70 countries and is powering some of the top supercomputing centers in the world. Examples (from the June '13 ranking) include: 6th, 462,462-core (Stampede) at Texas Advanced Computing Center (TACC), 19th, 125,980-core (Pleiades) at NASA, 21st, 73,278-core (Tsubame 2.0) at Tokyo Institute of Technology, 102nd, 16,160-core (Gordon) at UCSD/San Diego Supercomputer Center (SDSC) and 119th, 22,656-core (Lonestar) at TACC. As of June '13, more than 175,000 downloads have taken place from this project's site. This software is also being distributed by many InfiniBand, 10GigE/iWARP and RoCE vendors in their software distributions. MVAPICH2-X software package provides support for hybrid MPI+PGAS (OpenSHMEM) programming models with unified communication runtime for emerging exascale systems.

Various parts of the thesis been designed, developed and tested across various MVAPICH2 releases - 1.4 to 1.9 (current). All designs and their variants proposed as part of this thesis will be available in future MVAPICH2 releases.

Chapter 11: Conclusion and Contributions

Most of the traditional HEC applications and current petascale applications are written using the Message Passing Interface (MPI) [86] programming model. Consequently, MPI communication primitives (both point to point and collectives) are extensively used across various scientific and HEC applications. The Large-scale HEC systems on which these applications run, by necessity, are designed with multiple layers of switches with different topologies (fat-trees with different kinds of over-subscription, meshes, torus, etc.). Hence, the performance of an MPI library and in turn the applications, is heavily dependent upon how the MPI library has been designed and optimized to take the system architecture (processor, memory, network interface and network topology) into account. In addition, parallel jobs are typically submitted to such systems through schedulers (such as PBS [105] and Slurm [124]). Currently, most of these schedulers do not have intelligence to allocate MPI tasks based on the underlying topology of the system and the communication requirements of the applications. Thus, the performance and scalability of a parallel application can suffer (even using the best MPI library) if topology-aware scheduling is not done. Moreover, the placement of logical MPI ranks on a supercomputing system can significantly affect overall application performance. A naive task assignment can result in poor locality of communication. Thus, it is important to design optimal mapping schemes with topology information to improve the overall application performance and scalability. It is also critical for users of HPC installations to clearly understand the impact IB network topology can have on the performance of HPC applications. However, no tool currently exists that allows

users of such large scale clusters to analyze and to visualize the communication pattern of their MPI based HPC applications in a network topology-aware manner.

In this dissertation, we addressed several of these critical issues in the MVAPICH2 communication library. We designed an efficient and flexible topology management API that allows MPI applications, job schedulers and users to take advantage of the information provided by the topology detection service. We designed network topology aware communication schemes for multiple collective (Scatter, Gather, Broadcast, Alltoall and Alltoallv) as well as point to point operations using the proposed topology management APIs. We also designed a network topology-aware, scalable, low-overhead profiler to analyze the impact of our schemes on the communication pattern microbenchmarks and end applications. We also proposed the design of a network-topology-aware plugin for the SLURM job scheduler that uses dynamic topology information obtained through the network topology detection service to make scheduling decisions.

All designs discussed in this dissertation will be incorporated into the MVAPICH2 MPI library. Upcoming public releases will include support for all issues addressed in this dissertation. The release of network topology detection service and the topology-aware plugin for SLURM will be made in conjunction with collaborators at TACC. Scientific parallel applications can take advantage of our upcoming public releases to achieve better performance and scalability, through efficient network topology aware communication and scheduling.

Bibliography

[1] http://www.nas.nasa.gov/Resources/Systems/pleiades.html. NASA Pleiades Supercomputer.

[2] Arbitrary Lagrangian Eulerian in 3D (ALE3D). https://wci.llnl.gov/codes/ale3d/.

[3] Glenn, the ohio supercomputer center (osc). http://www.osc.edu/supercomputing/hardware/.

[4] Gnuplot. `http://www.gnuplot.info/`.

[5] Implicit Radiation Solver (IRS). https://asc.llnl.gov/sequoia/benchmarks/#irs.

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

[7] Integrated Performance Monitoring (IPM). http://ipm-hpc.sourceforge.net/.

