Programmable Interconnect Control Adaptive to Communication Pattern of Applications

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2019

Keichi TAKAHASHI

This work is dedicated to my parents and my wife

List of Publications by the Author

Journal

[1] Keichi Takahashi, S. Date, D. Khureltulga, Y. Kido, H. Yamanaka, E. Kawai, and S. Shimojo, "UnisonFlow: A Software-Defined Coordination Mechanism for Message-Passing Communication and Computation", *IEEE Access*, vol. 6, no. 1, pp. 23 372–23 382, 2018. DOI: 10.1109/ACCESS.2018.2829532.

[2] A. Misawa, S. Date, Keichi Takahashi, T. Yoshikawa, M. Takahashi, M. Kan, Y. Watashiba, Y. Kido, C. Lee, and S. Shimojo, "Dynamic Reconfiguration of Computer Platforms at the Hardware Device Level for High Performance Computing Infrastructure as a Service", *Cloud Computing and Service Science. CLOSER 2017. Communications in Computer and Information Science*, vol. 864, pp. 177–199, 2018. DOI: 10.1007/978-3-319-94959-8_10.

[3] S. Date, H. Abe, D. Khureltulga, Keichi Takahashi, Y. Kido, Y. Watashiba, P. Uchupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, "SDN-accelerated HPC Infrastructure for Scientific Research", *International Journal of Information Technology*, vol. 22, no. 1, pp. 789–796, 2016.

International Conference (with review)

[1] Y. Takigawa, Keichi Takahashi, S. Date, Y. Kido, and S. Shimojo, "A Traffic Simulator with Intra-node Parallelism for Designing High-performance Interconnects", in *2018 International Conference on High Performance Computing & Simulation (HPCS 2018)*, Jul. 2018, pp. 445–451. DOI: 10.1109/HPCS.2018.00077.

[2] Keichi Takahashi, S. Date, D. Khureltulga, Y. Kido, and S. Shimojo, "PFAnalyzer: A Toolset for Analyzing Application-Aware Dynamic Interconnects", in *2017 International Conference on Cluster Computing (CLUSTER 2017)*, Sep. 2017, pp. 789–796. DOI: 10.1109/CLUSTER.2017.18.

[3] A. Misawa, S. Date, Keichi Takahashi, T. Yoshikawa, M. Takahashi, M. Kan, Y. Watashiba, Y. Kido, C. Lee, and S. Shimojo, "Highly Reconfigurable Computing Platform for High Performance Computing Infrastructure as a Service: Hi-IaaS", in *7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*, Apr. 2017, pp. 163–174. DOI: 10.5220/0006302501630174.

[4] H. Morimoto, K. Dashdavaa, Keichi Takahashi, Y. Kido, S. Date, and S. Shimojo, "Design and Implementation of SDN-enhanced MPI Broadcast Targeting a Fat-Tree Interconnect", in *2017 International Conference on High Performance Computing & Simulation (HPCS 2017)*, Jul. 2017, pp. 252–258. DOI: 10.1109/HPCS.2017.46.

[5] T. Yamada, Keichi Takahashi, M. Muraki, S. Date, and S. Shimojo, "Network Access Control Towards Fully-controlled Cloud Infrastructure", in *Ph.D. Consortium, 8th International Conference on Cloud Computing Technology and Science (CloudCom2016)*, Dec. 2016, pp. 452–455. DOI: 10.1109/CloudCom.2016.0076.

[6] S. Date, H. Abe, K. Dashdavaa, Keichi Takahashi, Y. Kido, Y. Watashiba, P. U-Chupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, "An Empirical Study of SDN-accelerated HPC Infrastructure for Scientific Research", in *International Conference on Cloud Computing Research and Innovation (ICCCRI)*, Oct. 2015. DOI: 10.1109/ICCCRI.2015.13.

[7] Keichi Takahashi, D. Khureltulga, B. Munkhdorj, Y. Kido, S. Date, H. Yamanaka, E. Kawai, and S. Shimojo, "Concept and Design of SDN-Enhanced MPI Framework", in *2015 European Workshop on Software Defined Networks (EWSDN 2015)*, Sep. 2015, pp. 109–110. DOI: 10.1109/EWSDN.2015.72.

[8] P. Makpaisit, K. Ichikawa, P. Uthayopas, S. Date, Keichi Takahashi, and K. Dashdavaa, "An Efficient MPI_Reduce Algorithm for OpenFlow-Enabled Network", in *15th International Symposium on Communications and Information Technologies (ISCIT'15)*, Oct. 2015, pp. 261–264. DOI: 10.1109/ISCIT.2015.7458357.

[9] B. Munkhdorj, Keichi Takahashi, K. Dashdavaa, Y. Watashiba, Y. Kido, S. Date, and S. Shimojo, "Design and Implementation of Control Sequence Generator for SDN-enhanced MPI", in *5th International Workshop on Network-aware Data Management (NDM'15)*, Nov. 2015. DOI: 10.1145/2832099.2832103.

[10] Keichi Takahashi, D. Khureltulga, Y. Watashiba, Y. Kido, S. Date, and S. Shimojo, "Performance Evaluation of SDN-enhanced MPI_Allreduce on a Cluster System with Fat-tree Interconnect", in *2014 International Conference on High Performance Computing & Simulation (HPCS 2014)*, Jul. 2014, pp. 784–792. DOI: 10.1109/HPCSim.2014.6903768.

International Conference (without review)

[1] Keichi Takahashi, "An MPI Framework for HPC Clusters Deployed with Software-Defined Networking", in *27th Workshop on Sustained Simulation Performance (WSSP27)*, Mar. 2018.

[2] Keichi Takahashi, "Towards Realizing a Dynamic and MPI Application-aware Interconnect with SDN", in *26th Workshop on Sustained Simulation Performance (WSSP26)*, Oct. 2017.

[3] T. Yamada, Keichi Takahashi, M. Muraki, Y. Kido, S. Date, and S. Shimojo, "A Proposal of Access Control Mechanism for the IoT world", in *ICT Virtual Organization of ASEAN Institutes and NICT (ASEAN IVO) Meeting*, Sep. 2016.

[4] T. Yamada, Keichi Takahashi, M. Muraki, Y. Kido, S. Date, and S. Shimojo, "A Proposal of Access Control Mechanism Towards User-dedicated PRAGMA-ENT for IoT Era", in *The Pacific Rim Application and Grid Middleware Assembly (PRAGMA) Workshop 31*, Sep. 2016.

[5] K. Dashdavaa, M. Baatarsuren, Keichi Takahashi, S. Date, Y. Kido, and S. Shimojo, "A MPI Concept with Efficient Control of Network Functionality Based on SDN", in *The Pacific Rim Application and Grid Middleware Assembly (PRAGMA) Workshop 29*, Oct. 2015.

[6] Keichi Takahashi, B. Munkhdorj, K. Dashdavaa, S. Date, Y. Kido, and S. Shimojo, "Control Sequence Generator for Generic SDN-enhanced MPI Framework", in *The Pacific Rim Application and Grid Middleware Assembly (PRAGMA) Workshop 28*, Apr. 2015.

Summary

Recent breakthroughs in science have significantly benefited from high performance computing (HPC). The majority of HPC systems today adopt cluster architecture to achieve their massive scalable computing performance. A cluster is a system of computers connected through a high-performance network referred to as an interconnect. In a cluster, processes running on different compute nodes work collectively by exchanging data and messages with one another over the interconnect. Therefore, the communication performance of the interconnect is critically important for the total computing performance of a cluster.

The inter-process communication of an application running on a cluster system shows a distinctive pattern that originates from the mathematical model, discretization method and parallelization strategy used in the application. The design of an interconnect could be highly optimized for a representative application expected on the target system by taking the communication pattern of the application into account. However, this design approach is infeasible and unrealistic when designing a real-world cluster, since many users share a single HPC system and each user runs various applications. Therefore, in contrast to the application-dependent communication pattern, the interconnect is inherently designed in an application-independent manner, assuming a uniform communication pattern between processes. As a result, an imbalance in the packet flow in the interconnect can occur under a non-uniform communication pattern. This imbalance can lead to traffic congestion on a link in the interconnect, which lowers the throughput of communication and degrades the total application performance as a result.

This dissertation tackles this degradation of application performance by adapting the control of packet flow in the interconnect to the communication pattern of applications. Until recently, such dynamic adaptation of the interconnect control has been deemed infeasible due to the lack of a networking architecture, technology, or technique that allows flexible and dynamic reconfiguration. However, the recent emergence of programmable

networking architectures exemplified by Software-Defined Networking (SDN) has opened up the possibility to realize such adaptation. This dissertation leverages programmable network architectures to overcome the shortcoming of conventional application-independent interconnects described in the last paragraph. In particular, this dissertation aims at establishing a programmable interconnect control that dynamically manages the packet flow in the interconnect based on the communication pattern of applications.

This dissertation tackles the following three challenges to achieve the goal described above: (1) analyzing the packet flow in the interconnect, (2) dynamic adaptation of the interconnect control, and (3) coordinating the execution of application and interconnect control. The first challenge is required to observe and understand the imbalance of packet flow in the interconnect to perform an effective adaptation of the interconnect control. The second challenge is required to mitigate the imbalance of packet flow and improve the performance of inter-process communication. The third challenge is required since many real-world applications exhibit time-varying communication patterns and therefore the interconnect control needs to be performed in accordance with the execution of an application.

To address the first challenge, Chapter 2 proposes PFAnalyzer, a toolset for analyzing the packet flow in the interconnect. When designing and implementing an efficient programmable interconnect control, researchers need to conduct a systematic analysis over many combinations of applications and interconnects. Since performing such an analysis on a physical cluster is time-consuming, this research utilizes simulation to facilitate the analysis. The proposed toolset is a pair of tools: an interconnect simulator specialized for programmable interconnects, and a profiler to collect communication pattern from applications. PFSim allows researchers and designers working on interconnects to investigate possible congestion in the interconnect for an arbitrary cluster configuration and a set of communication patterns extracted by PFProf. In the evaluation, the accuracy of the simulation results obtained from PFSim is assessed. Furthermore, how PFAnalyzer can be used to analyze the effect of programmable interconnect control is demonstrated.

To address the second challenge, Chapter 3 proposes a framework to accelerate MPI collectives by dynamically controlling the packet flow in the interconnect. Message Passing Interface (MPI) is a standardized inter-process communication library widely used to develop parallel distributed applications for clusters. Out of the communication primitives provided by MPI, this research focuses on accelerating collective communication

because it occupies a significant fraction of the execution time of applications. The network programmability provided by Software-Defined Networking is integrated into MPI collectives in such a way that MPI collectives are able to effectively utilize the bandwidth of the interconnect. In particular, this research aims to reduce the execution time of MPI_Allreduce, which is a frequently used MPI collective communication in many simulation codes. The speedup of MPI_Allreduce when using the proposed collective acceleration framework is evaluated.

To address the third challenge, Chapter 4 proposes UnisonFlow, a software-defined coordination mechanism that performs interconnect control in synchronization with the execution of applications. In real-world applications, the communication pattern changes with the execution of application. Therefore, a mechanism to coordinate packet flow control and execution of application is essential. UnisonFlow is a kernel-assisted mechanism that realizes such coordination on a per-packet basis while maintaining significantly low overhead. Evaluation verifies that the interconnect control can be successfully performed in synchronization with the execution of the application and the overhead imposed by the coordination mechanism is small.

Chapter 5 concludes this dissertation and discusses future works.

Contents

List of Figures

1 Introduction

1.1 Background

1.1.1 High Performance Computing

Recent breakthroughs in science have significantly benefited from high performance computing (HPC). In modern science, computer simulation is heavily used because it can substitute experiments that are physically intractable or excessively expensive to conduct or observe. HPC allows scientists to simulate natural phenomena in an unprecedented scale and resolution, thereby helping scientists to develop a better understanding of nature and answer fundamental questions about our surrounding environment.

A wide spectrum of science has taken advantage of the massive computing capability provided by HPC. Various phenomena ranged from atomic scale to cosmological scale are simulated on HPC systems. For instance, molecular dynamics simulation reveals the molecular-level structure and property of matter and their interaction. This knowledge is used to design better drugs and materials. Earthquake and Tsunami simulation allows us to predict the impact of seismic activities and prepare for future natural disasters. In addition to simulation applications, data analysis and machine learning applications are also starting to leverage HPC systems.

Scientists have been trying to tackle increasingly larger and more complex problems. This ever-increasing demand from scientists has been the driving force behind the continuous improvement of computing performance. Figure 1.1 shows the development of computing performance of HPC systems based on the data published by the Top500 [1] list. The Top500 list is a biannual list of 500 most powerful HPC systems measured by the maximal LINPACK benchmark performance achieved. The plot clearly indicates the steady increase in performance over the past 20 years. Researchers and engineers have been striving to sustain this growth in performance and reach exascale (Exa FLOPS) in

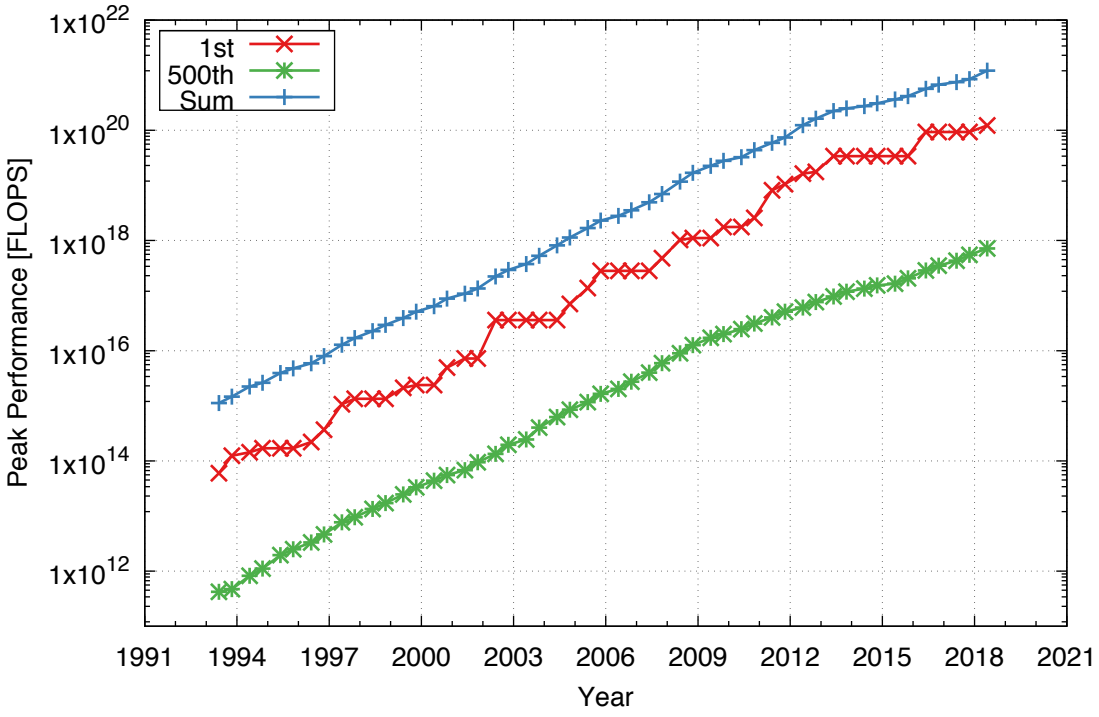


Figure 1.1: Performance Development of Top500 HPC Systems [1]

the near future. To achieve this daunting technological challenge, every aspect of the HPC system including hardware, operating system, middleware and application needs to be greatly improved, optimized and even redesigned.

1.1.2 Cluster Architecture

Modern HPC systems mostly adopt *cluster* architecture to achieve their massive computing performance. In fact, 87% of the recent Top500 systems as of July 2018 are based on the cluster architecture. A cluster is an aggregation of interconnected computers working cooperatively. Computers that constitute a cluster perform computation in parallel and exchange data to be required with one another.

Figure 1.2 shows the architectural overview of a typical cluster. A cluster consists of multiple computers (*i.e., compute nodes*), and a high-performance and low-latency network that integrates (*i.e., interconnect*) them together as a single system. For the purpose of sharing input and output data between the compute nodes, a shared file system is usually deployed as a part of the cluster. Since many users share a single cluster, a *job scheduler* is also commonly deployed to efficiently and effectively manage the computing

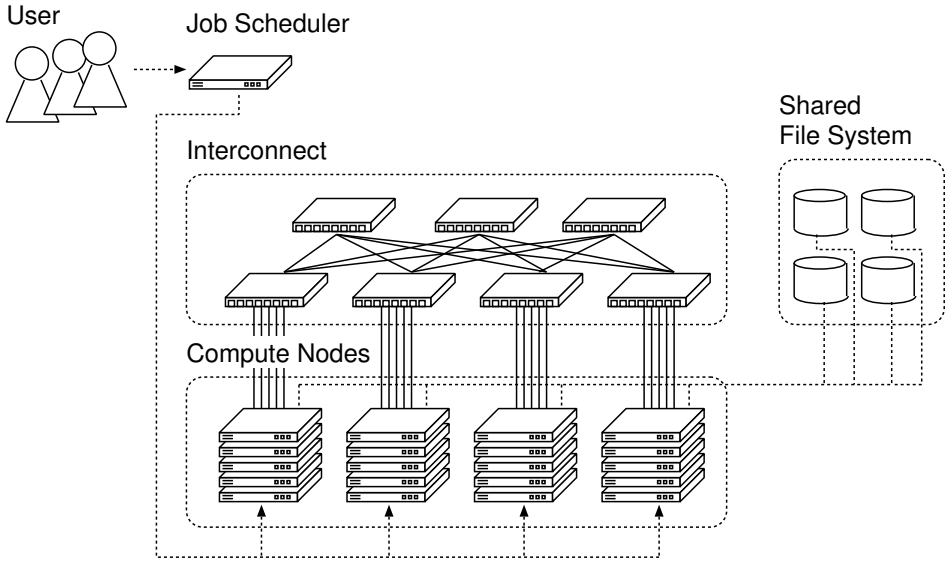


Figure 1.2: Cluster Architecture

resources in a cluster. A job scheduler accepts job requests from users and determines when to run each job to fulfill the request. The scheduler is also responsible for allocating compute nodes to the job request and launching a job on the allocated nodes.

The number of compute nodes composing a cluster has a strong trend to increase. Although the computing performance of a single processor and compute node have been steadily improving, the growth is not fast enough to meet the high demand for computing power from the scientists. Therefore, the designers of HPC systems need to scale out the number of compute nodes to further improve the total computing performance of the cluster. As a result, a single cluster accommodates tens of thousands of compute nodes and millions of cores nowadays.

The bandwidth and latency of communication between compute nodes over the interconnect, or the communication performance, is essential to the scalability of the cluster. In general, compute nodes need to frequently communicate with one another during parallel computation to exchange intermediate results and control messages. If the communication between compute nodes becomes a performance bottleneck, simply adding compute nodes to the cluster does not increase the total performance of the cluster. Therefore, great efforts have been put to the research and development of high-performance interconnects.

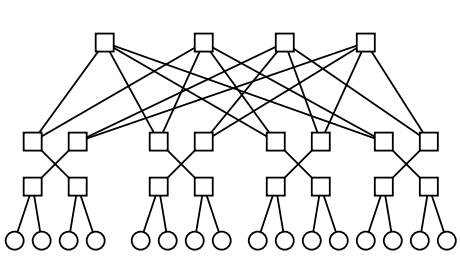
1.1.3 Interconnect

The main goal of high-performance interconnects is to provide high bandwidth and low latency in communication between a large number of compute nodes. In fact, the state-of-the-art interconnects provide more than 100 Gbps bandwidth and less than 1 μs latency between tens of thousands of compute nodes. While achieving such high communication performance, the monetary cost to build and maintain the interconnect must be reasonable.

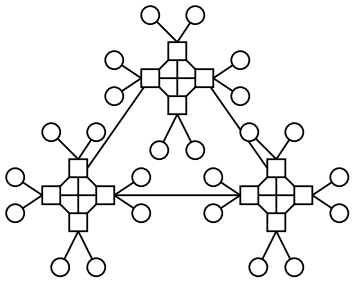
Ethernet [58] and InfiniBand [70] are network technology standards commonly utilized for interconnects today. Ethernet is a long-standing network technology that has been ubiquitously used in both local area networks and wide are networks. InfiniBand, on the other hand, is a network standard specifically designed for HPC. InfiniBand offloads most of its protocol stack onto hardware and realizes mechanisms such as kernel bypassing and Remote Direct Memory Access (RDMA) to reduces the communication latency. In addition to Ethernet and InfiniBand, some HPC system vendors develop proprietary interconnects for their systems. For instance, Cray has developed Gemini [45] interconnect and Aries [37] interconnect for their systems. Fujitsu has developed Tofu [36] interconnect.