[8] Intel multicore architecture. http://www.intel.com/technology/architecture/coremicro/.

[9] mpiP: Lightweight, Scalable MPI Profiling. http://www.llnl.gov/CASC/mpip/.

[10] MVAPICH2: High Performance MPI over InfiniBand, 10GigE/iWARP and RoCE. http://mvapich.cse.ohio-state.edu/.

[11] OSU Microbenchmarks. http://mvapich.cse.ohio-state.edu/benchmarks/.

[12] Scalable Earthquake Simulation on Petascale Supercomputers: 2010 ACM Gordon Bell Finalist. `http://sc10.supercomputing.org/schedule/event_detail.php?evid=gb118`.

[13] Schedmd. http://www.schedmd.com.

[14] A. R. Mamidala, A. Vishnu and D. K. Panda. Efficient Shared Memory and RDMA Based Design for MPI-Allgather over InfiniBand. In *13th European PVM/MPI User's Group Meeting, Bonn, Germany, September 17-20, 2006, Vol. 4192*, 2006.

[15] A. R. Mamidala and R. Kumar and D. De and D. K.Panda. MPI Collectives on Modern Multicore Clusters: Performance Optimizations and Communication Characteristics. In *8th IEEE International Symposium on Cluster Computing and the Grid, 2008, Lyon*, pages 130–137, May 2008.

[16] Adaptive Computing. Maui Scheduler. http://www.adaptivecomputing.com/resources/docs/maui/3.2environment.php.

[17] A.H. Baker, R.D. Falgout, T.V. Kolev and U.M. Yang. Scaling hypre's Multigrid Solvers to 100,000 Cores. In *High Performance Scientific Computing: Algorithms and Applications - A Tribute to Prof. Ahmed Sameh, M. Berry et al., eds., Springer, LLNL-JRNL-479591*, 2012.

[18] B. Jakimovski and M. Gusev. Improving Multilevel Approach for Optimizing Collective Communications in Computational Grids. In *Advances in Grid Computing - EGC, Lecture Notes in Computer Science, Volume 3470/2005*, pages 548–556, 2005.

[19] T. Baba, Y. Iwamoto, and T. Yoshinaga. A network-topology independent task allocation strategy for parallel computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 878–887, 1990.

[20] F. Berman and L. Snyder. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing*, 4:439–458, 1987.

[21] P.B. Bhat, V.K. Prasanna, and C.S. Raghavendra. Adaptive Communication Algorithms for Distributed Heterogeneous Systems. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, 1998.

[22] A. Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois, August 2010.

[23] A. Bhatele, E. J. Bohm, and L. V. Kalé. Optimizing communication for Charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 23(2):211–222, 2011.

[24] A. Bhatele and L. V. Kalé. An evaluative study on the effect of contention on message latencies in large supercomputers. In *IPDPS*, pages 1–8, 2009.

[25] Abhinav Bhatele, Laxmikant V Kale, Nicholas Chen, and Ralph E Johnson. A Pattern Language for Topology Aware Mapping. June 2009.

[26] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. K. Su. Myrinet: A Gigabit-per-Second Local Area Network. http://www.myricom.com.

[27] Shahid H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30:207–214, 1981.

[28] S. Wayne Bollinger and Scott F. Midkiff. Processor and link assignment in multicomputers using simulated annealing. In *ICPP (1)*, pages 1–7, 1988.

[29] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In *PDP2010*, 2010.

[30] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient Algorithms for All-to-All Communications in Multiport Message-Passing Systems. *Parallel and Distributed Systems, IEEE Transactions on*, 8(11):1143–1156, 1997.

[31] C. Coti, T. Herault, and F. Cappello. MPI Applications on Grids: A Topology Aware Approach . In *Euro-Par 2009 Parallel Processing, Lecture Notes in Computer Science, Volume 5704/2009*, pages 466–477, 2009.

[32] T. Chockalingam and S. Arunkumar. Genetic algorithm based heuristics for the mapping problem. *Computers & Operations Research*, 22:55–64, 1995.

[33] Hyunseung Choo, Seong-Moo Yoo, and Hee Yong Youn. Processor Scheduling and Allocation for 3D Torus Multicomputer Systems. *IEEE Trans. Parallel Distrib. Syst.*, 11(5):475–484, May 2000.

[34] Craig A. Lee. Topology-Aware Communication in Wide-Area Message-Passing. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, Volume 2840/2003*, pages 644–652, 2003.

[35] Y. Cui, K. B. Olsen, Y. Hu, S. Day, L. A. Dalguer, B. Minster, R. Moore, J. Zhu, P. Maechling, and T Jordan. Optimization and Scalability of a Large-scale Earthquake Simulation Application. In *American Geophysical Union, Fall Meeting*, 2006.

[36] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, and Thomas Jordan. Toward Petascale Earthquake Simulations. In *Acta Geotechnica*, pages 79–93, 2009.

[37] Yifeng Cui, Reagan Moore, Kim Olsen, Amit Chourasia, Philip Maechling, Bernard Minster, Steven Day, Yuanfang Hu, Jing Zhu, Amitava Majumdar, and Thomas Jordan. Enabling Very-Large Scale Earthquake Simulations on Parallel Machines. In *ICCS '07: Proceedings of the 7th international conference on Computational Science, Part I*, pages 46–53, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] D.A. Donis, P.K. Yeung and D. Pekurovsky. Turbulence Simulations on O(10^4) Processors. In *TeraGrid 2008*.

[39] James Dinan, Sriram Krishnamoorthy, Pavan Balaji, Jeff R. Hammond, Manojkumar Krishnan, Vinod Tipparaju, and Abhinav Vishnu. Noncollective Communicator Creation in MPI. In *EuroMPI*, 2011.

[40] Open Fabrics Enterprise Distribution. http://www.openfabrics.org/.

[41] F. Erçal, J. Ramanujam, and P. Sadayappan. Task allocation onto a hypercube by recursive mincut bipartitioning. *J. Parallel Distrib. Comput.*, 10(1):35–44, 1990.

[42] Robert D. Falgout and Ulrike Meier Yang. Hypre: A Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science-Part III*, ICCS '02, pages 632–641, London, UK, UK, 2002. Springer-Verlag.

[43] E. Frachtenberg, F. Petrini, J. Fernandez, S. Pakin, and S. Coll. STORM: Lightning-Fast Resource Management. In *Supercomputing*, 2002.

[44] M. Geimer, F. Wolf, B.J.N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurr. Comput. : Pract. Exper.*, 22(6):702–719, April 2010.

[45] Patrick Geoffray and Torsten Hoefler. Adaptive routing strategies for modern high performance networks. *High-Performance Interconnects, Symposium on*, 0:165–172, 2008.

[46] Yiannis Georgiou and Matthieu Hautreux. Evaluating Scalability and Efficiency of the Resource and Job Management System on Large HPC Clusters. In *Job Scheduling Strategies for Parallel Processing*, pages 134–156. Springer, 2013.

[47] Ronald I. Greenberg and C.E. Leiserson. Randomized routing on fat-tress. In *Foundations of Computer Science, 1985., 26th Annual Symposium on*, pages 241–249, 1985.

[48] Manish Gupta. Challenges in Developing Scalable Software for BlueGene/L. In *Scaling to New Heights Workshop*, Pittsburgh, PA, May 2002.

[49] He, Jun and Kowalkowski, Jim and Paterno, Marc and Holmgren, Don and Simone, James and Sun, Xian-He. Layout-Aware Scientific Computing: A Case Study using MILC. In *Proceedings of the Second Workshop on Latest AdScalable Algorithms for Large-Scale Systems*, ScalA '11, pages 21–24. ACM, 2011.

[50] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Proceedings of the 2008 IEEE Cluster Conference*, Sep. 2008.

[51] T. Hoefler and M. Snir. Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In *Proceedings of the 2011 ACM International Conference on Supercomputing (ICS'11)*, pages 75–85. ACM, Jun. 2011.

[52] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. The Impact of Network Noise at Large-Scale Communication Performance. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, 2009.

[53] E. Huedo, M. Prieto, I. M. Llorente, and F. Tirado. Impact of pe mapping on cray t3e message-passing performance. In *European Conference on Parallel Processing*, pages 199–207, 2000.

[54] IAA Interconnection Network Workshop. http://www.csm.ornl.gov/workshops/IAA-IC-Workshop-08/.