Topology is a key factor that determines the performance of an interconnect. A fully-connected topology is ideal since it has dedicated links between any pair of compute nodes. However, implementing a fully-connected topology in a large scale cluster is unrealistic due to extremely high cost and complexity. Therefore, various topologies have been proposed to balance the trade-off between cost and performance. Figure 1.3 illustrates some of the popular topologies. For example, fat-tree [74], dragonfly [53], multi-dimensional torus [36, 45], and hypercube [67] have been widely adopted as interconnect topologies in HPC systems.

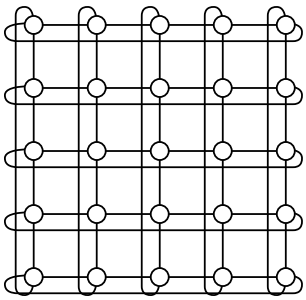
Mostly, the interconnects of computer cluster systems are full-bisection. The bisection bandwidth for an interconnect is defined as the minimum bandwidth between two halves of the interconnect. A full-bisection interconnect is an interconnect whose bisection bandwidth is larger or equal to the aggregated bandwidth between compute nodes. Interconnects that are not full-bisection are referred as oversubscribed. Full-bisection interconnect design are preferred since such interconnect does not suffer from network contention even in the worst case scenario where one half of the compute nodes communicate with the other half at maximum speed of their network interfaces. This



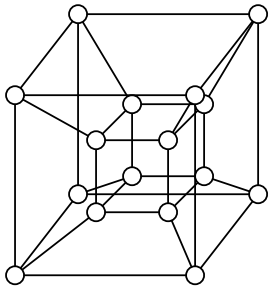
(a) Fat-tree



(b) Dragonfly



(c) 2D Torus



(d) 4D Hypercube

Figure 1.3: Topology of Interconnects

characteristic is beneficial for applications since it removes the need to be aware of the current contention state of the interconnect. However, there is a common problem with full-bisection design: the monetary cost to implement such a design increases superlinearly as the number of node scales out. Therefore, researches have have worked on the effective utilization of oversubscribed interconnects [9, 10] under the assumption that oversubscribed interconnects become inevitable in the future.

1.1.4 Message Passing Interface

Message Passing Interface (MPI) [19] is a *de facto* standard specification for inter-process communication libraries used to develop parallel applications running on distributed memory system such as clusters. MPI defines a suite of communication primitives that help application developers to build parallel distributed applications that require complex communications among compute nodes.

A remarkable feature of MPI is that it abstracts the underlying network of clusters. As shown in Fig. 1.4, each interconnect technology requires the application developers to

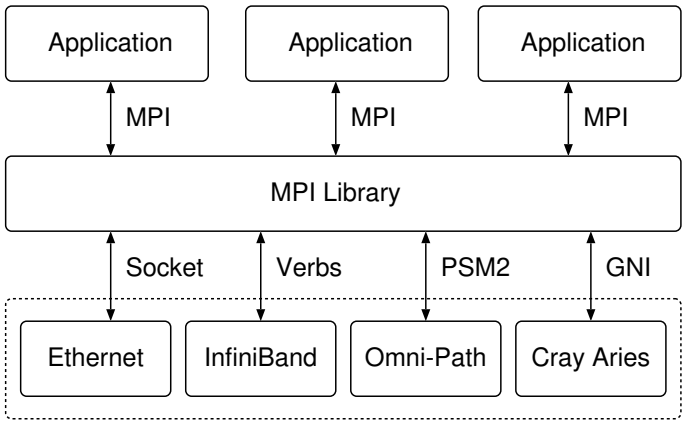


Figure 1.4: Abstraction Provided by MPI

use a different set of APIs. MPI hides this difference and allows application developers to build applications without forcing them to study the detailed architecture or structure of the underlying network. For instance, every process is identified by a *rank* number, a consecutive non-negative integer. The mapping between rank numbers and network addresses is automatically handled by the MPI library. Every process belongs to one or more groups of processes, which are called *communicators*. MPI communication is restricted between processes that belong to the same communicator. These abstractions make MPI applications portable and easy to be ported to different clusters.

The communication primitives defined in MPI can be roughly categorized into point-to-point communication, collective communication and one-sided communication. Table 1.1 shows several examples from each category. Note that this list covers only a small fraction of all the primitives defined in MPI.

Point-to-point communication is a communication between a sender process and another receiver process. For example, the sender calls MPI_Send whereas the receiver calls MPI_Recv. It is similar to BSD sockets, but MPI point-to-point communication differs from sockets in three aspects. First, establishing a connection between the sender and receiver (*i.e.,* connect, listen, accept, *etc.*) is not necessary since the MPI library takes care of it internally. Second, the order of calls to MPI_Send on the sender side and MPI_Recv on the receiver side does not matter due to the internal buffering mechanism in the MPI library. In contrast, calling the recv function after the send function when using sockets may result in packet drops. This characteristic ensures deterministic behavior of applications. Finally, a *tag* can be associated to each message to indicate the type or

Table 1.1: Examples of MPI Primitives

Type	Name	Description
Point-to-point	Send/Recv	Point-to-point send and receive
	Isend/Irecv	Non-blocking point-to-point send and receive
Collective	Bcast	Broadcast
	Reduce	Reduction
	Allreduce	Broadcast result of reduction
	Ibcast	Non-blocking Bcast
	Ireduce	Non-blocking Reduce
	Iallreduce	Non-blocking Allreduce
One-sided	Put	One-sided put
	Get	One-sided get

context of the message. In sockets, messages are merely binary sequences without having any context.

Collective communication involves a group of processes and provides frequently used communication patterns by scientists when implementing parallel algorithms such as broadcast, reduction, all-to-all exchange, *etc.* In principle, any collective communication can be implemented using a combination of point-to-point communication. However, the use of collective communication over the use combinational use of point-to-point communication is generally preferable because the MPI library implements each communication primitive based on optimized algorithms that are selected depending on the message size and number of processes. For instance, a naive implementation of broadcast where the root process sends the same data to each non-root process requires $O(n)$ time to complete (n represents the total number of processes). A more efficient implementation of broadcast is a tree-like data delivery where non-root processes help the root process by sending the received data to other processes that have not received the data yet. The time complexity of tree-like broadcast is $O(\log n)$.

Both point-to-point communication primitives and collective communication primitives have blocking and non-blocking versions. Blocking primitives waits until the communication completes. In contrast, non-blocking primitives returns immediately after

initiating the corresponding communication operation. A separate synchronization is prepared to complete the communication. The advantage of non-blocking communication is that it allows overlapped computation and communication. In other words, computation can be performed while the communication is in progress. Overlapping computation and communication is a way to hide the cost of communication and usually results in better application performance. As a trade-off, non-blocking communication slightly increases the programming effort compared to blocking communication.

One-sided communication is a form of Remote Memory Access (RMA). In point-to-point communication, the sender and receiver need to mutually agree to start a communication. In contrast, one-sided communication allows a process to directly read or write the memory of a peer process without the involvement of its counterpart. This communication model reduces the overhead incurred by memory copy and synchronization points. One-sided communication is especially beneficial on interconnects that support Remote Direct Memory Access (RDMA).

There are multiple implementations of the MPI standard. Open MPI [62], MPICH [69] and MVAPICH [59] are representative examples of actively developed open source implementations of MPI. Furthermore, HPC system vendors such as Cray, Intel and Fujitsu offer their own proprietary MPI implementations based upon the open source implementations, taking the feature and characteristics of their own HPC systems into consideration. Vendor MPI implementations are highly optimized for the vendor's HPC system and able to take advantage of the special features of their system. In addition to the header files and libraries that implement the APIs defined in the specifications, MPI implementations usually offer a launcher that allows users to quickly start MPI processes on multiple compute nodes.

Until today, countless scientific applications have been developed with MPI. Accompanied by the recent scale-out of clusters, the execution time of MPI primitives has become a critical factor that determines the total performance of these applications using MPI. In other words, the total performance of MPI applications can be improved by optimizing the performance of MPI communications. For this reason, researchers have been striving to improve the communication performance of MPI from various aspects.

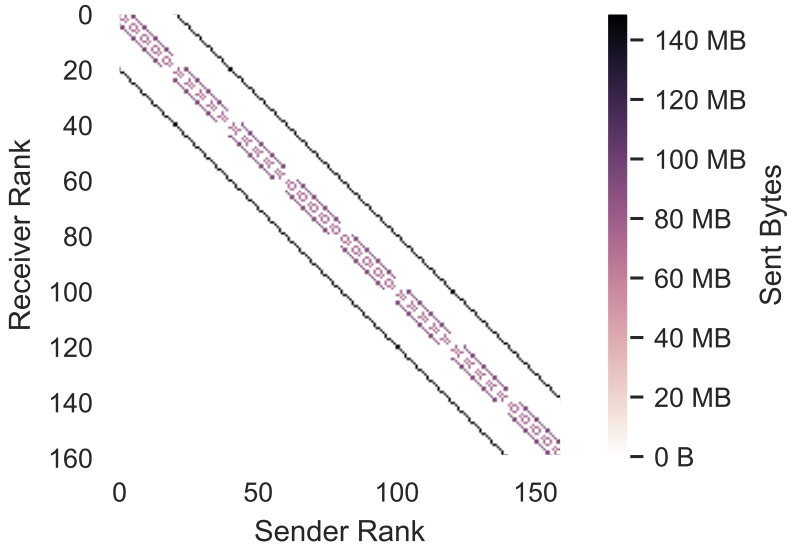
1.1.5 Problem and Motivation

The inter-process communication of MPI applications shows a distinctive pattern. This pattern varies depending on the application and originates from the mathematical model, discretization method, and parallelization strategy. Figure 1.5 (a) shows the communication pattern of a three-dimensional finite-difference solver that performs the nearest neighbor communication as an example. In this figure, the total size of messages sent from a process to another process is visualized using a heat map. The horizontal axis and vertical axis represent the sender and receiver rank, respectively. The visual representation evidently exhibits a regular and local pattern along the diagonal, which originates from the nearest neighbor communication required by the finite-difference method. As another example, Fig. 1.5 (b) shows the communication pattern of a three-dimensional finite-element solver. Clearly, this communication pattern differs from the one of a finite-difference solver shown in Fig. 1.5 (a). The existence of dots apart from the diagonal suggests that distant processes communicate with one another in this application.

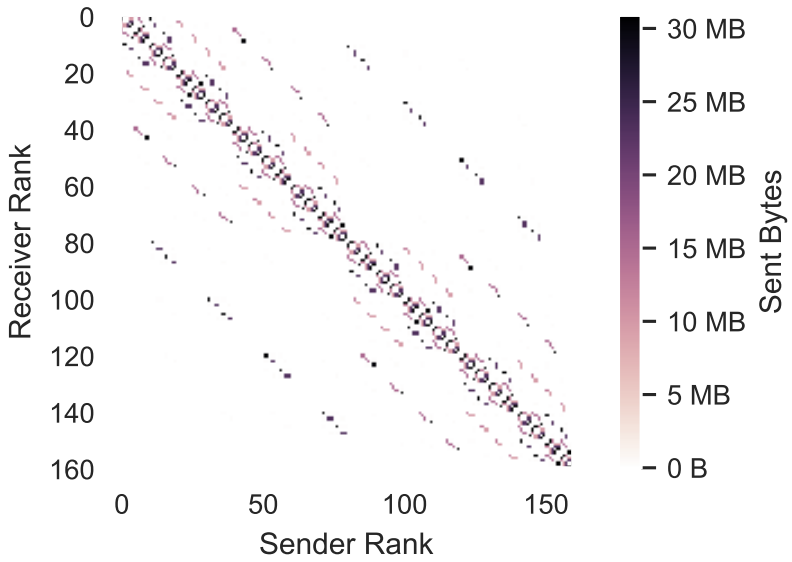
The design of an interconnect could be highly optimized for an application by taking the communication pattern of applications into account. For instance, the interconnect with a three-dimensional torus topology would be ideal for an application that performs nearest neighbor communication in three-dimensional space. However, this approach is infeasible when designing a real-world cluster. The reason can be explained from the fact that HPC systems are shared by many users where each user runs various applications on the cluster. Therefore, in contrast to the application-dependent communication pattern, the interconnect is inherently application-independent.

As a result, a concentration of packet flow in the interconnect, or the imbalance of packet flow, can take place under a certain combination of communication pattern and interconnect. The imbalance can lead to the concentration of traffic on a link in the interconnect and the slowdown of MPI communication that uses the link. The degraded MPI communication can ultimately result in serious degradation of total application performance.

A number of previous studies have tried to address the mismatch between application-dependent communication patterns and application-independent interconnects. Researchers have attempted to adapt the communication pattern of applications to the interconnect. For instance, interconnect-aware MPI collectives [14, 17, 26, 31] have



(a) Three-dimensional Finite-difference Solver



(b) Three-dimensional Finite-element Solver

Figure 1.5: Communication Pattern of Applications

been developed to improve the performance of MPI collectives by taking the interconnect of a cluster into account. MPI implementation often leverages tree-based algorithms to aggregate and distribute messages from and to processes. Interconnect-aware MPI collectives use the information on the interconnect, such as topology and link bandwidth, to build a delivery tree that matches the underlying interconnect of the cluster. Another approach is to optimize the placement of MPI processes on the compute nodes [8, 10, 42]. In this approach, the communication pattern of an application is considered as a weighted graph where nodes represent processes and edges represent the volume of data exchanges between two processes. Various heuristic algorithms have been proposed to embed the communication pattern graph onto the interconnect topology.

To date, however, there has been few studies on adapting the interconnect to the communication pattern of applications. This is mostly because it has been assumed that flexibly and dynamically reconfiguring the interconnect at run-time is infeasible. However, this assumption might not hold anymore with the recent emergence of programmable networking architecture that allows the on-the-fly reconfiguration of the interconnect.

1.1.6 Software-Defined Networking

Software-Defined Networking (SDN) is a novel networking architecture that separates the control plane and data plane into different devices. In conventional networking architectures, the decision on how to handle packets (control plane) and the packet transfer (data plane) are implemented as unified and inseparable features. The separation of the control plane and data plane allows SDN to deliver the following three benefits:

1. *Programmable*: The control plane can be handled by a software controller. Network operators develop software controllers tailored for their needs.

2. *Dynamic*: SDN allows the controller to quickly reconfigure the network. For instance, it is possible to dynamically optimize packet flows in the network based on the real-time traffic pattern.

3. *Centralized*: A centralized controller configures the entire SDN-enabled network, thus reducing efforts to administer and manage the network. In conventional networking architectures, the operators need to configure each network device separately because the control plane is distributed on individual devices.

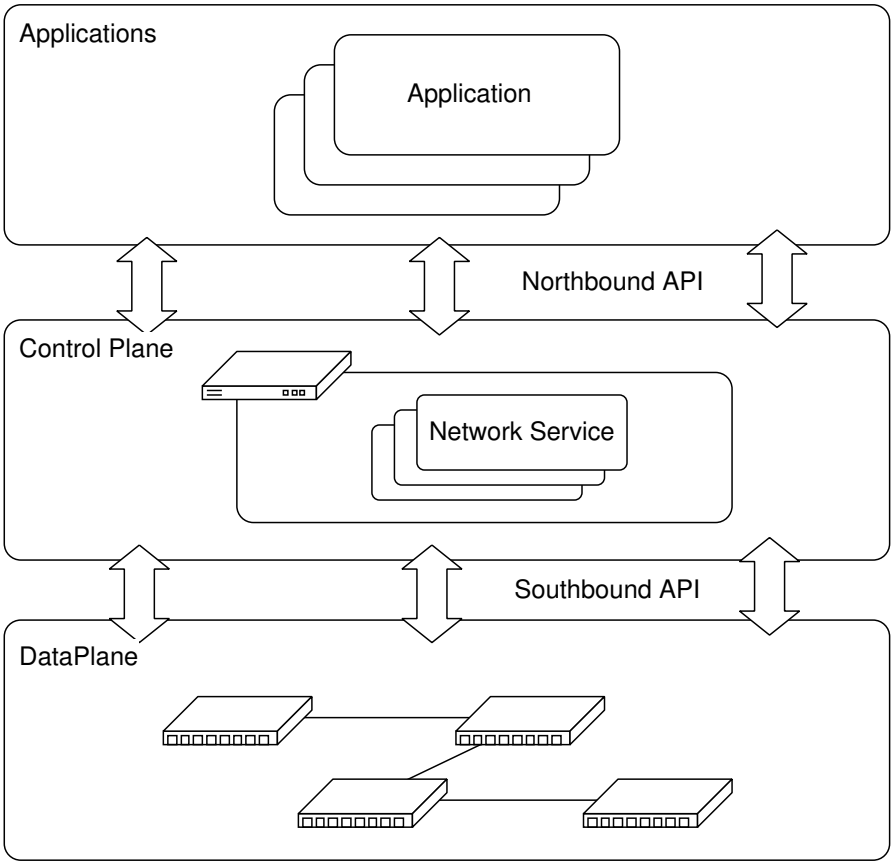


Figure 1.6: Software-Defined Networking Architecture

Matching Condition			Action
Dst MAC	Src IP	Dst IP	
	192.0.2.12	192.0.2.34	Output to port 1
	192.0.2.34	192.0.2.56	Output to port 2
ff:ff:ff:ff:ff:ff			Output to port 1 and 2
72:42:c1:e4:75:8c			Drop

Figure 1.7: An Example of a Flow Table

OpenFlow [55] is a widely accepted open standard of SDN. In an OpenFlow-enabled network, the data plane is handled by OpenFlow switches. Each OpenFlow switch holds a logical construct called a *flow table*, which is a collection of *flow entries*. A flow entry defines what kind of packet control should be performed on what kind of packets (Fig. 1.7). Every time a packet arrives at an OpenFlow switch, the switch looks up a matching flow entry in its flow table using the header fields of the packet. Once the switch finds a matching flow entry, the corresponding action of the matched flow entry is applied to the incoming packet. An arbitrary combination of pre-defined header fields can be used for the matching condition. Table 1.2 shows the twelve header fields defined in OpenFlow 1.0. OpenFlow switches usually implement specialized hardware such as Content Addressable Memory (CAM) or Ternary Content Addressable Memory (TCAM) to perform the matching efficiently.

The OpenFlow controller is a component responsible for the control plane. It manages the flow table of each switch by adding, modifying and removing flow entries. The controller and switches communicate with each other by asynchronously exchanging messages defined in the OpenFlow protocol specification. Table 1.3 shows a list of message types defined in OpenFlow 1.0. In this table, messages are classified into three categories by its initiator. Controller-initiated messages are used by the controller to update or inspect the state of a switch. On the other hand, switch-initiated messages are used by switches to notify the controller of network events and update in the switch state. Symmetric messages are initiated from both sides and used primarily for establishing and maintaining the connection between the controller and switch. Messages may or may not require a response from its receiver depending upon the message type. One of the

Table 1.2: Header Fields Defined in OpenFlow 1.0

Layer	Field Name	Width (bits)
L1	Ingress Port	
L2	Ethernet Source	48
	Ethernet Destination	48
	Ethernet Type	16
	VLAN ID	12
	VLAN Priority	3
L3	IP Source	32
	IP Destination	32
	IP Protocol	8
	IP ToS	6
L4	TCP/UDP Source Port	16
	TCP/UDP Destination Port	16

most frequently used message type is *packet-in*, which is sent out from a switch to the controller when a matching flow entry is missing for an incoming packet. In response, the controller can send a *modify flow entry* message to install a new flow entry on the switch.

OpenFlow controllers are usually implemented as a software for flexibility and reduced development cost. OpenFlow controller frameworks have been developed to support the developments of OpenFlow controller software by providing reusable building blocks. Common building blocks include parser and deparser of OpenFlow messages and common network protocol packets, state machine for the OpenFlow protocol and high-performance concurrent server for handling the connection with the switches. Trema [5], Ryu [29], ONOS [22] and OpenDaylight [27] are among the most popular OpenFlow controller frameworks. Developers of OpenFlow controllers can focus to the application logic by taking advantage of these controller frameworks.