[55] IBM. IBM BlueGene. http://www-03.ibm.com/systems/technicalcomputing/solutions/bluegene/index.html.

[56] IBM. LoadLeveler Job Scheduler for Linux. http://www-03.ibm.com/systems/software/loadleveler/.

[57] IBM Platform Computing. LoadLeveler Job Scheduler for Linux. http://www-03.ibm.com/systems/technicalcomputing/platformcomputing/products/lsf/index.html.

[58] InfiniBand Trade Association. http://www.infinibandta.org/.

[59] Tera-scale Computing Intel Corporation. http://www.intel.com/go/terascale.

[60] J. Liu and W. Jiang and P. Wyckoff and D. K. Panda and D. Ashton and D. Buntinas and B. Gropp and B. Tooney. High Performance Implementation of MPICH2 over InfiniBand with RDMA Support. In *IPDPS*, 2004.

[61] Emmanuel Jeannot and Guillaume Mercier. Near-Optimal Placement of MPI Processes on Hierarchical NUMA Architectures. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part II*, Euro-Par'10, pages 199–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[62] Jesper Larsson Traff. Hierarchical Gather/Scatter Algorithms with Graceful Degradation. In *IPDPS, vol. 1, pp.80, 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*.

[63] Stephen C. Johnson. Hierarchical Clustering Schemes. *Psychometrika*, 32(3):241–254, September 1967.

[64] J. Jose, K. Kandalla, M. Luo, and D. K. Panda. Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation. In *Int'l Conference on Parallel Processing (ICPP '12) (Accepted for publication)*, 2012.

[65] J. Jose, M. Luo, S. Sur, and D. K. Panda. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Fourth Conference on Partitioned Global Address Space Programming Model*, 2010.

[66] J. Jose, M. Luo, S. Sur, and D. K. Panda. UPC Queues for Scalable Graph Traversals: Design and Evaluation on InfiniBand Clusters. In *Fifth Conference on Partitioned Global Address Space Programming Model (PGAS '11)*, 2011.

[67] K. B. Olsen. Simulation of Three-dimensional Wave Propagation in the Salt Lake Basin. Technical report, University of Utah, Salt Lake City, Utah, 1994.

[68] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop and D. K.Panda. Designing Multi-leader-based Allgather Algorithms for Multi-core Clusters. In *IEEE International Symposium on Parallel and Distributed Processing, IPDPS, 2009, Rome*.

[69] James A. Kahle. The Cell Processor Architecture. In *MICRO*, page 3, 2005.

[70] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[71] Krishna Kandalla, Hari Subramoni, and Dhabaleswar K. Panda. Designing Topology-Aware Collective Communication Algorithms for Large Scale InfiniBand Clusters : Case Studies wih Scatter and Gather. In *Proceedings of the 2010 IEEE International Symposium on Parallel and Distributed Processing*, 2010.

[72] Krishna Kandalla, Hari Subramoni, Gopal Santhanaraman, Matthew Koop, and Dhabaleswar K. Panda. Designing Multi-leader-based Allgather Algorithms for Multi-core clusters. In *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society.

[73] George Karypis and Vipin Kumar. METIS GRAPH Files. http://people.sc.fsu.edu/ jburkardt/data/metis_graph/metis_graph.html.

[74] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–308, 1970.

[75] Michel A. Kinsy, Myong Hyon Cho, Tina Wen, Edward Suh, Marten van Dijk, and Srinivas Devadas. Application-aware deadlock-free oblivious routing. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 208–219, New York, NY, USA, 2009. ACM.

[76] Elie Krevat, José G. Castaños, and José E. Moreira. Job Scheduling for the BlueGene/L System. In *Revised Papers from the 8th International Workshop on Job*

Scheduling Strategies for Parallel Processing, JSSPP '02, pages 38–54. Springer-Verlag, 2002.

[77] J. Lawrence and X. Yuan. An MPI Tool for Automatically Discovering the Switch Level Topologies of Ethernet Clusters. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008, Miami, FL*, pages 1–8, April, 2008.

[78] S.-Y. Lee and J. K. Aggarwal. A Mapping Strategy for Parallel Processing. *IEEE Trans. Comput.*, 36(4):433–442, April 1987.