Table 1.3: Messages Types Defined in OpenFlow 1.0

Initiator	Message Type	Response	Purpose
Controller	Packet-Out		Inject a packet to data plane
	Flow-Mod		Add/Modify/Delete a flow entry
	Port-Mod		Modify state of a port
	Stats	✓	Get statistics of individual flow
	Barrier	✓	Synchronize controller and switch
	Queue-Get-Config	✓	Query state of queues
	Features	✓	Get capabilities of switch
	Get-Config	✓	Get fragmentation setting of switch
	Set-Config		Set fragmentation setting of switch
Switch	Packet-In		Notify an unmatched packet
	Flow-Removed		Notify when a flow entry been removed
	Port-Status		Notify status update of a port
Symmetric	Hello		Negotiate OpenFlow version
	Error		Notify failure
	Echo	✓	Check liveness of connection
	Vendor		Vendor-specific extensions

1.2 Research Objective

As discussed in Section 1.1.5, the mismatch between application-dependent communication patterns and application-independent interconnects have led to the imbalance of packet flow in the interconnect. This dissertation aims to address this imbalance problem by establishing a programmable interconnect control that dynamically controls the packet flow in the interconnect based on the communication pattern of applications.

Figure 1.8 illustrates the high-level concept of the envisioned programmable interconnect control in this dissertation. The core idea of this programmable interconnect control is an iteration of three steps: *observe*, *decide* and *adapt*. The envisioned interconnect control functions as follows. First, the inter-process communication patterns of an application are observed and recorded. Second, the decision on how to control the packet flow in the interconnect is made based on the collected communication patterns of the application. The goal of this packet flow control is to mitigate load imbalance in the interconnect and achieve higher communication performance between compute nodes. Third, the interconnect is adapted to perform the planned packet flow control using programmable networking architecture. These three steps are performed in synchronization with the execution of application.

To materialize this concept, the following three challenges must be achieved:

1. *Analyzing the packet flow in the interconnect*: To effectively control the packet flow in the interconnect, interconnect designers first need to carefully analyze and understand the packet flow generated in the interconnect when running an application on a cluster.

2. *Accelerating MPI communication by dynamically controlling the packet flow in the interconnect*: Given a communication pattern of an application and an interconnect, interconnect designers need to determine how to control the packet flow in the interconnect to mitigate load imbalance in the interconnect and improve the performance of MPI communication.

3. *Coordinating the execution of application and interconnect control*: The communication pattern of the application changes rapidly with the execution of the application. Therefore, the interconnect control must be performed in accordance with the execution of application.

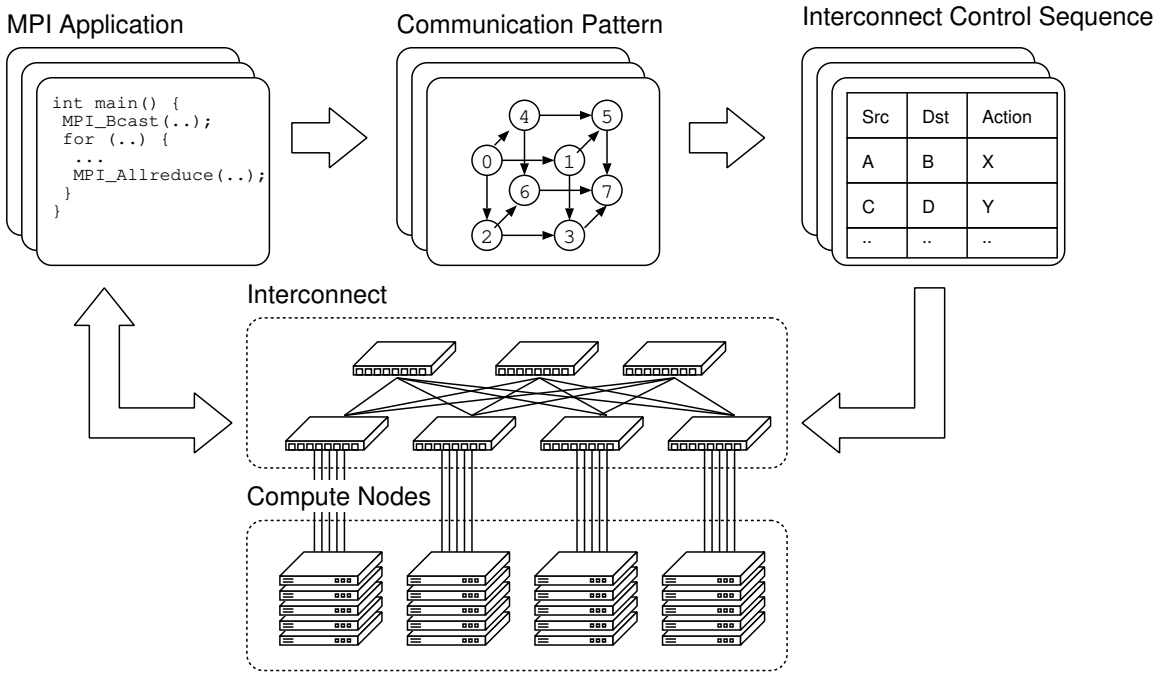


Figure 1.8: Concept of the Envisioned Programmable Interconnect Control

1.3 Organization of the Dissertation

The rest of this dissertation is organized as follows. Chapter 2 proposes PFAnalyzer, a toolset for analyzing the packet flow in an interconnect, to address the first challenge listed in Section 1.2. The packet flow generated in the interconnect highly depends on parameters such as the communication pattern of application, interconnect design and cluster configuration. When designing and implementing an efficient programmable interconnect control, researchers need to conduct a systematic analysis over many combinations of these parameters. Because performing such analysis on a physical cluster is time-consuming, this dissertation utilizes simulation to facilitate the analysis. PFAnalyzer is a pair of two tools: PFSim, an interconnect simulator capable of simulating programmable interconnects, and PFProf, a profiler to accurately collect communication patterns from MPI applications. PFSim allows interconnect researchers and designers to rapidly investigate possible congestion in a programmable interconnect for a wide variety of cluster configurations and communication patterns collected by PFProf. This dissertation evaluates the accuracy of the simulation results obtained from PFSim and demonstrates how PFAnalyzer can be used to analyze the effect of programmable interconnect control.

Chapter 3 addresses the second challenge. Out of the communication primitives provided by MPI, this dissertation focuses on accelerating collective communication because it occupies a significant fraction of the execution time of applications. This dissertation proposes a framework to accelerate MPI collectives by dynamically controlling the packet flow in the interconnect. The network programmability provided by Software-Defined Networking is integrated into MPI collectives so that collectives are able to effectively utilize the bandwidth of the interconnect. In particular, this dissertation aims to reduce the execution time of MPI_Allreduce, which is a frequently used MPI collective communication in many simulation codes. The speedup of MPI_Allreduce when using the proposed collective acceleration framework is evaluated.

Chapter 4 addresses the third challenge. This chapter proposes UnisonFlow, a software-defined coordination mechanism that performs interconnect control in synchronization with the execution of applications. In a real-world application, the communication pattern changes with the execution of the application. Therefore, a mechanism to coordinate packet flow control and execution of the application is essential. UnisonFlow is a kernel-assisted mechanism that realizes such coordination on a per-packet basis while maintaining significantly a low overhead. The evaluation verifies that the interconnect control can be successfully performed in synchronization with the execution of the application and the overhead incurred by the coordination mechanism is small.

Chapter 5 concludes this dissertation and discusses future works.

2 Toolset for Analyzing Packet Flow in Interconnect

2.1 Introduction

Inter-node communication performance of high-performance computing (HPC) clusters heavily affects the total performance of communication-intensive applications. Communication-intensive applications require low-latency and high-bandwidth communication between compute nodes to fully exploit the computational power and parallelism of the compute nodes. High performance networks that provide such low-latency and high-bandwidth communication between compute nodes of a cluster are often referred to as *interconnects*. Message Passing Interface (MPI) [19, 24] is a commonly used inter-process communication library to describe communication on HPC clusters.

In this dissertation, the interconnects are roughly classified into *static interconnects* and *dynamic interconnects*. In the former category, it is assumed that packet flow is statically controlled solely based on its source and/or destination. A well-known exemplifier is InfiniBand [70], where forwarding tables held by switches are populated with pre-computed forwarding rules in advance of the execution of applications. In contrast, in the latter category, it is assumed that packet flow is dynamically controlled to mitigate load imbalance and improve utilization of the interconnect.

Nowadays, the majority of HPC clusters employ the former static interconnects because of their technological maturity despite the potential cost advantage dynamic interconnects could provide. The static interconnects are controlled without taking the communication patterns of individual applications into account. However, they are usually designed to be able to accommodate the worst-case traffic demand to achieve good performance for a variety of applications, each of which has a different communication pattern. Interconnect designers have respected such criteria as full bisection bandwidth and non-blocking

networks.

The continuously growing demand for computing power from academia and industry has inevitably forced HPC clusters to scale out more and more. As a result of the growing number of compute nodes, interconnects have been increasingly large-scale and complex. This technical trend is making static and over-provisioned interconnects more cost-ineffective and difficult to build.

Based on this background and these trends, this study explores the feasibility and applicability of programming dynamic interconnect within HPC [13]. In particular, *SDN-enhanced MPI* [23, 30], a framework that incorporates the dynamic network controllability of Software-Defined Networking (SDN) [40] into MPI, has been researched based on the idea that dynamically optimizing the packet flow in the interconnect according to the communication patterns of applications can increase the utilization of the interconnect and then improve application performance. The goal of SDN-enhanced MPI is to accelerate individual MPI collectives by dynamically optimizing the packet flow in the interconnect. Several MPI collectives have been successfully accelerated in the previous works up to this time. One of the core challenges in the research on SDN-enhanced MPI lies in the design of an algorithm to effectively control the packet flow in the interconnect for each MPI collective called by the application.

More generally, an algorithm that takes the communication patterns of applications as its input and then determines how to manage the packet flow is essential towards realizing a dynamic and application-aware interconnect. In order to develop a generic algorithm that achieves good performance on a variety of applications and interconnects, the algorithm must be investigated and evaluated by targeting different applications and interconnects. However, utilizing actual clusters to analyze the performance characteristics of the interconnect is restricted in the following three points. First, the execution time of real-world HPC applications typically ranges from hours up to days, sometimes even months. Second, large-scale deployments of dynamic interconnects that allow execution of highly parallel applications have not yet been seen because the research and development of dynamic interconnects are still at their early stage. Third, network hardware such as switches may not support measuring traffic in the interconnect with enough high frequency and precision to obtain meaningful insights.

From the three restrictions mentioned above, an interconnect simulator that allows researchers to conduct a systematic investigation of clusters with diverse topologies

and parameters is essentially demanded to accelerate the research and development of application-aware dynamic interconnects that control packet flow in response to the communication patterns of applications. A wide spectrum of interconnect simulators have been developed with different focus and purpose until today. However, existing simulators mostly focused on static interconnects and few research has been done to simulate dynamic and application-aware interconnects.

This chapter describes the design and implementation of PFAnalyzer, a toolset for analyzing application-aware dynamic interconnects. PFAnalyzer consists of two components: PFSim and PFProf. PFSim is an interconnect simulator specialized for application-aware dynamic interconnects. PFSim takes a set of communication patterns derived from applications and a cluster configuration as its input and then simulates the traffic on each link of the interconnect. PFProf is a custom profiler to extract communication patterns from applications which are supplied to PFSim.

The contributions of this chapter are summarized as follows:

- A lightweight interconnect simulator for simulating dynamic and application-aware interconnects is proposed.
- A custom profiler for extracting communication patterns from applications is presented.
- Simulation results for NAS CG benchmark and MILC on a fat-tree interconnect are presented to demonstrate the practicality of of the proposed toolset.

The rest of this chapter is organized as follows. Section 2.2 examines the requirements of an interconnect simulator for dynamic and application-aware interconnects. Section 2.3 describes the design and implementation of PFAnalyzer. Section 2.4 evaluates the accuracy of the simulated traffic. Furthermore, the NAS CG benchmark and MILC are taken as example applications to demonstrate how PFAnalyzer can be used by researchers to analyze the joint effect of node allocation, process placement and routing on the distribution of traffic in the interconnect. Section 2.5 reviews the related work. Section 2.6 concludes this chapter and outlines future work.

2.2 Research Objective

This chapter aims to realize a toolset for analyzing the packet flow in the interconnect to facilitate the research and development of application-aware dynamic interconnects. The packet flow generated in an interconnect highly depends on parameters such as the communication pattern of application, interconnect design and cluster configuration. When designing and implementing an efficient application-aware dynamic interconnect, researchers need to conduct a systematic analysis over many combinations of these parameters. However, conducting such analysis on a physical cluster can be extremely time- and resource-consuming. Therefore, this research takes the approach of utilizing simulation to facilitate the analysis.

This research attempts to provide a toolset that performs a lightweight simulation of the interconnect and predicts the packet flow generated in the interconnect. The toolset should allow researchers to predict the packet flow in the interconnect by supplying the communication pattern of real-world application alongside with the interconnect design and cluster configuration. The predicted packet flow can then be analyzed by researchers to gain an insight on the interconnect while significantly saving the time and cost compared to experiments on a physical cluster. Based on the discussion above, the toolset should be able to satisfy the following requirements.

- *Extraction of communication patterns from applications*: Communication patterns of real-world HPC applications should be fed into the simulator to reproduce the characteristics of communication generated by real-world applications. To simulate the packet flow in the interconnect when an application is being executed, the simulator needs to predict the volume of point-to-point communication exchanged between compute nodes. In the actual computing scene, the traffic among compute nodes is generated by the processes executed on the compute nodes. Thus, some means to analyze the traffic volume of point-to-point communication exchanged between the processes from applications is essential. In this chapter, applications that leverage MPI for inter-process communication are targeted.

- *Reproduction of process placement characteristic*: As described in Section 1.1.2, multiple jobs are running concurrently on a real-world cluster. Under such a cluster environment, the process placement characteristic of the job scheduler such as job

scheduling, node selection and process placement algorithms heavily affect the distribution of packet flow generated in the interconnect. Since the job scheduler and its configuration largely varies across deployments, the process placement characteristic of the job scheduler needs to be reproduced.

- *Efficient prediction of packet flow in the interconnect*: The simulator should be designed to be lightweight and fast to carry out a large number of simulations with different parameters in a reasonable amount of time. If necessary, appropriate approximation should be introduced to improve simulation performance.

2.3 Proposal

This research proposes *PFAnalyzer*, a toolset for analyzing the performance characteristics of application-aware dynamic interconnects. PFAnalyzer is composed of *PFProf*, a profiler to extract communication patterns from applications, and *PFSim*, a simulator capable of simulating application-aware dynamic interconnects.

2.3.1 Representation of a Communication Pattern

In this dissertation, the communication pattern of an application is represented using a traffic matrix of the application. The reason why this research has adopt traffic matrix as representation of communication pattern is explained below.

The traffic matrix of an application is defined as a matrix of which element is equal to the traffic volume exchanged between two processes in the application. Here, the volume of traffic between processes is approximated as being constant during the execution of a job. In other words, the traffic volume between a process pair is assumed to be the total bytes transferred divided by the runtime of the application.

This approximation is introduced to reduce the size of the communication pattern as well as to simplify and to speed up the simulation. The idea behind this approximation is based on the fact that many HPC applications (*e.g.* partial differential equation solvers) exhibit an iterative nature. These applications spend most of their execution time inside a repetitive loop and thus their communication patterns do not significantly change over time. Therefore, omitting the temporal change of the communication pattern and assuming the traffic volume between processes as being constant is a good approximation. In the

future, trace segmentation [12] techniques can be applied on the communication trace. This technique segments a trace into multiple communication phases, which could then be simulated individually to further improve the accuracy of simulation for applications with significantly time-varying communication patterns.

2.3.2 PFProf (MPI Profiler)

PFProf is a profiler that extracts the inter-process communication patterns from MPI applications. The collected communication patterns are later fed into PFSim to simulate the packet flow generated in the interconnect.

To capture the inter-process communication of an MPI application, either profiling or static analysis is commonly utilized. Profiling gives accurate results but requires the users to run their application once. In contrast, static analysis of the source code does not require the users to run their application. However, the communication patterns obtained through the use of static analysis are usually less accurate compared to the patterns obtained by profiling. This research takes the profiling approach because an accurate communication pattern is essential to obtain a simulation result with high fidelity.

Initially, existing MPI profilers and tracers such as Score-P [39], Vampir [54] and TAU [61] were considered for collecting the traffic matrices from MPI applications. However, these tools can capture only a subset of the communication pattern when profiling an application that uses MPI collectives. The reason can be explained from the following technological aspect. Existing MPI profilers replace the standard MPI functions provided by MPI libraries with instrumented functions by utilizing the MPI Profiling Interface (PMPI). Although this approach works regardless of a specific MPI implementation, it fails to capture the function calls made within the MPI library. Meanwhile, collective communication functions are internally implemented as a combination of point-to-point communication in MPI implementations. These underlying point-to-point communication functions are hidden from PMPI-based profilers and excluded from the communication patterns emitted by profilers. Therefore, an instrumentation mechanism other than PMPI is required to capture the hidden point-to-point communication.

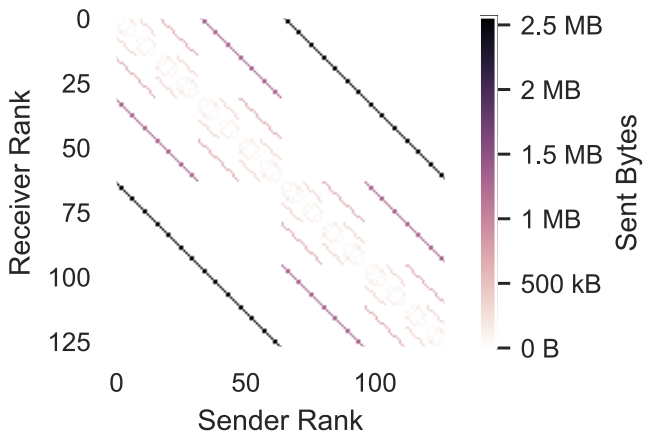
Furthermore, the combination of point-to-point communication that composes a collective communication is unknown until the collective communication function is called during the execution of the application. This is because MPI libraries usually

implement multiple algorithms for each MPI collective that are selected depending on the message size and number of processes. For example, Open MPI [64] implements three algorithms to realize MPI_Allgather: the recursive doubling algorithm, the Bruck algorithm, and the ring algorithm [66]. Figure 2.1 illustrates the underlying point-to-point communication of MPI_Allgather for each algorithm. It is evident that the point-to-point communication of MPI_Allgather is significantly different depending on the algorithm being used. For the reason, the mapping between a collective communication and its corresponding underlying point-to-point communication cannot be statically generated before executing the application.

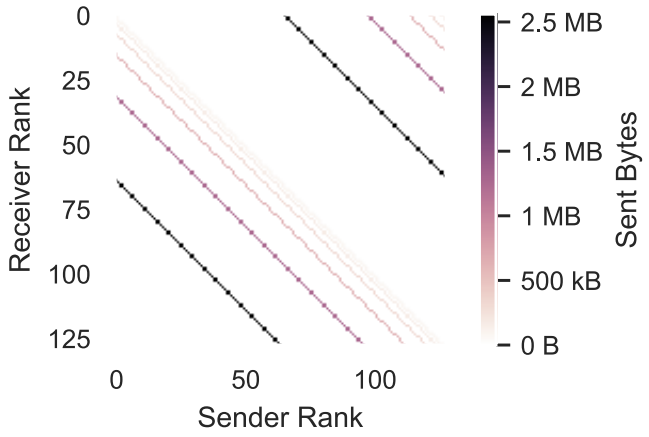
To accurately capture the underlying point-to-point communication of collective communication, this research has taken the strategy of combining the MPI Performance Examination and Revealing Unexposed State Extension (PERUSE) [60] with PMPI. PERUSE was designed to provide the internal information of an MPI implementation that was not exposed through PMPI to applications and performance analysis tools. PERUSE delivers the internal information of an MPI implementation to applications and performance analysis tools through an event-driven API.

Figure 2.2 illustrates how PFProf, the MPI library and an MPI application interact with one another. PFProf intercepts the function calls from the application to several MPI functions using PMPI. MPI_Init and MPI_Finalize are intercepted to perform initialization and finalization when the application starts or exits (step 1 in Fig. 2.2). In addition, PFProf intercepts MPI functions that create or destroy communicators to maintain a mapping between global ranks (rank number within `MPI_COMM_WORLD`) and local ranks (rank number within a communicator created by the user). This mapping is necessary because PERUSE events are reported with local ranks, while profiling results should be described with global ranks for the ease of analysis. During the initialization, PFProf subscribes to two PERUSE events: `PERUSE_COMM_REQ_XFER_BEGIN` and `PERUSE_COMM_REQ_XFER_END` (step 2). These events are emitted each time a transfer of a message begins and ends, respectively.

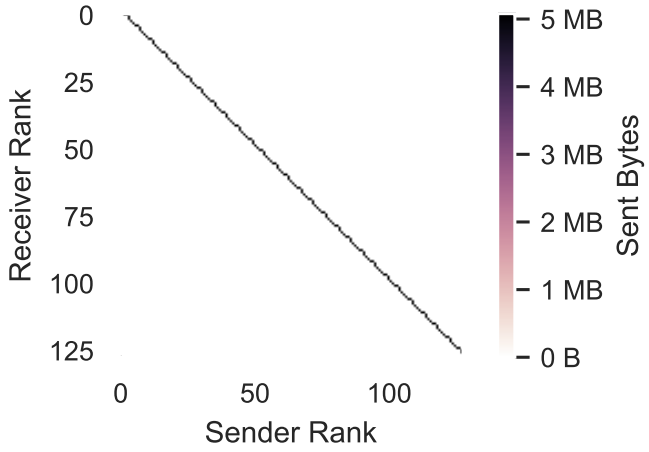
After the application starts, PFProf receives PERUSE events from the MPI library every time the application calls an MPI function that causes inter-process communication (step 3). PERUSE extracts the sender, receiver and transferred bytes from each PERUSE event and updates the traffic matrix online. Once the application calls MPI_Finalize, the communication pattern is written out to disk as a JSON file (step 4).



(a) Recursive Doubling Algorithm



(b) Bruck Algorithm



(c) Ring Algorithm

Figure 2.1: Underlying Point-to-Point Communication of MPI_Allgather

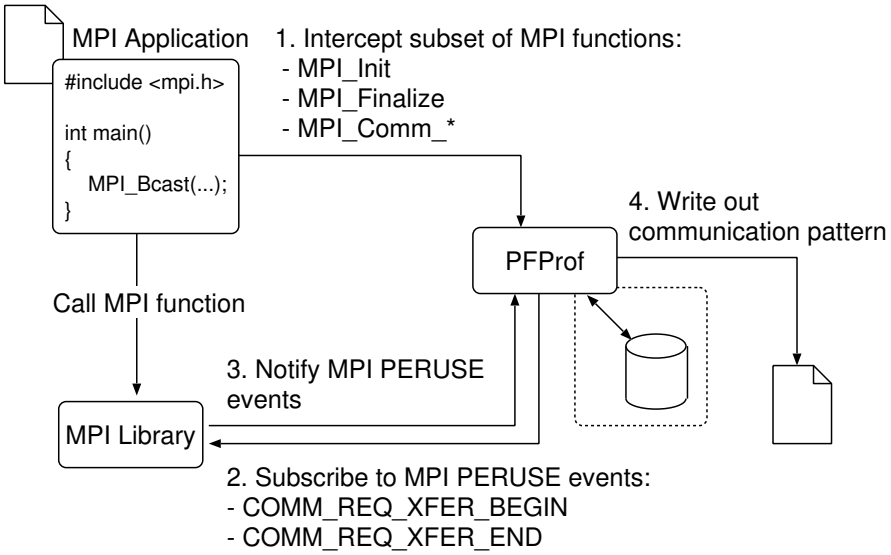


Figure 2.2: Block Diagram of PFProf

Finally, PFProf is designed to be provided in the form of a shared library so that users do not need to modify the source code of their applications. Users can either set the `LD_PRELOAD` environment variable to load the shared library at run-time or dynamically link the shared library with their application at build-time.

2.3.3 PFSim (Interconnect Simulator)

PFSim uses a set of communication patterns of applications and a cluster configuration as its input and then simulates the packet flow generated by the applications. The packet flow is aggregated per link to compute the traffic load on each link. The simulated traffic load of links can be summarized into statistics for quantitative analysis or visualized. Using these outputs from PFSim, users can locate hot-spots and assess load imbalance in the interconnect. These insights on the interconnect can be useful for designing better algorithms for controlling the packet flows in application-aware dynamic interconnects.

Methodologies for simulating interconnects are roughly classified into packet-level simulation [46] and flow-level simulation [49]. In the packet-level simulation, the behavior of how each packet travels through the interconnect is precisely reproduced. Therefore, the communication time of applications can be predicted accurately in exchange for long execution time and large memory foot print. In contrast, the flow-level simulation estimates the steady state behavior of the interconnect and does not track individual

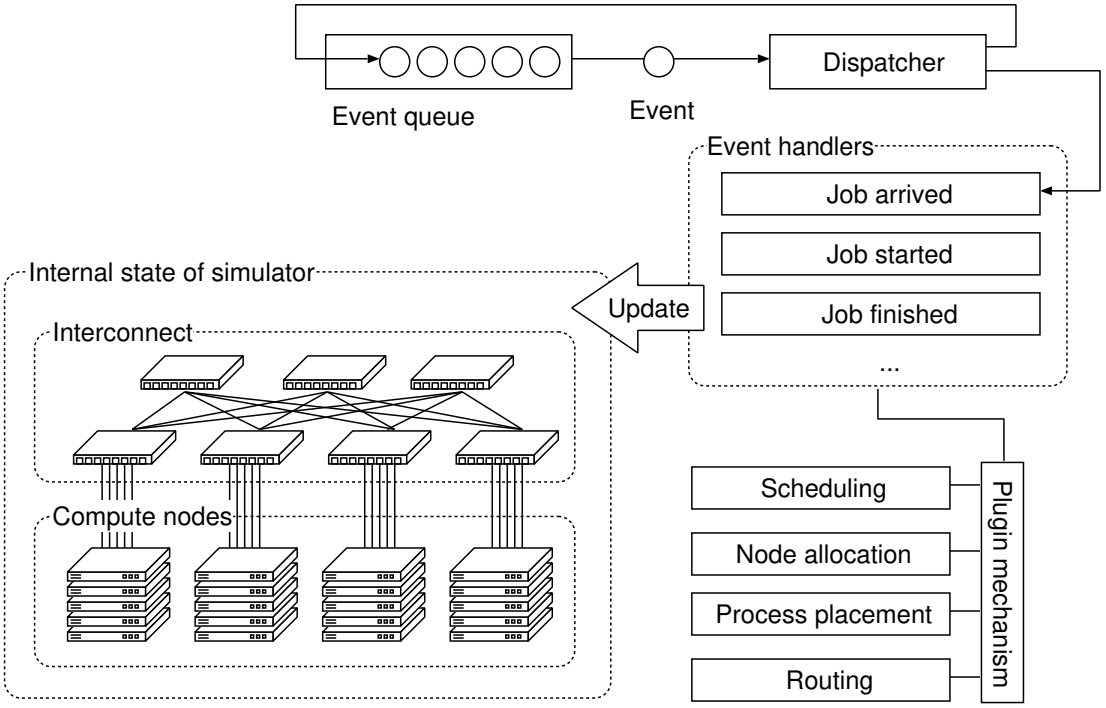


Figure 2.3: Internal Structure of PFSim

packets. Thus, the packet flow in the interconnect can be speedily estimated compared to the packet-level simulation. As a trade-off, predicting the communication time of applications using flow-level simulation is challenging. As described in Section 2.2, this research aims at realizing a toolset for efficiently predicting the packet flow in the interconnect under diverse configurations. Therefore, PFSim is based on the flow-level simulation for execution efficiency.

Figure 2.3 shows the internal structure of PFSim. This simulator is based on a discrete-event simulation model. Under this model, the simulation is driven by events, each of which indicates a change in the internal state of the simulator. Each event holds (1) type of the event, (2) time when the event will occur, and (3) additional information indicating what kind of state change the event will cause. An event queue is a priority queue that stores the events prioritized by the time each event occurs. An event handler is a function that is associated to a specific event type and invoked when an event of the associated type occurs. The dispatcher manages the event queue. It pops events from the event queue one by one and calls the associated event handler for each event.

In PFSim, three event types exist: job-arrived, job-started and job-finished. The event

handler for the job-arrived event first checks if the job queue is empty and if there are enough compute nodes unallocated to run the job. If the job is not runnable at the time, the job is enqueued to the job queue and the event handler finishes. If the job is runnable, the event handler initiates the job startup routine shown in Algorithm 1. First, a set of available compute nodes are allocated to the job (line 1 in Algorithm 1). Next, the placement of processes on the allocated compute nodes is determined (line 2–3). Subsequently, each process is associated to the compute node that accommodates it (line 4–5). Finally, a job-started event is inserted into the event queue (line 6).

Algorithm 1: Job start routine

```
1 nodes ← allocateNodes(job);
2 procs ← Set of processes composing job;
3 mapping ← mapProcs(procs, nodes);
4 foreach (proc, node) ∈ mapping do
5 |   Associate proc with node;
6 Insert job-started event to event queue;
```

Algorithm 2 shows the overview of the event handler for the job-started event. This event handler first obtains the traffic matrix of the application that has started (line 1 in Algorithm 2). Next, based on the traffic matrix, the traffic load on each link in the interconnect is updated as follows: first, compute nodes that accommodate the source and destination processes are obtained (line 3–4). Subsequently, the path from the source compute node to destination compute node is calculated (line 5–8). Lastly, the traffic load on each link along the path is increased based on the amount of traffic transferred between the source and destination processes (line 9–10). After the traffic load is updated, the event handler inserts a job-finished event into the job queue (line 11).

The event handler for the job-finished event releases the compute nodes allocated to the job. If there are one or more jobs in the job queue, one of them is selected. If there are enough number of available compute nodes, the job startup routine shown in Algorithm 1 is invoked.

PFSim aims at predicting the packet flow in the interconnect generated by applications under diverse cluster configurations. The configuration of the simulation must be able to be edited by users. For the reason, the configuration is described in a human-readable

Algorithm 2: Event Handler for Job-started Event

```
1  tm ← Traffic matrix of job;
2  foreach (srcProc, dstProc, traffic) ∈ tm do
3      src ← Compute node that accommodates srcProc;
4      dst ← Compute node that accommodates dstProc;
5      if path between (src, dst) is already computed then
6          path ← Cached path between (src, dst);
7      else
8          path ← route(src, dst, job);
9      foreach link ∈ path do
10         Increase traffic load of link for traffic;
11 Insert job-finished event to event queue;
```

simulation scenario file. The simulation scenario file is designed to be described in a structured serialization format called YAML. YAML has been adopted for its high readability and editability compared to other alternatives such as JSON and XML.

Listing 2.1 shows an example of a simulation scenario file. In this simulation scenario file, the topology of the interconnect (line 1 in Listing 2.1), output directory for the simulation results (line 2), and a set of jobs to simulate (line 16–21) are specified. Moreover, the algorithms that control the execution and communication of jobs, which are shown in Table 2.1, are also specified (line 3–15). Each configuration value can be a list of parameters. The simulation is executed multiple times, each time with a different combination of configuration values until all combinations are completed. In Listing 2.1, one scheduling algorithm (line 4–5), two node allocation algorithms (line 6–8), two process placement algorithms (line 9–11), and three routing algorithms (line 12–15) are specified. When this scenario file is supplied to PFSim, 12 simulations are executed in total in a combination manner.

Since PFSim accepts the topology of the interconnect as its input and outputs the traffic load on each link of the interconnect, the file format for representing the interconnect needs to be considered. The file format should be designed for graphs since the interconnect can be considered as a graph. In addition, the format should be supported by existing analysis

Listing 2.1: Example of a Simulation Scenario

```
topology: topologies/milk.graphml
output: output/milk-cg-dmodk
algorithms:
  scheduler:
    - pfsim.scheduler.FCFSScheduler
  node_selector:
    - pfsim.node_selector.LinearNodeSelector
    - pfsim.node_selector.RandomNodeSelector
  process_mapper:
    - pfsim.process_mapper.LinearProcessMapper
    - pfsim.process_mapper.CyclicProcessMapper
  router:
    - pfsim.router.DmodKRouter
    - pfsim.router.GreedyRouter
    - pfsim.router.GreedyRouter2
jobs:
  - submit:
      distribution: pfsim.math.ExponentialDistribution
      params:
        lambd: 0.1
    trace: traces/cg-c-128.tar.gz
```

Table 2.1: List of Configurable Algorithms

Algorithm	Description
Job Scheduling	Selects the job to execute from the job queue. (*e.g.* FCFS, Backfill)
Node Selection	Selects which compute nodes to assign for a job. (*e.g.* Linear, Random, Topology-aware algorithms)
Process Placement	Determines on which compute node to place a process. (*e.g.* Block, Cyclic, Application-aware algorithms)
Routing	Computes a route between a pair of processes. (*e.g.* D-mod-K, S-mod-K, Random, Dynamic algorithms)

and visualization tools so that users can take advantage of these software assets. Based on the requirements above, PFSim is designed to use GraphML [33], an XML-based markup language for graphs, as its input and output format of the interconnect. Popular graph visualization tools such as Cytoscape and Gephi can be used to view and edit GraphML files. Users can use these tools to visually and intuitively locate bottlenecks and load imbalance in the interconnect.

2.4 Evaluation

The first experiment is conducted to verify if PFProf is able to capture the underlying point-to-point communication behind collective communication. Then, the accuracy of the simulation performed by PFSim is evaluated through the comparison of the traffic estimated by PFSim with the traffic measured on a cluster when actually running an application. Then, the performance of point-to-point communication between two processes with and without PFProf are compared to assess the overhead incurred by PFProf. Lastly, how PFAnalyzer can be used by researchers to analyze the joint effect of node allocation, process placement and routing on the distribution of traffic in the interconnect is demonstrated.

2.4.1 Comparison of PFProf and Conventional Profiler

This experiment verifies if PFProf is able to capture the underlying point-to-point communication of collective communication. A simple MPI application is profiled using TAU and PFProf. Subsequently, the extracted communication patterns are compared. Listing 2.2 shows the source code of the MPI application. This application first executes MPI_Allreduce, one of the commonly used collective communications. Then, every process performs a point-to-point communication with its neighboring process. This application is executed with 128 processes.

Figure 2.4 (a) shows the communication pattern obtained with TAU. In this figure, the horizontal and vertical axis indicate the sender and receiver rank, respectively. The point-to-point communication between the neighbor processes is clearly visualized as a pattern along the diagonal. However, the underlying point-to-point communication of MPI_Allreduce was not observed. On the other hand, Figure 2.4 (b) shows the

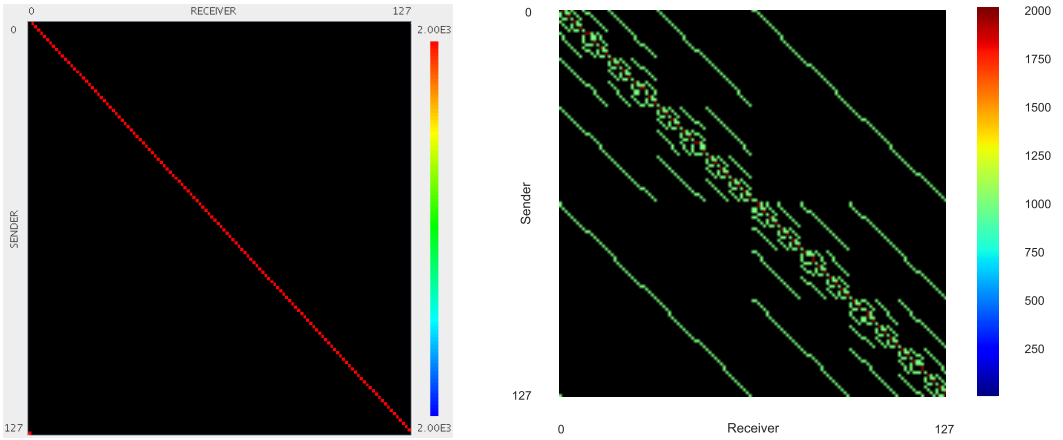
Listing 2.2: MPI Application Used for Evaluation

```
1  #include <stdio.h>
2  #include <mpi.h>
3
4  #define BUF_SIZE (1000)
5
6  int rank, size;
7  MPI_Request req;
8  char buf[BUF_SIZE] = {};
9  char buf2[BUF_SIZE] = {};
10
11 int main(int argc, char *argv[])
12 {
13     MPI_Init(&argc, &argv);
14
15     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
16     MPI_Comm_size(MPI_COMM_WORLD, &size);
17
18     /* Collective communication */
19     MPI_Allreduce(buf, buf2, BUF_SIZE, MPI_CHAR,
20                   MPI_SUM, MPI_COMM_WORLD);
21        \label{fig:allgather-ring}
22
23     /* Point-to-point communication */
24     MPI_Irecv(buf, BUF_SIZE, MPI_CHAR,
25               (rank - 1) % size,
26               0, MPI_COMM_WORLD, &req);
27     MPI_Send(buf, BUF_SIZE, MPI_CHAR,
28              (rank + 1) % size,
29              0, MPI_COMM_WORLD);
30     MPI_Wait(&req, MPI_STATUS_IGNORE);
31
32     MPI_Finalize();
33 }
```

communication pattern obtained with PFProf. This communication pattern reveals the MPI communication generated from MPI_Allreduce in detail. The underlying point-to-point communication is clearly detailed. These observations suggest that PFProf is able to capture the underlying point-to-point communication of collective communication.

2.4.2 Accuracy of Traffic Estimated by PFSim

This experiment evaluates the accuracy of the simulation performed with PFSim. The traffic estimated by PFSim and the traffic measured when running an application on an actual cluster are compared.



(a) TAU (b) PFProf

Figure 2.4: Extracted Communication Patterns

The simulated cluster was modeled after a small-scale cluster installed at our institution. The cluster is composed of 20 compute nodes, each equipped with two quad-core Intel Xeon E5520 processors. Compute nodes are interconnected with a two-tier fat-tree topology as illustrated in Fig. 2.5. A single NEC ProgrammableFlow PF5240 switch is logically divided into six switches that constitute the fat-tree topology. The upper-layer two switches are referred to as spine1–spine2 and the lower-layer four switches as leaf1–leaf4. The CG benchmark from the NAS Parallel Benchmark Suite [73] was executed with 128 processes on 16 compute nodes. Since the CG benchmark only allows power-of-two number of processes, some compute nodes could not be utilized.

Figure 2.6 shows the comparison of simulated traffic using PFSim and the measured traffic on the original cluster. The traffic on each link between the spine switches and leaf switches were normalized by the traffic on the link spine1→leaf1 and shown in the plot. The plot indicates the error of simulation result is small. The largest error was 1.9% and was observed on the link leaf4→spine1. These results indicate that the simulation result is sufficiently accurate to analyze performance of dynamic interconnects.