[79] M. Luo, J. Jose, S. Sur, and D. K. Panda. Multi-threaded UPC Runtime with Network Endpoints: Design Alternatives and Evaluation on Multi-core Architectures. In *Int'l Conference on High Performance Computing (HiPC '11)*, Dec. 2011.

[80] Allen D. Malony and Sameer Shende. Performance Technology for Complex Parallel and Distributed Systems. In *Proc. DAPSYS 2000, G. Kotsis and P. Kacsuk (Eds)*, pages 37–46, 2000.

[81] N. Mansour and G. Fox. Allocating data to multicomputer nodes by physical optimization algorithms for loosely synchronous computations. *Concurrency - Practice and Experience*, 4(7):557–574, 1992.

[82] N. Mansour, R. Ponnusamy, A. Choudhary, and G. C. Fox. Graph contraction for physical optimization methods: a quality-cost tradeo for mapping data on parallel computers. In *7th International Conference on Supercomputing*, pages 1–10, 1993.

[83] J.C. Martinez, J. Flich, A. Robles, P. Lopez, and J. Duato. Supporting fully adaptive routing in infiniband networks. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 10 pp.–, 2003.

[84] Robert McLay. Windjammer. http://www.tacc.utexas.edu/tacc-projects.

[85] Guillaume Mercier and Emmanuel Jeannot. Improving MPI Applications Performance on Multicore Clusters with Rank Reordering. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 39–49, Berlin, Heidelberg, 2011. Springer-Verlag.

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

[87] Miao Luo and Hao Wang and Dhabaleswar K. Panda. Multi-Threaded UPC Runtime for GPU to GPU communication over InfiniBand. In *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[88] Müller, Matthias S. and van Waveren, Matthijs and Lieberman, Ron and Whitney, Brian and Saito, Hideki and Kumaran, Kalyan and Baron, John and Brantley, William C. and Parrott, Chris and Elken, Tom and Feng, Huiyu and Ponder, Carl. SPEC MPI2007–An Application Benchmark Suite for Parallel Systems using MPI. *Concurr. Comput. : Pract. Exper.*, 22(2):191–205, February 2010.

[89] N. T. Karonis, B. R. de Supinski, I. T. Foster, W. Gropp and E. L. Lusk. A Multilevel Approach to Topology-Aware Collective Operations in Computational Grids. *CoRR*, cs.DC/0206038, 2002.

[90] W. E. Nagel, A. Arnold, M. Weber, H.-Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12:69–80, 1996.

[91] Wayne Pfeiffer Nicholas J. Wright and Allan Snavely. Characterizing Parallel Scaling of Scientific Applications using IPM. In *10th LCI Conference*, March 2009.

[92] Sabine R. Öhring, Maximilian Ibel, Sajal K. Das, and Mohan J. Kumar. On generalized fat trees. In *Proceedings of the 9th International Symposium on Parallel Processing*, IPPS '95, pages 37–, Washington, DC, USA, 1995. IEEE Computer Society.

[93] K.B. Olsen, S.M. Day, J.B. Minster, Y. Cui, A. Chourasia, M. Faerman, R. Moore, P. Maechling, and T Jordan. Strong Shaking in Los Angeles Expected from Southern San Andreas Earthquake. In *Geophysical Research Letters, Vol 3*, pages 1–4, 2006.

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

[95] OpenSHMEM. OpenSHMEM Application Programming Interface. http://www.openshmem.org.

[96] P. Bar, C. Coti, D. Groen, T. Herault, V. Kravtsov, A. Schuster and M. Swain. Running Parallel Applications with Topology-Aware Grid Middleware. In *5th IEEE International Conference on e-Science (eScience 2009), Oxford, UK*, 2009.

[97] C. H. Papadimitriou and K. Steiglitz. *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[98] Parallel Three-Dimensional Fast Fourier Transforms (P3DFFT) library, San Diego Supercomputer Center (SDSC).

[99] P. Patarasuk and X. Yuan. Bandwidth Efficient All-reduce Operation on Tree Topologies. In *IEEE International Symposium on Parallel and Distributed Processing, 2007. IPDPS 2007, Long Beach, CA*, pages 1–8, 2007.

[100] V.A. Patil and V. Chaudhary. Rack Aware Scheduling in HPC Data Centers: An Energy Conservation Strategy. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 814 – 821, may 2011.

[101] PBS Works. PBS Professional 11.3 - Administrator's Guide. http://www.pbsworks.com/documentation/support/PBSProAdminGuide11.3.pdf.

[102] Dmitry Pekurovsky. P3dfft library, 2006–2009.

[103] François Pellegrini and Jean Roman. Scotch: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *High-Performance Computing and Networking*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1996.

[104] F. Petrini, W. C. Feng, A. Hoisie, S. Coll, and E. Frachtenberg. The Quadrics Network (QsNet): High-Performance Clustering Technology. In *Hot Interconnects*, 2001.

[105] Portable Batch System (PBS). http://www.pbsgridworks.com/.

[106] S. Potluri, P. Lai, K. Tomko, S. Sur, Y. Cui, M. Tatineni, K. Schulz, W. Barth, A. Majumdar, and D. K. Panda. Quantifying Performance Benefits of Overlap using MPI-2 in a Seismic Modeling Application. In *International Conference on Supercomputing (ICS'10)*, 2010.

[107] Bogdan Prisacari, German Rodriguez, Cyriel Minkenberg, and Torsten Hoefler. Bandwidth-optimal All-to-all Exchanges in Fat Tree Networks. In *27th International Conference on Supercomputing*, 2013.

[108] Intel pushes for 80-core CPU by 2010. http://www.vnunet.com/vnunet/news/2165072/intel-unveils-tera-scale.

[109] R. Graham and G. Shipman. MPI Support for Multi-core Architectures: Optimized Shared Memory Collectives. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science*, volume Volume 5205/2008, pages 130–140, 2008.

[110] R. Kumar, A. R. Mamidala and D. K. Panda. Scaling Alltoall Collective on Multi-core Systems. In *IEEE International Symposium on Parallel and Distributed Processing, 2008, IPDPS, 2008, Miami, FL*, 2008.

[111] R. Thakur and W. Gropp. Improving the Performance of Collective Operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, Lecture Notes in Computer Science, 2003, Vol. 2840/2003*, pages 257–267, 2003.

[112] S. Radhakrishnan, R. Brunner, and L. V. Kalé. Branch and bound based load balancing for parallel applications. In *Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments*, ISCOPE '99, pages 194–199, London, UK, 1999. Springer-Verlag.

[113] Mohammad Javad Rashti, Jonathan Green, Pavan Balaji, Ahmad Afsahi, and William Gropp. Multi-core and Network Aware MPI Topology Functions. In *Proceedings of the 18th European MPI Users' Group conference on Recent advances in the message passing interface*, EuroMPI'11, pages 50–60, Berlin, Heidelberg, 2011. Springer-Verlag.

[114] S. Laizet, E. Lamballais and J.C. Vassilicos. A Numerical Strategy to Combine High-Order Schemes, Complex Geometry and Parallel Computing for High Resolution DNS of Fractal Generated Turbulence. In *Computers&Fluids*, volume 39, pages 471–484, 2010.

[115] S. Wayne Bollinger and Scott F. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans. Comput.*, 40:325–333, March 1991.

[116] Paul Sack and William Gropp. A Scalable MPI_Comm_split Algorithm for Exascale Computing. In *Recent Advances in the Message Passing Interface*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010.

[117] Naruya Saitou and Masatoshi Nei. The neighbor-joining method: a new method for reconstructing phylogentic trees. *Mol. Biol. Evol*, 4:406–425, 1987.

[118] Naruya Saitou and Masatoshi Nei. The Neighbor-Joining Method: A New Method for Reconstructing Phylogentic Trees. *Mol. Biol. Evol*, 4:406–425, 1987.

[119] Sameer Kumar, Yogish Sabharwal, Rahul Garg,Philip Heidelberger. Optimization of All-to-All Communication on the Blue Gene/L Supercomputer . In *Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 320–329, 2008.

[120] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel Static and Dynamic Multi-Constraint Graph Partitioning. *Concurrency and Computation: Practice and Experience*, pages 219–240, 2002.