2.4.3 Overhead Incurred by PFProf

In this experiment, the performance of point-to-point communication between two processes with and without PFProf were compared to inspect the overhead incurred by the profiler. OSU Micro Benchmark [4] was used to measure the throughput and latency

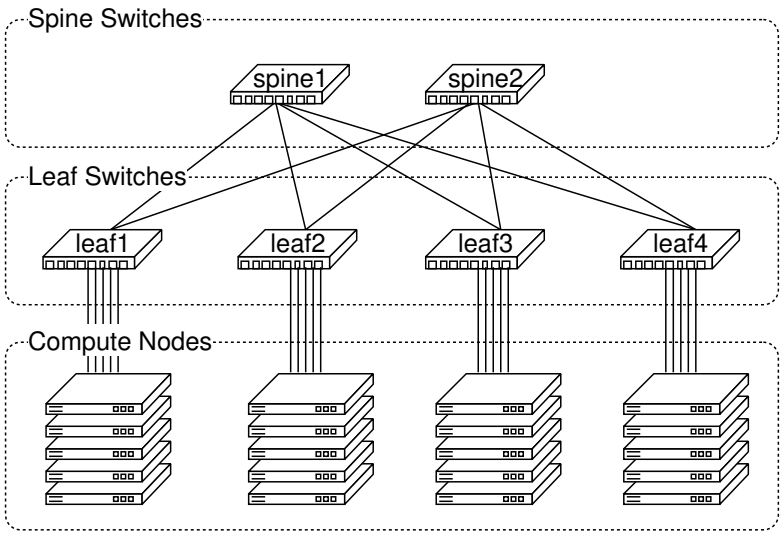


Figure 2.5: Simulated Cluster with Fat-tree Interconnect

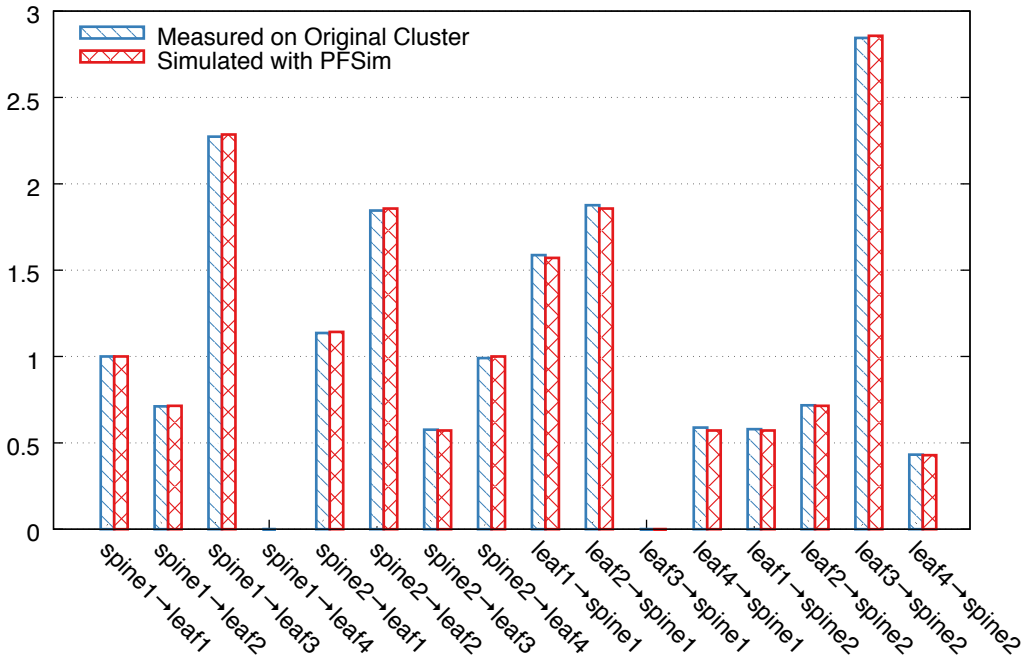


Figure 2.6: Comparison of Simulated Traffic against Measured Traffic

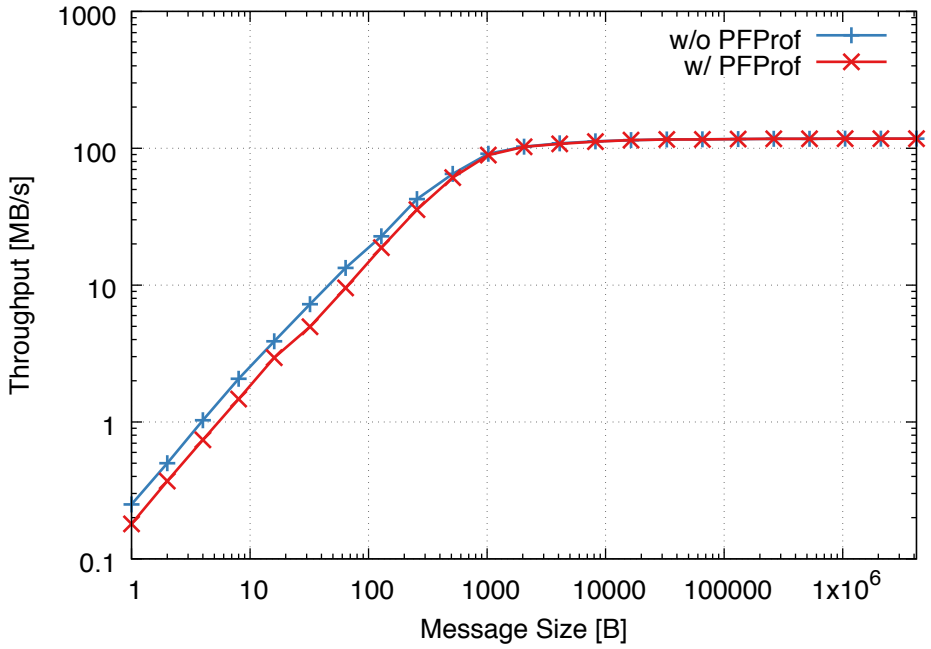


Figure 2.7: Throughput of Point-to-point Communication

of point-to-point communication between two processes for varying message sizes. The comparison of throughput and relative throughput are shown in Fig. 2.7 and Fig. 2.7, respectively. For messages larger than 1KB, the overhead was ignorable. For messages smaller than 1KB, up to 30% of overhead was incurred. Comparison of latency and relative latency are shown shown in Fig. 2.9 and Fig. 2.10, respectively. These plots suggest that there is almost no overhead for latency. These results indicate that PFProf is able to extract the communication pattern from applications without significantly hurting the performance of applications.

2.4.4 Use Case of PFAnalyzer

This experiment shows how PFAnalyzer can be used by researchers to analyze the joint effect of node allocation, process placement and routing on the distribution of traffic in the interconnect. Communication-intensive MPI applications were executed on the proposed simulator. The maximum traffic load observed on links composing the interconnect was compared in both cases of static interconnect control and dynamic interconnect control. The maximum traffic load observed on the links was used as an indicator of the communication performance of an application. In most cases, a hot spot link can

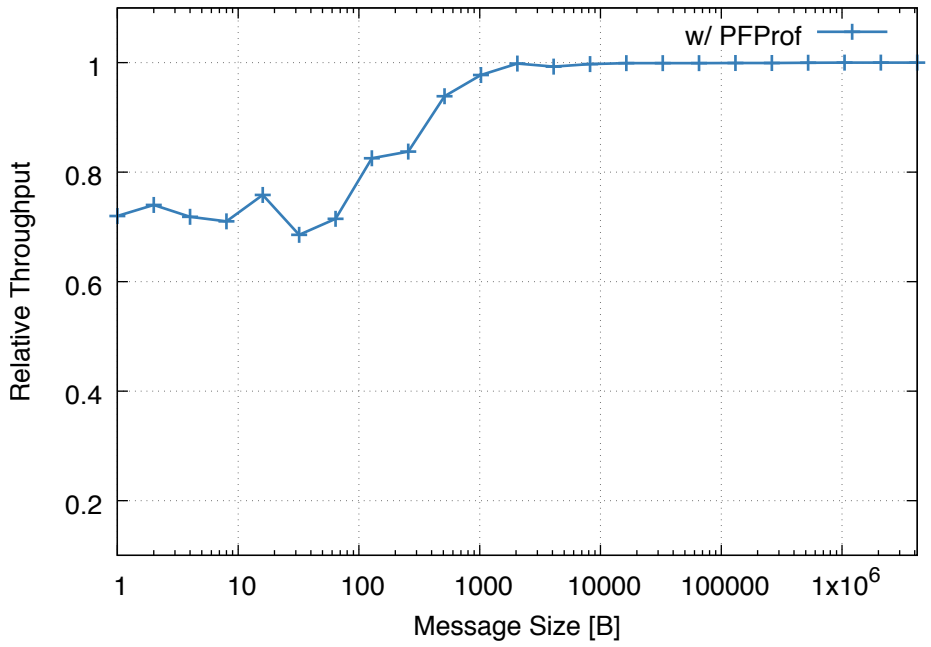


Figure 2.8: Relative Throughput of Point-to-point Communication

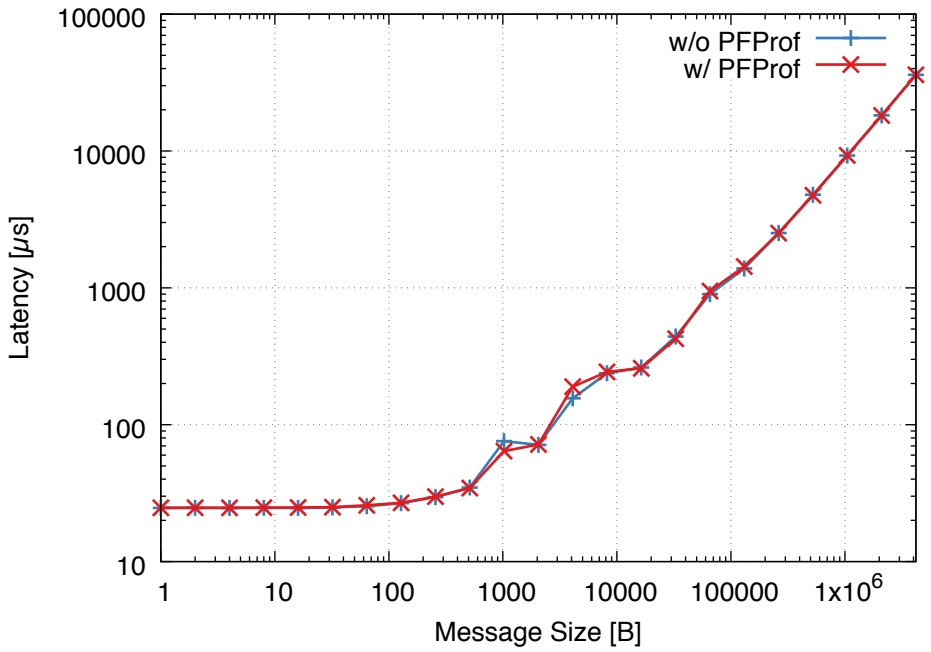


Figure 2.9: Latency of Point-to-point Communication

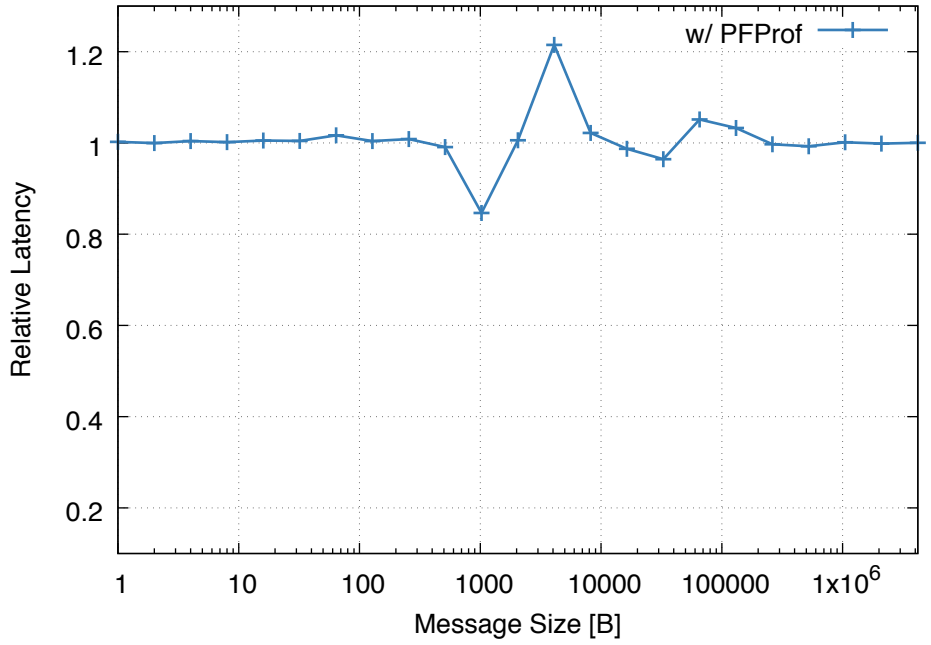


Figure 2.10: Relative Latency of Point-to-point Communication

slow down the whole application, because every process needs to wait until the slow communication crossing the hot spot link completes when collective communication or synchronization is performed by an application. Therefore, mitigating the traffic load on the hot spot link is considered to improve the performance of the application.

Two applications were selected as representatives of communication-intensive applications. The first one was the CG benchmark from the NAS Parallel Benchmark Suite [73]. The CG benchmark estimates the largest eigenvalue of a sparse matrix using the inverse power method. Internally it uses the conjugate gradient method, which frequently appears in irregular mesh applications. The second application (`ks_imp_dyn`) was from MIMD Lattice Computation (MILC) [3], a collection of applications used to study Quantum Chromodynamics (QCD). As for the input data, the data set provided by NERSC as a part of the NERSC MILC benchmark was used. These two applications were executed with 128 MPI processes. Thread parallelism was not put in use (*i.e.,* flat MPI model was adopted).

To analyze the effect of dynamic interconnect control, simulations were carried out using static routing and dynamic routing control. Furthermore, in order to investigate the impact of node selection and process placement to the traffic load, the node selection algorithm and process placement algorithm were also changed. As a result, exhaustive

combinations of two node selection algorithms, two process placement algorithms and two routing algorithms were investigated with the scheduling algorithm fixed. Below are the descriptions of the algorithms used in this experiment:

- Scheduling: A simple *First-Come First-Served (FCFS)* scheduling without back-filling was adopted.
- Node Selection: Either *linear* or *random* node selection was adopted. Linear node selection assumes that compute nodes are lined up in a one-dimensional array and minimizes the fragmentation of allocation. This is essentially the same as the default node selection policy of Slurm [68]. Random node selection, as the name indicates, randomly selects compute nodes. This algorithm simulates a situation where the allocation of compute nodes is highly fragmented.
- Process Placement: Either *block* or *cyclic* process placement was adopted. Block process placement assigns rank i to the $\lfloor i/c \rfloor$-th compute node where c represents the number of cores per node. Cyclic process placement assigns rank i to the (i mod n)-th compute node where n denotes the number of compute nodes.
- Routing: Either *D-mod-K* routing or a *dynamic* routing was adopted. Destination-modulo-K (D-mod-K) routing is a popular static load balancing routing algorithm that distributes packet flow over multiple paths based on the destination address of the packet. The dynamic routing algorithm implemented here computes and allocates routes from the heaviest communicating process pair. A route is computed to minimize the traffic of the maximum-traffic link in the path.

Under this condition, the maximum traffic load observed on links through the simulation were measured and compared. Figure 2.11 shows the simulation results in the NAS CG benchmark. In this graph, the blue hatched bars represent the results for D-mod-K routing while the red crosshatched bars represent the results for dynamic routing. The vertical axis represents the simulated maximum traffic load normalized by the maximum traffic load when linear node selection, block process placement and D-mod-K routing were adopted.

What stands out in Fig. 2.11 is that dynamic routing consistently achieves lower traffic load compared to static D-mod-K routing. The reduction of traffic load was the largest

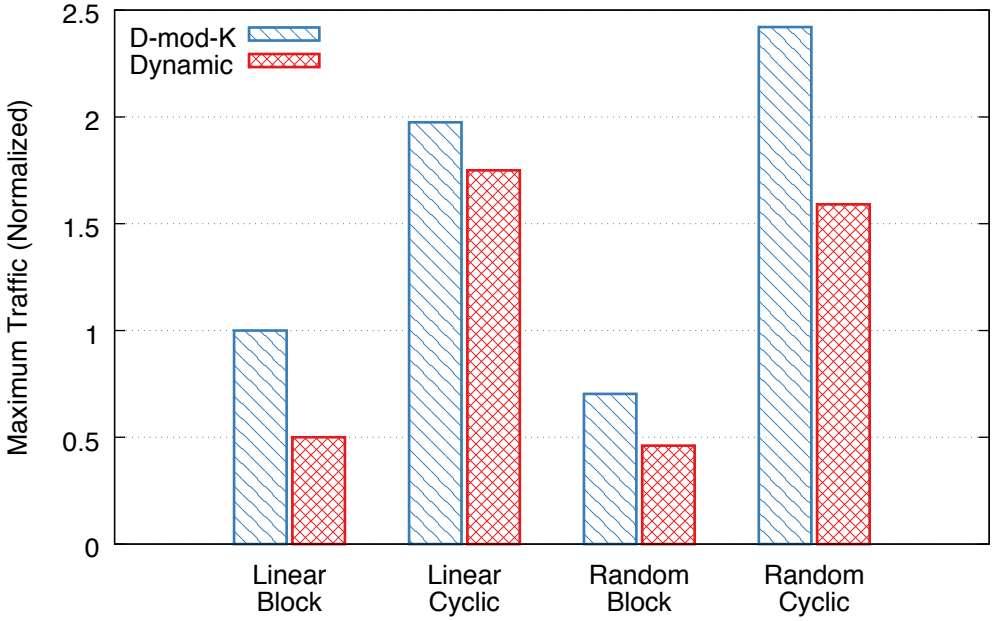


Figure 2.11: Comparison of Maximum Traffic (NAS CG)

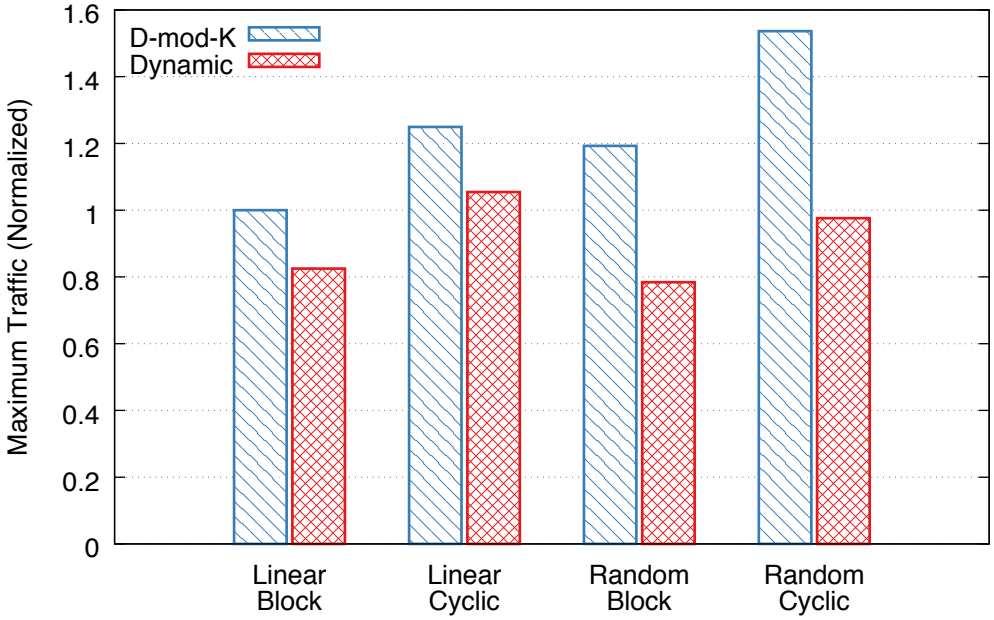


Figure 2.12: Comparison of Maximum Traffic (MILC)

when linear node selection and block process placement were adopted. Under this combination of node selection and the process placement algorithm, dynamic routing slashed the maximum traffic load by 50% in comparison with D-mod-K routing. Also, the graph reveals that cyclic process placement always increased maximum traffic load compared to block process placement because neighboring ranks were placed on different compute nodes despite the locality of the communication pattern.

Figure 2.12 shows the result in the case of MILC. The graph reveals that dynamic routing outperforms D-mod-K routing. In this case, the reduction of the link load was the largest when random node selection and cyclic process placement was adopted. When using linear node selection and block process placement, the reduction of the maximum link load was 18%. Compared to NAS CG benchmark, the effect of dynamic routing was smaller.

To investigate the impact of traffic load on the application performance of an actual environment, the configuration described in the previous Section 2.4.4 was reproduced on a actual cluster and then the execution time of each benchmark was measured. This cluster was equipped with switches that support OpenFlow, which is a de facto standard implementation of SDN. The routing algorithms were implemented based on OpenFlow. In this experiment, linear node selection and block process placement was adopted. The average execution time of 10 runs was compared when using D-mod-K routing and dynamic routing. Figure 2.13 shows the measured execution time for both benchmarks. The use of dynamic routing reduced the execution time of NAS CG benchmark for 23%. Meanwhile, the execution time of MILC benchmark was reduced for 8%, which was smaller than the case of NAS CG benchmark. This matches with the simulation result that predicted NAS CG benefits from larger reduction in maximum traffic load by using dynamic routing compared to MILC.

These results suggest that application performance is actually improved by alleviating the traffic load on the hot spot link. This suggestion implies that researchers working on dynamic interconnects can take advantage of the proposed toolset to simulate different packet flow controlling algorithms and assess their performance improvement effect on real-world applications by using indicators such as maximum traffic load.