[121] S. L. Scott and G. Thorson. The cray t3e network: adaptive routing in a high performance 3d torus. In *Proceedings of Hot Interconnects Symposium IV*, August 1996.

[122] S. L. Scott and G. M. Thorson. The Cray T3E Network: Adaptive Routing in a High Performance 3D Torus. In *Proceedings of the Symposium on High Performance Interconnects (Hot Interconnects 4)*, pages 147–156, August 1996.

[123] SiCortex Inc. http://www.sicortex.com.

[124] Simple Linux Utility for Resource Management (SLURM). http://www.llnl.gov/linux/slurm/.

[125] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. A Scalable Debugger for Massively Parallel Message-Passing Programs. *IEEE Parallel Distrib. Technol.*, 2(2):50–56, June 1994.

[126] C. Smith, B. McMillan, and I. Lumb. Topology Aware Scheduling in the LSF Distributed Resource Manager.

[127] Cloyce D. Spradling. SPEC CPU2006 Benchmark Tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, March 2007.

[128] Sreeram Potluri and Krishna Kandalla and Devendar Bureddy and Minzhe Li and Dhabaleswar K. Panda. Efficient Internode Designs for OpenSHMEM on Multicore Clusters. In *Proceedings of the 6th Conference on Partitioned Global Address Space Programming Models (PGAS'12)*, 2012.

[129] Frederick H. Streitz, James N. Glosli, Mehul V. Patel, Bor Chan, Robert K. Yates, and Bronis R. de Supinski. 100+ TFlop Solidification Simulations on BlueGene/L. In *SuperComputing*, 2005.

[130] H. Subramoni, K. Kandalla, J. Vienne, S. Sur, B. Barth, K. Tomko, R. McLay, K. Schulz, and D. K. Panda. Design and Evaluation of Network Topology-/Speed-Aware Broadcast Algorithms for InfiniBand Clusters. In *IEEE Cluster '11*, 2011.

[131] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A.Moody, and D. K. Panda. Design of a Scalable InfiniBand Topology Service to Enable Network-Topology-Aware Placement of Processes. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (Accepted for publication)*, 2012.

[132] Stampede Cluster Texas Advanced Computing Center. http://www.tacc.utexas.edu/stampede.

[133] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. In *Int'l Journal of High Performance Computing Applications*, pages (19)1:49–66, 2005.

[134] The MIMD Lattice Computation (MILC) Collaboration. `http://physics.indiana.edu/~sg/milc.html`.

[135] The NERSC-6 Benchmarks. http://www.nersc.gov/research-and-development/benchmarking-and-workload-characterization/nersc-6-benchmarks/.

[136] The NERSC SDSA Benchmark Codes. http://www1.nersc.gov/projects/SDSA/software/.

[137] UPC Consortium. UPC Language Specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[138] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient Asynchronous Memory Copy Operations on Multi-Core Systems and I/OAT. In *International Conference on Cluster Computing*, 2007.

[139] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *In Proceedings of Workshop on Communication Architecture for Clusters (CAC); In Conjunction with the International Parallel and Distributed Processing Symposium*, 2007.

[140] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*. Civil-Comp Ltd., 2007.

[141] Travis J. Wheeler. Large-scale neighbor-joining with ninja. In S.L. Salzberg and T. Warnow, editors, *Proceedings of the 9th Workshop on Algorithms in Bioinformatics*, pages 375–389. WABI, Springer, Berlin, 2009.

[142] Andy B. Yoo, Morris A. Jette, and Mark Grondona. SLURM: Simple Linux Utility for Resource Management. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *JSSPP*, volume 2862 of *Lecture Notes in Computer Science*, pages 44–60. Springer, 2003.

[143] Eitan Zahavi. Fat-Tree Routing and Node Ordering Providing Contention Free Traffic for MPI Global Collectives. *Journal of Parallel and Distributed Computing*, 72(11):1423 – 1432, 2012.

[144] Eitan Zahavi, Gregory Johnson, Darren J. Kerbyson, and Michael Lang. Optimized InfiniBandTM Fat-Tree Routing for Shift All-to-all Communication Patterns. *Concurr. Comput. : Pract. Exper.*, 22(2):217–231, February 2010.