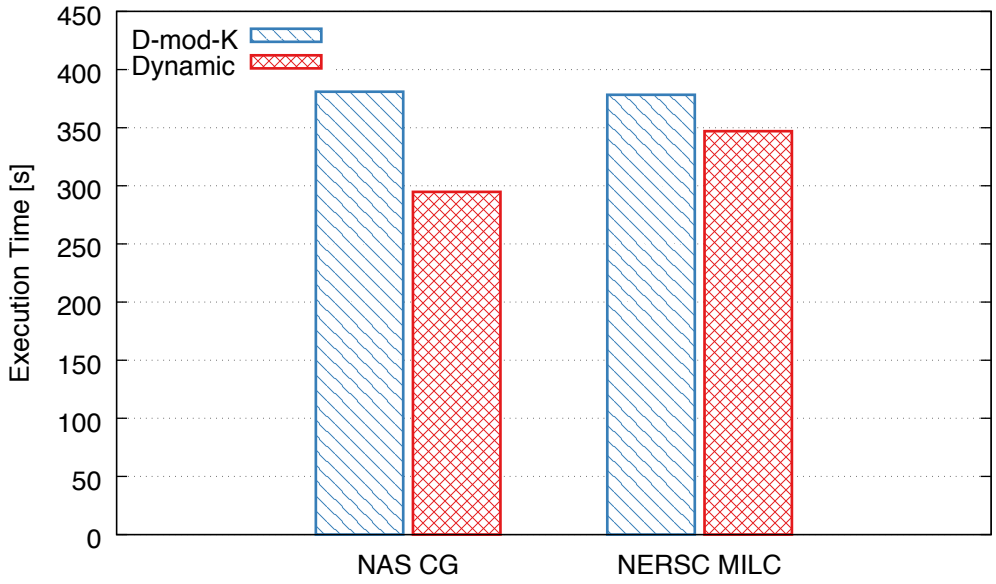


Figure 2.13: Comparison of Execution Time on an Actual Cluster

2.5 Related Work

The novelty of PFAnalyzer is three-fold. First, PFProf can record the underlying point-to-point communication of a collective communication, which conventional MPI profilers failed to capture. Second, PFSim rapidly simulates the traffic and identify load imbalance in an interconnect owing to an adequate approximation of the communication pattern. Third, PFSim is capable of simulating programmable interconnects, which were not targeted by conventional interconnect simulators.

Several interconnect simulators have been proposed in the past research. PSINS [50] is a trace-driven simulator for HPC applications. Traces obtained from applications are used to predict the performance of applications on a variety of HPC clusters with different configurations. LogGOPSim [46] simulates the execution of MPI applications based on the LogGOP network model. A limitation of LogGOPSim is that the interconnect is assumed to have full bisection bandwidth and thus congestion is not simulated. These two simulators can provide accurate performance predictions owing to their per-message simulation capability. However, the topology and the routing algorithm of interconnects are abstracted in the network models of PSINS and LogGOPSim. Therefore, these simulators cannot be used for predicting and comparing the performance of different topologies or routing algorithms. In contrast, the simulator proposed in this chapter

allows users to compare the performance characteristic of different topologies and routing algorithms.

ORCS [49] simulates the traffic load on each link in the interconnect for a given topology, communication pattern and routing algorithm in the same way as PFSim. The simulated traffic load of links can be summarized into various performance metrics and used for further analysis. A limitation of ORCS is that only pre-defined communication patterns can be used as its input. Moreover, ORCS assumes static routing as in InfiniBand. On the contrary, PFSim can handle dynamic routing algorithms that use communication patterns of applications and interconnect usage to make routing decisions.

In [18], simulations are carried out to examine the performance characteristics of an SDN-based multipath routing algorithm for data center networks. A simulator was developed based on MiniSSF to simulate the throughput and delay of a packet flow under diverse settings. However, communication patterns are randomly generated and not based on real-world applications. PFSim is designed to accept arbitrary communication patterns obtained from real-world applications using our custom profiler.

INAM^2 [15] is a comprehensive tool to monitor and analyze network activities in an InfiniBand network. The tight integration with the job scheduler and a co-designed MPI library allows *INAM*^{2} to associate network activities with jobs and MPI processes. For instance, it can identify hot spots in the interconnect and inspect which node, job, and process is causing the congestion. Although *INAM*^{2} is a useful tool for system administrators to diagnose the performance issues of interconnects, it is not suitable for studying diverse interconnects since it only supports physical clusters.

2.6 Conclusion

This chapter described the design and implementation of PFAnalyzer, a toolset for analyzing the performance characteristics of application-aware dynamic interconnects. PFAnalyzer is composed of PFProf, a profiler to extract communication patterns from applications, and PFSim, a simulator capable of simulating application-aware dynamic interconnects. PFSim takes a set of communication patterns of applications and a cluster configuration as its input and then simulates the traffic on each link of the interconnect. Evaluation conducted in this dissertation verified that PFProf is able to capture the underlying point-to-point communication of collective communication, which is essential

for understanding the communication characteristic of applications. Also, it also indicated that the incurred overhead is small. Furthermore, the error of traffic simulated with PFSim was 1.9% at maximum, which is small enough to analyze the performance characteristic of interconnects.

PFAnalyzer contributes to realizing a programmable interconnect control adaptive to the communication pattern of application in two aspects. First, PFProf allows researchers to accurately extract the inter-process communication pattern from applications. The obtained communication pattern is a crucial information for understanding the communication characteristic of applications and adapting the interconnect control to the application. Second, PFSim helps researchers to design and implement effective algorithms to control the packet flow in the interconnect for mitigating imbalance in the interconnect and achieving higher communication performance between compute nodes.

Further work is necessary to investigate the performance characteristics of dynamic interconnects on large-scale and highly parallel clusters. Moreover, application-aware node selection and process placement algorithms are planned to be implemented on PFSim. The impact of such application-aware algorithms on the performance of dynamic interconnects should be evaluated.

3 Accelerated MPI Collective Using Software-Defined Networking

3.1 Introduction

As described in Section 1.1.4, communication primitives defined in MPI can be roughly categorized into point-to-point communication, collective communication and one-sided communication. Out of these three categories, this dissertation particularly focus on accelerating collective communication because of its significant impact to the application performance. In fact, a recent analysis of MPI usage on a production HPC system has revealed that approximately 34% of the total core-hours of the system are expended in MPI communication and 66% of the total MPI communication time is spent in collective communication [7]. This fact clearly indicates that reducing the time of collective communication is of great importance.

MPI collectives require intensive communication among multiple compute node pairs. However, some compute node pairs communicate less, whereas other pairs have to communicate much more. Because of this imbalance, even in the case where the interconnect of a cluster system has multiple redundant routes between compute nodes, the packet flow generated from MPI collectives could collide on a single link of the interconnect without any control of packet flow.

This chapter aims at accelerating MPI collectives by dynamically controlling the packet flow in the interconnect. In particular, a cluster system deployed with an interconnect that contains multiple redundant routes is targeted. This is because the majority of interconnects today are provisioned with redundant routes to improve the bisection bandwidth and fault tolerance. Specifically, this research specifically focuses on fat-tree. This research designs and implements a framework that effectively makes use of redundant routes by dynamically controlling the packet flow in the interconnect using SDN described

in Section 1.1.6.

The rest of this chapter is organized as follows. Section 3.2 reviews conventional traffic balancing methods and analyzes their problems. Subsequently, the goal of this chapter is clarified. In Section 3.3, the design and implementation of the proposed architecture of SDN-enhanced MPI_Allreduce is presented. In Section 3.4, an evaluation on a cluster system is conducted to verify the feasibility of the proposed framework. Section 3.5 discusses related works. Section 3.6 concludes this chapter and discusses challenges for further improving the practicality of our proposed framework.

3.2 Research Objective

MPI collective operations require a large number of simultaneous communication among multiple process pairs. However, some compute node pairs communicate less, whereas other pairs have to communicate much more. When the underlying network of the cluster system that interconnects with the compute nodes has a full-bisection bandwidth, the imbalance of source and destination in communication would not be a problem although structuring this type of a network is hard to scale out due to economic and physical restrictions [51]. Under the assumption that the network is oversubscribed, the shortage of available bandwidth could give rise to a serious problem.

Suppose a cluster system of four compute nodes interconnected with a two-tier fat-tree topology as illustrated in Fig. 3.1. Fat-tree is a network topology that has multiple redundant routes between the upper layer switches and lower layer switches. By distributing traffic among these redundant routes, it is possible to gain higher bisection bandwidth compared to a simple tree topology.

The traffic load balancing algorithm deployed on the interconnect becomes important on such an interconnect. For example, suppose compute node 1 is sending data to node 3, and node 2 is sending different data to node 4 at the same moment in Fig. 3.1. If those two communications are routed within the exact same route, link contention could happen. When these two communications are routed so that they make use of the redundant routes as depicted in Fig. 3.2, link contention can be avoided.

Various algorithms for balancing traffic in a network with redundant routes have been proposed. Equal-Cost Multi-Path routing (ECMP) [25] is a standardized load balancing strategy mainly used in L3 switches. For each communication between two compute

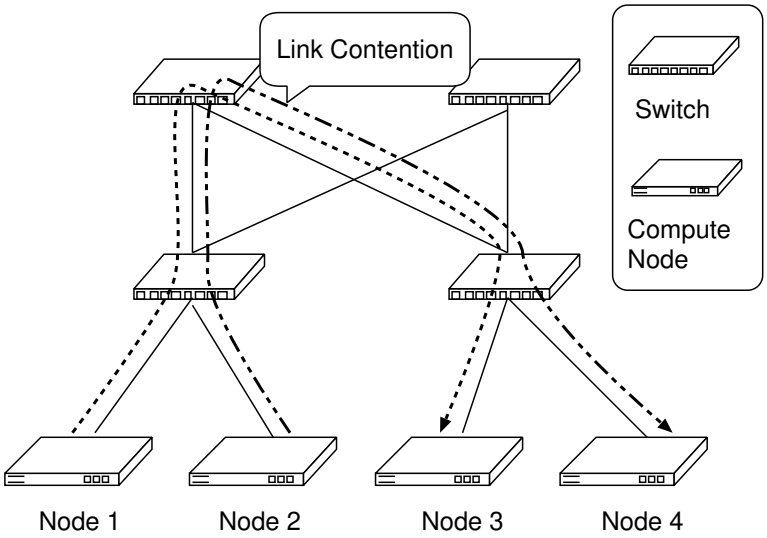


Figure 3.1: Link Contention on a Fat-tree

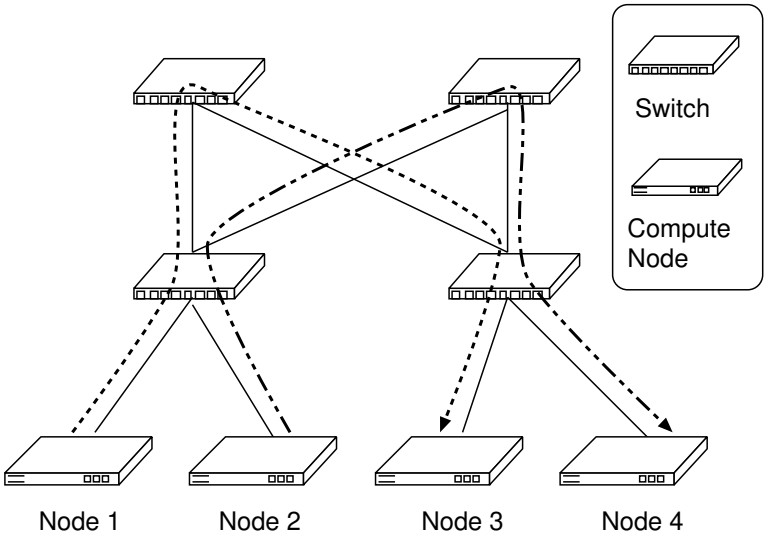


Figure 3.2: Load Balanced Routes on a Fat-tree

nodes, if multiple equal cost routes are available, ECMP selects one route from among them. The decision on which route to use is based on the header fields (*e.g.* source and destination addresses) of each packet. A hash function is applied to the header fields to generate the corresponding hash value for the header fields, where every value in the hash value space are evenly assigned to one of the equal cost routes.

InfiniBand [70] is a computer network communication link commonly used in the area of HPC and data centers. InfiniBand supports multiple routing methods. One of those methods is a min-hop routing algorithm, which calculates the minimum hop route between every compute node pair. If multiple minimum hop routes for a single compute node pair are available, the algorithm assigns a route so that usage among links is equalized.

The problem of these existing conventional load balancing mechanisms is that they are application-agnostic and never consider the communication pattern of MPI applications. In general, MPI applications cannot retrieve the usage information of the underlying network nor control it. Meanwhile, MPI communication in an MPI application usually shows a strong locality where each process communicates with a limited number of processes. This non-uniform communication pattern combined with application-agnostic network control causes unequal link usage that decreases available bandwidth between compute nodes. Ultimately, this decrease in available bandwidth can lead to the performance degradation of MPI applications.

From the observations above, an application-aware network control mechanism which recognizes the communication pattern needed by MPI collectives is essential. Also, the mechanism must effectively utilize the bandwidth of each link by distributing the traffic among redundant routes. Therefore, this research leverage Software-Defined Networking (SDN) to enable such dynamic control of packet flow depending on the communication patterns of MPI applications.

In particular, this research focuses on accelerating MPI_Allreduce. MPI_Allreduce is one of the most frequently used and time-consuming collective communication functions of MPI. In fact, an analysis of 100K jobs executed on a large-scale production HPC system has revealed that MPI_Allreduce accounts for 19.4% of the total core-hours spent in MPI, and is the most significant among all MPI collectives regarding both core-hours and number of calls [7]. This collective communication reduces values from all processes with an operator and broadcasts the result of the reduction to every process. More specifically, suppose there are n processes in a communicator and each process with rank

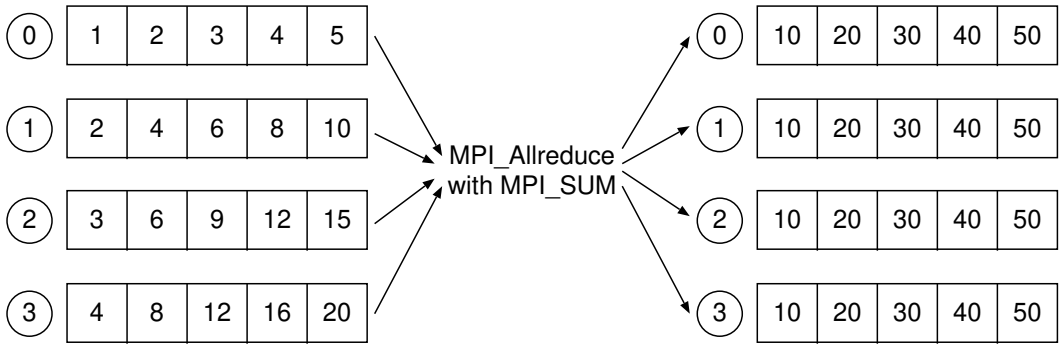


Figure 3.3: Effect of MPI_Allreduce

i ($0 \leq i < n$) has l values $x_i^0, x_i^1, \ldots, x_i^l$. After MPI_Allreduce is completed, all processes have values $y^0, y^1, \ldots, y^l$ where $y^j = x_0^j \oplus x_1^j \oplus \cdots \oplus x_{n-1}^j$ and $\oplus$ is the operator used for reduction. The operator can be any user-defined associative operator or one of the pre-defined operators such as sum, product, or maximum. Figure 3.3 shows an example where four processes each having five values call MPI_Allreduce with the sum operator.

One of the use cases of MPI_Allreduce is parallel Conjugate Gradient (CG) method. CG method is an iterative algorithm to solve a system of linear equations $Ax = b$ whose coefficient matrix A is positive-definite and symmetric. In the parallel CG method, a significant amount of time is spent in MPI_Allreduce to compute the inner product of vectors [38]. Another use case is parallel Stochastic Gradient Descent (SGD). SGD is a continuous optimization algorithm that minimizes an objective function f parameterized by w for a given set of input S. SGD incrementally updates w in the following way: $w^{t+1} = w^t - \eta \nabla f(w^t; z^t)$ where w^t is the parameter for the t-th iteration, z^t is an input data randomly sampled from S, and η is a small constant. In parallel SGD, the gradient $\nabla f(w^t; z^t)$ is computed in parallel by each process using different samples. Subsequently, MPI_Allreduce is used to compute the average of the individual gradients computed by all processes.

This research attempts to accelerate MPI_Allreduce on a cluster system with a fat-tree interconnect. The dynamic network controllability of SDN is integrated with MPI in order to mitigate link contention. The speedup of MPI_Allreduce based on the proposed framework is evaluated.

3.3 Proposal

This section presents the proposed framework for accelerating MPI collectives. First, the basic idea behind the proposed framework is outlined. After that, the design and implementation of the framework is described in detail.

3.3.1 Basic Idea

The motivation behind the proposed framework is to improve the inefficient communication in MPI in order to enhance the performance of MPI applications. The fact that MPI applications are unable to retrieve the usage of the underlying network results in a potential inefficiency in terms of communication. One of such inefficiencies is the inequality of link usage among links. Since the bandwidth of a link is limited, an inequality of link usage can cause congestion in heavy-loaded links. Therefore, this research focuses on the inequality of the link usage among the interconnect of a cluster system.

In this chapter, redundant routes in the interconnect are used to mitigate the inequality of link usage, based on an assumption that the cluster system has a fat-tree interconnect. Through this traffic distribution, contention in the links is expected to be alleviated, which as a result speeds up communication.

3.3.2 Design of Collective Acceleration Framework Using SDN

The proposed framework is composed of three modules. These three modules are SDN controller, LLDP (Link Layer Discovery Protocol) [48] daemon and SDN MPI library. They are deployed onto a cluster system as illustrated in Fig. 3.4. The SDN controller is designed to be deployed onto the management node of a cluster system. The management node is used for controlling the whole cluster system such as deploying jobs to the system. The LLDP daemon is designed to run in the background on all compute nodes. The SDN MPI library is a library that needs be statically linked to MPI applications at compile time.

Although each compute node can run one or more MPI processes, in this chapter, it is assumed that each compute node runs only a single MPI process. This is a reasonable assumption since many applications nowadays leverage *hybrid parallelism*, which combines distributed memory programming and shared memory programming. Under the hybrid parallelism model, the application starts a single MPI process per node

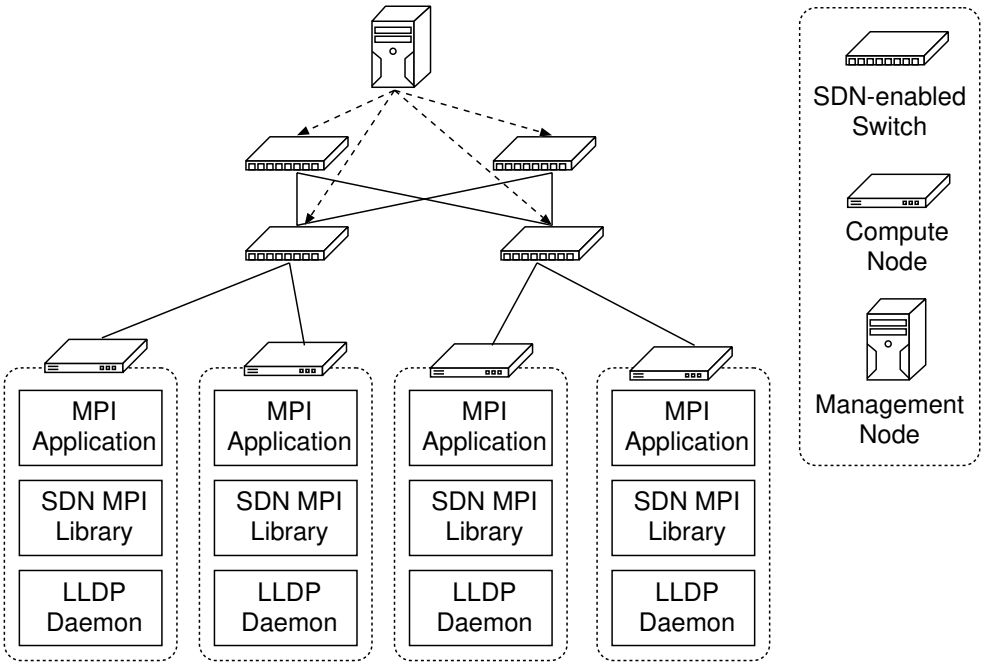


Figure 3.4: Placement of the Modules Composing SDN-enhanced MPI_Allreduce

and then spawns a thread for each socket or core on the node. After that, the application uses MPI for intra-node communication and threading frameworks such as OpenMP for intra-node communication.

The interaction among these modules is roughly divided into two phases: the initialization phase at the MPI application startup and the main phase at each MPI collective call. Figure 3.5 is a UML sequence diagram that illustrates how these modules cooperate with each other. MPI_Init is the MPI function that initializes the MPI execution environment, which must be called on the application startup. After MPI_Init finishes, all MPI processes notify their own IP address and MPI rank number to the SDN controller. This information obtained from MPI processes is held by the controller until the execution of the MPI application finishes.

When an MPI collective is called, the rank 0 process generates the communication pattern of the MPI collective. This communication pattern is a set of sender process and receiver process pairs during the MPI collective communication. After the communication pattern is generated, this set is sent to the SDN controller by the rank 0 process. As soon as the SDN controller receives the communication pattern, it generates a route for each sender-receiver pair. Subsequently, the SDN controller programs each SDN-enabled

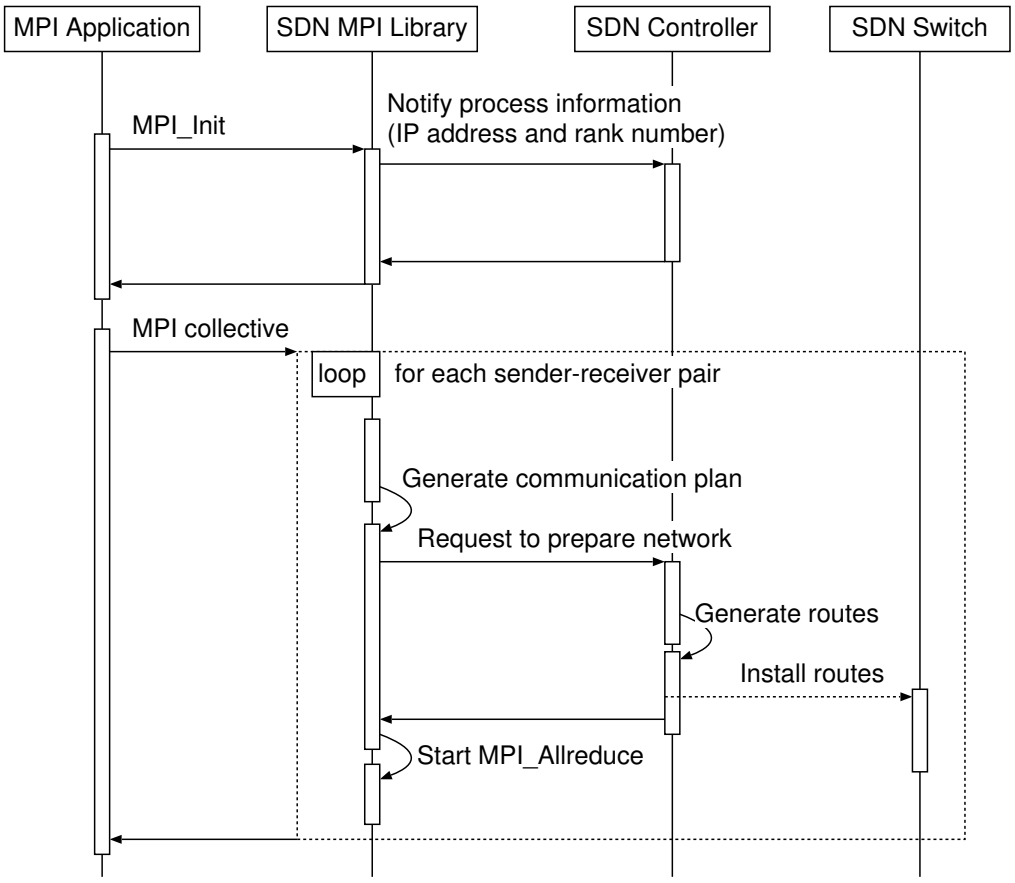


Figure 3.5: Sequence Diagram of the Proposed Framework

switch so that the MPI packets are routed along the pre-generated routes. After the entire communication pattern is processed, the MPI collective is called to start the actual data transfer and computation for the collective operation.

3.3.3 Implementation of Collective Acceleration Framework Using SDN

To realize the proposed framework, three modules that work as an integrated system has been developed: LLDP daemon, SDN controller and SDN MPI library. This section explains the implementation of these three modules in detail.

LLDP Daemon

Each compute node runs an LLDP daemon in the background. This daemon is designed to emit LLDP packets containing hardware information periodically, which are received by the SDN-enabled switches and used for topology discovery. Some LLDP daemon implementations already exist. However, a new, minimal daemon has been developed to easily add and tweak features so that it can cooperate with the other programs composing the whole system.

The developed daemon detects all available network interfaces installed on a computer and queries its interface index, MAC address and IP address. This information is packed into a single LLDP packet and sent out from each network interface periodically. The interval is set to one second in this prototypical implementation to speed up the topology discovery. However, it can be a longer period in practical systems so that its topology does not change frequently.

As described above, the LLDP daemon emits a few hundred byte long packets to the network every second. This traffic is considered to be small enough so that it does not cause serious side effects on the actual MPI process, for instance taking CPU time away from the application or consuming too much bandwidth that could interfere with the MPI communication. Generating such LLDP packets is also not a difficult task for a today's computer, so the impact to the application is considered to be ignorable.

SDN Controller

The SDN controller was developed on top of Trema [5], a framework designed for easily developing OpenFlow controllers in the Ruby language. It has the following four core functionalities:

1. Generating routes for MPI collectives to mitigate link contention and installing the generated routes to SDN-enabled switches
2. Detecting the topology and usage of the interconnect using LLDP
3. Responding to Address Resolution Protocol (ARP) requests from compute nodes to avoid broadcast storms
4. Routing of non-MPI traffic such as ICMP and SSH

The first functionality is topology detection. How detection is performed is shown in Fig. 3.6. The controller periodically requests every switch to send out an LLDP packet from each of their physical ports (step 1 in Fig. 3.6). This LLDP packet contains two kinds of information: datapath ID (a number that uniquely distinguishes the switches) and port number (port index where the packet is sent out). Moreover, all compute nodes also emit LLDP packets from the LLDP daemon described in the previous section. The controller is notified of a LLDP packet arrival at a switch. After that, it parses the packet to obtain the information on the packet's origin, and then examines whether the packet came from a compute node or an SDN-enabled switch. If the sender is a compute node, its MAC address and interface index is acquired. Otherwise if the sender is a switch, its datapath ID and port number is acquired (step 2). Using this information from its neighbors, an adjacency list is generated (step 3). From this adjacency list, a network topology graph is constructed, which is used in the route generation and routing. If the packet is from a compute node, the source MAC address and IP address are registered in a MAC/IP address translation table used in the ARP responding functionality.

The second functionality is replying to ARP requests. ARP requests are L2 broadcast packets and therefore cause a *broadcast storm* in a network that contains a cycle. Since this chapter targets a network that has redundant routes, the topology of the network is not a tree. Therefore, the network is always cyclic and affected by the broadcast storm problem. To eliminate broadcast storms, the controller instructs the switches to reply to

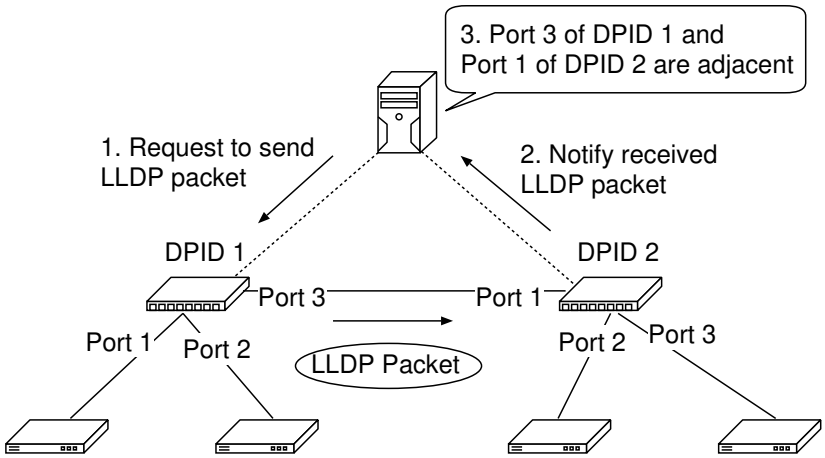


Figure 3.6: Topology Detection using LLDP

the received ARP requests on behalf of the compute node that has the corresponding IP address. The IP address that corresponds to the MAC address is obtained from the above-described MAC/IP address translation table.

The third functionality is route generation and installation for MPI collective communication. The route generation algorithm is implemented as a pluggable module so that different algorithms can be specifically tailored for each MPI collective.

Since this research focuses on accelerating MPI_Allreduce, a route generation algorithm targeting MPI_Allreduce has been designed and implemented. The goal of this route generation algorithm is to mitigate the interference between the packet flows generated by the underlying point-to-point communication of MPI_Allreduce. In particular, the proposed algorithm tries to generate routes so that the packet flows are evenly distributed among the redundant routes in the interconnect. In other words, the proposed algorithm aims to minimize the maximum number of packet flows sharing a link.

Note that this problem is a combinatorial optimization problem on a multi-commodity flow network and thus requires heavy computation to find the optimal solution. Meanwhile, the SDN controller needs to perform the route generation every time an MPI collective is called as described in Section 3.3.2. Therefore, the proposed algorithm is designed to be a heuristic algorithm based on a greedy strategy that quickly finds an approximate solution rather than an optimal solution.

The basic idea behind this heuristic algorithm is to assign a route to each point-to-point communication iteratively. At each iteration, the algorithm selects a pair of a sender and

a receiver and finds the least utilized route between them. This is achieved by considering a weighted graph where the weight of a link is equal to the total number of packet flows going through the link, and then finding the shortest route between the sender process and receiver process in this graph. Here, the *Dijkstra* algorithm is used to find the shortest route because of its speed and simplicity. After a route is assigned, the weight of each link along the generated route is incremented. This procedure is repeated for all sender-receiver pairs.

Algorithm 3 is a pseudo-code for the algorithm. First, an empty array *routes* is initialized (line 1 in Algorithm 3). After that, the route search is performed for each sender-receiver pair (line 4–7). The resulting route is added to *routes* (line 5) and the link weight (which is the number of total routes that use that link) of each link is incremented (line 6–7). After all routes are generated, these routes are installed to the SDN-enabled switches.

Algorithm 3: Pseudocode of Route Generation

```
1 routes ← empty array;
2 nodes ← nodes in the topology graph;
3 links ← links in the topology graph;
4 foreach (sender, receiver) ∈ sender-receiver pairs do
5     route ← dijkstra(nodes, links, sender, receiver);
6     foreach link ∈ route do
7         Increment weight of link;
```

The fourth functionality is route generation and installation for non-MPI traffic. For non-MPI traffic such as ICMP and SSH packets, the SDN controller generates the minimum hop routes between compute nodes and installs them on demand.

SDN MPI Library

An MPI application that wants to use the proposed framework must be linked with the SDN MPI library. This library contains two classes of functions: SDN_MPI_Init, which is an initialization function for the library, and MPI collective functions prefixed with SDN_MPI_, which replace the conventional MPI collectives with their SDN-enhanced versions.

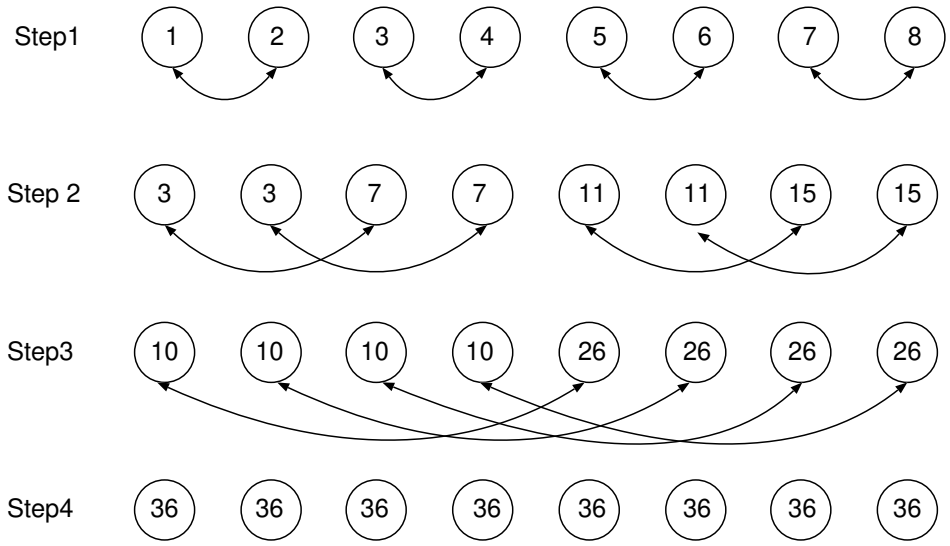


Figure 3.7: Recursive Doubling Algorithm

The application is required to call SDN_MPI_Init when it launches. This function opens a TCP connection with the SDN controller and notifies the IP address and MPI rank number of the process that has called itself.

SDN_MPI collectives are called by the application when it needs to perform collective communication. Each SDN_MPI collective generates the communication pattern (set of sender process and receiver process pairs during the collective communication) for the MPI application and sends the pattern to the SDN controller.

Several algorithms to realize the Allreduce operation have been proposed [11, 38, 63, 66]. This research focuses on *recursive doubling* [63], since it requires more inter-node communication compared with other algorithms, which means more room for optimization in terms of communication. Figure 3.7 illustrates how the recursive doubling algorithm works. The recursive doubling algorithm requires $\log p$ communication steps where p denotes the number of processes. For explanatory purposes, the *distance* between two MPI processes is defined as the absolute difference of their rank numbers here. In the first step, processes that are one distance apart exchange their data and perform the reduction operation between the data that the process has originally held and with the just exchanged data. In the second step, processes that are 2 distance apart exchange their data, and in the i-th step, processes that are 2^{i-1} distance apart exchange their data. The SDN MPI library memorizes all process pairs that have to communicate and exchange data by following each step of the recursive doubling algorithm. For each of those pairs, the library notifies

the SDN controller to prepare each route.

3.4 Evaluation

3.4.1 Experimental Environment

An experiment was conducted to compare the execution time of MPI_Allreduce accelerated with the proposed framework against conventional MPI_Allreduce. The experimental environment is illustrated in Fig. 3.8. This experiment was performed on a real cluster system consisting of 28 compute nodes and 6 SDN-enabled switches, which formed a two-tier fat-tree topology. The compute nodes and SDN-enabled switches were all connected through Gigabit Ethernet links; hence the interconnect was oversubscribed.

In addition to the network connecting the compute node and switches, another network was prepared for control and management. This network connects compute nodes, SDN-enabled switches and the SDN controller. The interaction between the SDN controller and SDN switches is performed via OpenFlow protocol with this management network. The compute node that runs MPI's rank 0 process and the SDN controller also communicates with this network. Other compute nodes were connected to the management network as well, but those connections were not used in this experiment.

CentOS 6.4 was installed on all computers including the compute nodes and SDN controller. The SDN controller was developed using a SDN controller framework Trema [5] 0.4.6 and Ruby 1.9.3. The SDN MPI Library and the benchmark application were written in C and compiled with gcc 4.4.7. As a representative of a conventional MPI, Open MPI [64] 1.5.4 was used.

3.4.2 Measurement Result

A micro-benchmark that repeats MPI_Allreduce 20 times and measures the average execution time of the function was used for comparing the execution time of the proposed MPI_Allreduce with its Open MPI counterpart.

Figure 3.9 shows the measurement results using 8 nodes, where the horizontal axis indicates the message size and the vertical axis shows the average time taken to execute MPI_Allreduce. The solid line and dashed line represent the execution time of the

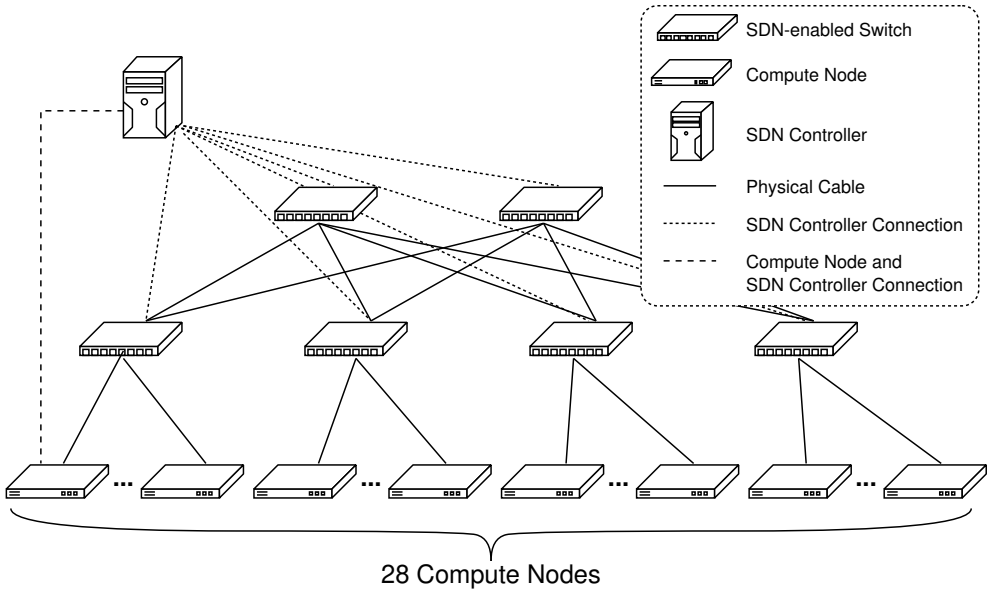


Figure 3.8: Experimental Environment

proposed framework and Open MPI implementation, respectively. Figure 3.10 shows the speedup of MPI_Allreduce accelerated using the proposed framework in comparison with the Open MPI implementation. The maximal speedup was 41%. Figure 3.11 shows the result in the case of using 16 nodes. Figure 3.12 indicates the speedup of the proposed MPI_Allreduce. It shows that the proposed framework succeeded to realize a 56% speedup at maximum.

Both Figs. 3.10 and 3.12 clearly show that MPI_Allreduce accelerated with the proposed framework is consistently faster than the Open MPI implementation. However, some fluctuation of the speedup is also observed. This fluctuation of performance is considered to be caused by the non-deterministic aspect of the network. For example, queueing and scheduling of packets at compute nodes and switches, TCP congestion control, and OS noise heavily affect the time it takes for each packet to travel through the network. Such variation in the latency of each point-to-point communication significantly impacts the execution time of a collective communication. A larger number of benchmark runs is expected to exhibit a constant speedup ratio.

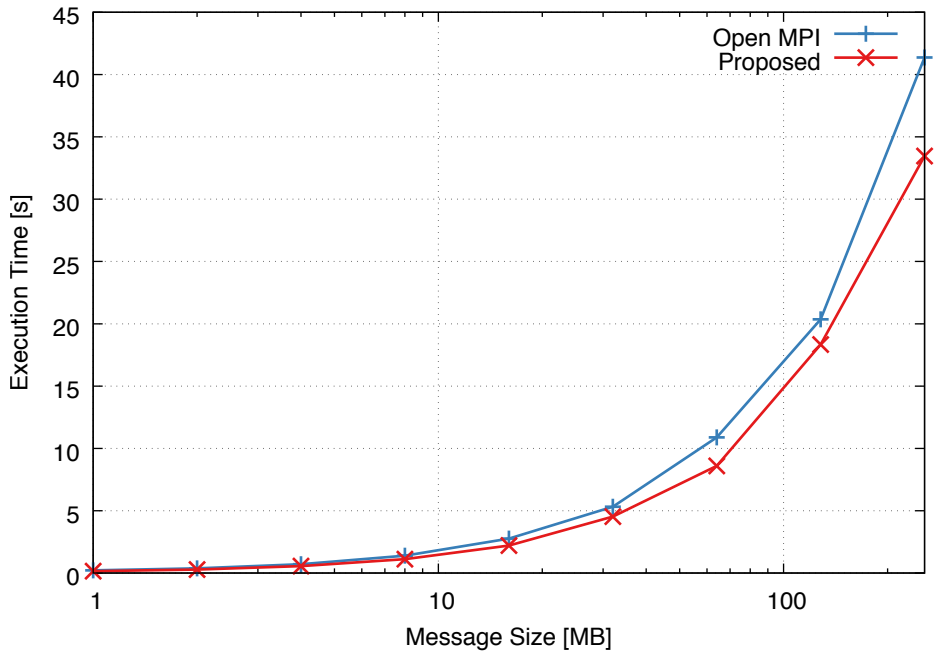


Figure 3.9: Comparison of Execution Time of MPI_Allreduce (8 Compute Nodes)

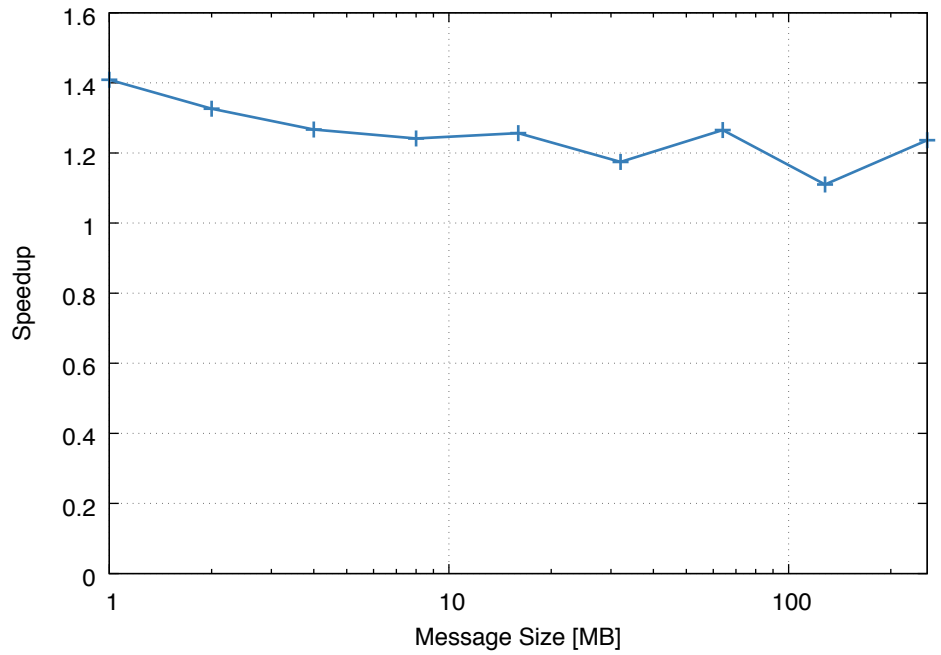


Figure 3.10: Speedup of Proposed MPI_Allreduce (8 Compute Nodes)

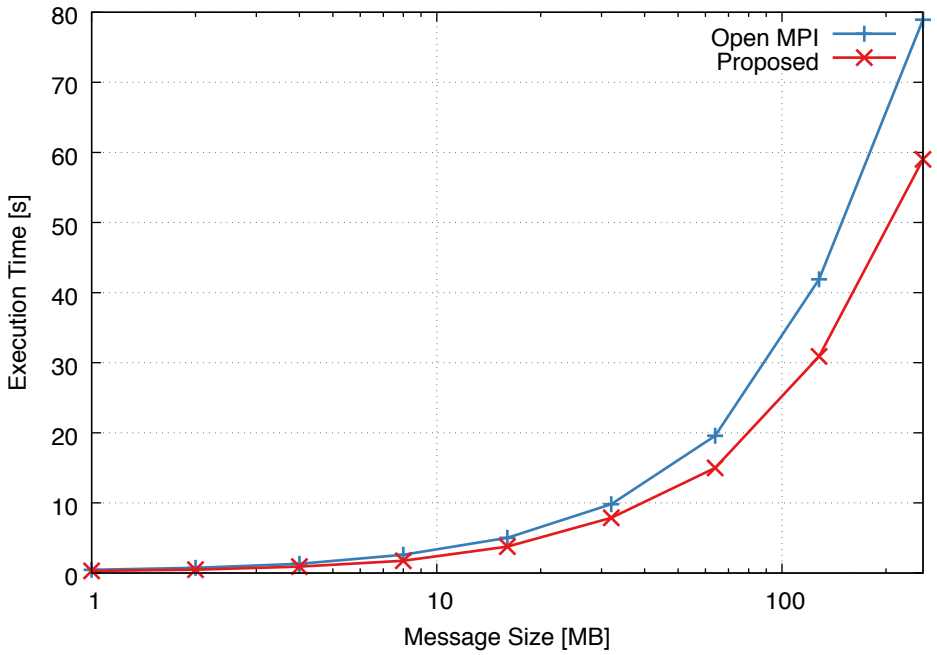


Figure 3.11: Comparison of Execution Time of MPI_Allreduce (16 Compute Nodes)

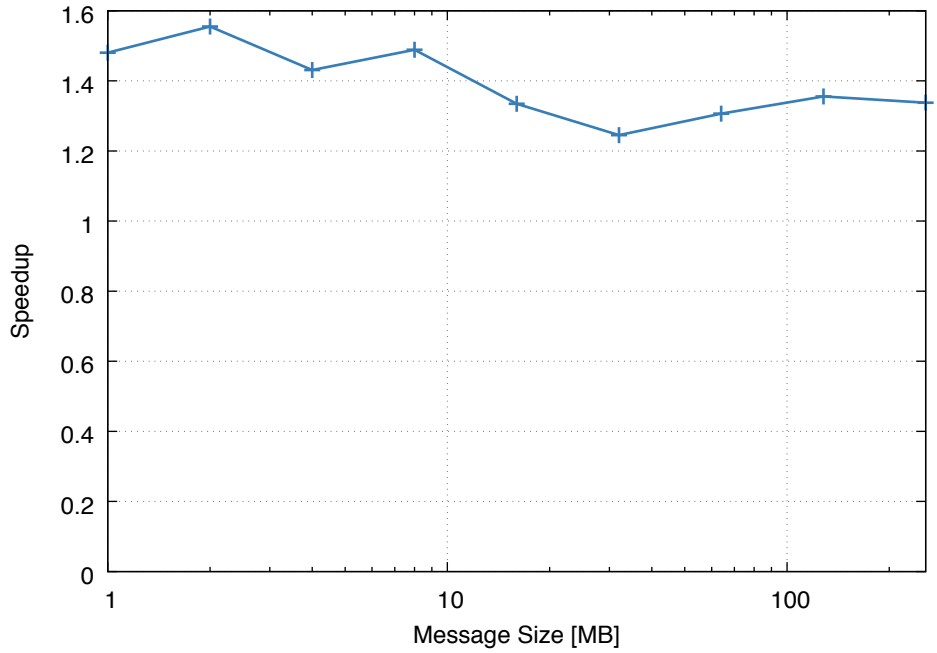


Figure 3.12: Speedup of Proposed MPI_Allreduce (16 Compute Nodes)

3.5 Related Work

There have been many research works related to MPI. Since MPI is merely a specification for standard APIs for parallel programming, several algorithms for collective operations have been proposed and implemented targeting several network technology. As a representative example of such works, MVAPICH [59] can be raised. MVAPICH is an MPI implementation targeting InfiniBand, which most of high-performance computing systems ranked in Top500 have adopted. Sur *et al.* [43] designed an MPI library by leveraging novel InfiniBand-offered features. They explored new architectures from a system point of view and new programming paradigms from an application point of view to keep scaling out applications on more powerful computing systems. Jiuxing *et al.* [65] also investigated an MPI communication protocol focusing on RDMA operations in InfiniBand. The approach of this research has some points in common in terms that this research also aims to benefit from the features of the underlying network. However, this research leverages Software-Defined Networking instead of InfiniBand from the purpose of investigating the feasibility of dynamic control of network from an application point of view.

Researchers have attempted to elaborate algorithms for MPI collective operations. Optimized algorithms have been proposed for MPI_Alltoall [71], MPI_Reduce_scatter [72], MPI_Reduce and MPI_Allreduce. Most of the optimized algorithms are specialized either for latency or throughput, so switching multiple algorithms depending on the message size or process number makes the MPI implementation behave faster for various message sizes and process numbers [63].

Another approach is to offload the communication or computation of collective communication operations to hardware. For example, the *K-computer* [44] has a hardware module called *Tofu Barrier Interface* [36] on all nodes. This module executes Barrier, Broadcast, Reduce and Allreduce collective operations in hardware instead of software. However, this approach does not mitigate the congestion on links.

Past works including existing MPI implementations have been successful in switching between multiple algorithms depending on message size, node number, *etc.* to accelerate collective communication in MPI. Using such a mechanism, some estimation of the threshold value for parameters like message size, node number, *etc.* is essential. Pješivac-Grbović *et al.* [57] compared several parallel communication models that are

frequently used for dynamically estimating the threshold values. In contrast, the current implementation of the proposed framework always uses recursive doubling for executing MPI_Allreduce. Therefore, the proposed framework has a disadvantage in the cases where small data size is treated on MPI_Allreduce and thus the latency is more respected than the bandwidth. For this disadvantage, automatic switching of algorithms that leverages estimation models is planned to be introduced.

The Fabric Collective Accelerator (FCA) [2] is a product by Mellanox Technologies with the target of accelerating collective communication on clusters with InfiniBand interconnect. FCA accelerates collective communication by offloading computations to an InfiniBand Host Channel Adapter (HCA). It also optimizes communication flow according to job and topology. FCA optimizes collective tree and rank placement to control communication flow. In contrast, the proposed framework is capable of adaptively reconfiguring the network itself, which is more flexible. FCA also requires InfiniBand hardware, but this research focuses on a commodity Ethernet network.

Furthermore, there have been many research reports focusing on adaptive use of networks for high-performance computing. Geoffray *et al.* [52] proposed an adaptive routing method on Myrinet and the above-mentioned literature [65] explored the adaptive use of multiple independent networks on InfiniBand. This research also aims for a dynamic use of the underlying interconnection network, but differs in the fact that this research attempted to use a different interconnection network.

3.6 Conclusion

This chapter has attempted to reduce the execution time of MPI collectives by dynamically controlling the packet flow in the interconnect using Software-Defined Networking (SDN). By using SDN, this research has proposed a novel framework for accelerating collectives that can effectively make use of redundant routes by having SDN interact with the communication pattern of collectives. A system of three modules cooperating together has been designed and implemented to realize the proposed framework. The evaluation conducted in this chapter showed that the proposed framework speeds up the execution time of MPI_Allreduce for 56% at most compared with a conventional implementation of MPI_Allreduce. This result confirms the superiority of the proposed framework over conventional methods.

Several issues to tackle have remained for the realization of practical and useful collective acceleration framework leveraging SDN. The first issue is the support for multiple processes on a single compute node. On modern computing platforms where multiple cores are implemented in a single compute node, the assumption set in this preliminary stage of this research is neither practical nor realistic. Therefore, intra-node communication for pure MPI jobs needs to be considered by integrating kernel-assisted communication such as KNEM [34] or Cross Memory Attach (CMA) with the proposed method. Also, computing paradigms such as the hybrid use of OpenMP with MPI must be considered for enhancing practicality. Also, other implementations of MPI such as MVAPICH [59] are essential for investigating the practicality and usefulness of SDN.

The second issue regards the necessity of additional experiments on a larger scale environment. This chapter has verified the feasibility and possibility of the proposed framework through the experiments using a simple prototypical implementation. However, because OpenFlow switches were expensive and thus only a small-scale cluster was available, only a limited number of experiments in a small cluster environment could be conducted. Therefore, further experiments on a larger scale environment are essential for the evaluation of future scalability.

The third issue is the scalability issue caused by the SDN controller. Since the current implementation requires communication between MPI processes and SDN controller and route generation for each MPI communication request, the SDN controller might become a scalability bottleneck on larger environments. Therefore, the IP address and MPI rank number for each process need be cached in the SDN controller. Furthermore, the current implementation needs to be enhanced so that only the root process interacts with the SDN controller and conveys information of all participating processes in the collective communication.

4 Coordination Mechanism of Communication and Computation

4.1 Introduction

Recent scientific research has been taking a major advantage of computational analysis and simulation. Sustained proliferation in the volume of data generated by scientific experiments has led to a rise in the importance of data-intensive computing. For example, approximately 15 PB of experimental data is annually generated and processed at the Large Hadron Collider (LHC), an experimental facility for high energy physics [41].

Today, in general, data-intensive computations are performed on high-performance computer clusters. A computer cluster is composed of a set of compute nodes connected to a high-performance network, usually referred to as an *interconnect*. Applications designed to run on computer clusters are based on a parallel distributed processing model. In this processing model, a large computation is decomposed into smaller fractions of computation and then performed by processes running in parallel. These processes communicate with each other for data exchange and synchronization. For this reason, the inter-node communication performance among processes can significantly impact the total performance of data-intensive applications. Recent advancements of high performance computing has heavily relied upon the high degree of parallelism rather than the improvement of CPU clock speed. Consequently, the total number of processes and compute nodes involved in a computation has kept increasing. As a result, communication between distributed processes is becoming the principal bottleneck of data-intensive applications.

Each application running on a computer cluster has a distinct pattern of communication among processes [47]. These communication patterns are difficult to predict precisely in prior to the execution of the application. Furthermore, most of the current interconnects

available have adopted static network control and thus they are unable to adaptively reconfigure themselves to match requirements from applications. In fact, in InfiniBand [70], which is a currently dominant interconnect technology, the forwarding tables on switches are usually pre-configured and remain unchanged until hardware failure or topology change occurs.

Furthermore, current interconnects are designed to be over-provisioned in order to satisfy the communication performance requirements from various applications with diverse communication patterns. Such over-provisioned interconnects are designed and provided with sufficient network resources (*e.g.* bandwidth) to minimize the overload of interconnect such as congestion.

However, the recent scale-out in number of compute nodes has revealed two potential shortcomings of over-provisioned designs. First, the cost for building interconnects has become increasingly higher, which makes it difficult to implement over-provisioned designs. This increased cost is because of the scale and complexity of interconnects that grow superlinearly as the number of compute nodes increases. The second shortcoming is the under-utilization of interconnects. A discrepancy between the performance characteristics of the over-provisioned interconnect and the aggregated network requirements of the applications may cause some portion of the interconnect not being fully utilized.

Based on these considerations, a novel cluster architecture which dynamically controls the packet flow in the interconnect based on the communication pattern of the application is considered to alleviate the aforementioned two shortcomings derived from the conventional over-provisioned designs. For the reason, Software-Defined Networking enhanced Message Passing Interface (*SDN-enhanced MPI*), which is an unconventional MPI framework that incorporates the flexible network controllability brought by SDN into interconnects, was proposed in our past research. Furthermore, past research towards SDN-enhanced MPI has demonstrated that the acceleration of collective MPI communication is feasible.

However, a technical challenge remains in this research, that is, applying the research achievements to real-world MPI applications. The preliminary stage of this research has mainly focused on verifying the feasibility of the idea by investigating whether individual MPI collective communications could be accelerated or not. Meanwhile, real-world applications commonly call multiple MPI collectives during their execution. Therefore, how MPI communication accelerated with SDN could be synchronized with the execution

of an MPI application remains a question that requires a new technical innovation.

To this end, this research proposes *UnisonFlow*, a mechanism for SDN-enhanced MPI to perform network control in synchronization with the execution of an MPI application, based on the strategy shown in [20]. The synchronization does not incur a large overhead so it avoids performance degradation of the applications. Furthermore, the proposed mechanism is designed to work on actual hardware OpenFlow switches, and is not limited to software switches or specialized hardware.

The main contributions of this chapter are summarized as follows:

- UnisonFlow, a software-defined coordination mechanism of network control and execution of an MPI application is proposed.
- A low-overhead implementation of the proposed concept that works on actual hardware OpenFlow switches is presented.
- An experiment is carried out to verify whether the interconnect control is successfully performed in synchronization with the execution of an application.
- A performance measurement of point-to-point communication is conducted to evaluate the overhead incurred by the proposed mechanism.

The remainder of this chapter is organized as follows. Section 4.2 introduces SDN-enhanced MPI and its key technologies. Subsequently, the challenge to realize SDN-enhanced MPI is derived. Section 4.3 describes the proposed mechanism and its implementation. Section 4.4 shows the result of the experiments conducted to demonstrate the feasibility of the proposal. Section 4.5 reviews related literature and clarifies the contributions of this chapter. Finally, Section 4.6 discusses future issues to be tackled and concludes this chapter.

4.2 Research Objective

This section first briefly describes the two key technologies of SDN-enhanced MPI: the Message Passing Interface (MPI) and Software Defined Networking (SDN). After outlining the current development status of SDN-enhanced MPI, the central challenge in realizing a practical SDN-enhanced MPI is clarified.

4.2.1 SDN-enhanced MPI

The basic idea of SDN-enhanced MPI is to incorporate the flexible network controllability of SDN into MPI. As described in Section 1.1.4, MPI mainly focuses on hiding the complexity of the underlying network architecture. Therefore, MPI does not provide any functionality for explicitly controlling the network. The integration of SDN into MPI could complement such lack of a network control feature in MPI and allow MPI to optimize the packet flow in the network in accordance with the communication pattern of applications.

At the time of writing this dissertation, the above described basic idea has been applied and tested on to two collective MPI primitives, MPI_Bcast and MPI_Allreduce as proof of concept. Experiments conducted on a real computer cluster comprising bare metal servers and hardware OpenFlow switches have demonstrated that the execution time of these primitives has been successfully reduced [23, 30]. SDN-enhanced MPI_Bcast [23] accelerates MPI_Bcast by utilizing the hardware multicast functionality of OpenFlow switches. SDN-enhanced MPI_Allreduce [30] dynamically reconfigures the path allocation based on the communication pattern of MPI_Allreduce so that congestion in links is minimized.

4.2.2 Central Challenge of SDN-enhanced MPI

The central challenge in realizing a practical SDN-enhanced MPI lies in a coordination mechanism between the application and network control. Although the previous works on SDN-enhanced MPI have shown the feasibility of accelerating individual primitives as described in Section 4.2.1, actual MPI applications have not yet taken the advantage of network programmability brought by SDN, since each of the distinct network control algorithms designed for an MPI primitive cannot be activated along with the execution of an MPI application. In other words, no mechanism exists that conveys the type and option of the MPI primitive being executed at the moment by an application to the network controller in charge of acceleration of the corresponding primitive.

This chapter aims at realizing a software-defined coordination mechanism to perform network control in synchronization with the execution of an application. Furthermore, the following technical requirements must be fulfilled by the mechanism:

- *Low overhead*: The overhead incurred by the proposed coordination mechanism

should not degrade the communication performance of MPI, since the final goal is to improve the total performance of the MPI application.

- *Interoperability with hardware OpenFlow switches*: This research places emphasis on developing a practical implementation that works on computer clusters. Therefore, the mechanism should work on actual hardware OpenFlow switches, and should not be limited to software switches or specialized hardware.

- *Compatibility with existing MPI library*: To mitigate the cost to port the existent MPI applications on SDN-enhanced MPI, the existent MPI applications should work on SDN-enhanced MPI without the source code being modified or recompiled. Compatibility with existing MPI implementations is essential for the portability of applications.

4.3 Proposal

4.3.1 Basic Idea

The basic idea of UnisonFlow is to embed MPI context information as a *tag* into each packet released through the MPI library and handle packets based on their tags in switches. The tag is stored in the header field of each packet. In this dissertation, MPI context information is defined as a collection of application-aware data which identifies an communication of an MPI communication primitive. Specifically, an MPI primitive type, source/destination rank and communicator constitute the MPI context information.

A straightforward approach to realize application-aware network control is to enhance the packet processing feature of OpenFlow switches in a way that switches can read the application-layer information from packets and then make decisions based on that information. However, this approach requires significant alteration to the switch hardware itself and the OpenFlow protocol, because packet processing on switches is mostly performed on fixed dedicated hardware components. The proposed mechanism stores the application-layer information into a header field of packets so that OpenFlow switches can perform application-aware packet flow control.

Technologically, the tag is embedded into the destination MAC address field of the packet header field. The location of the destination MAC address field in a packet and

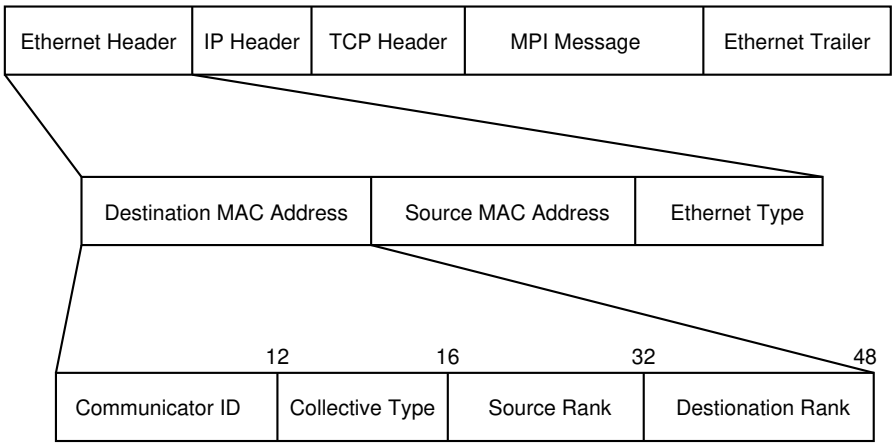


Figure 4.1: Tag Information Embedded in a Packet

the binary layout of a tag are shown in Fig. 4.1. Two main reasons exist for using the destination MAC address header field. The first reason is that the MAC address is defined as one of the header fields that can be used as a matching condition in OpenFlow. In other words, there is no need to extend or modify existing OpenFlow switches to support this header field. The second reason is explained from the advantage in the number of installable flow entries. Although there are header fields other than the destination MAC address that can be used as a matching condition in OpenFlow, switches are typically equipped with a special hardware dedicated for L2 header field lookups. As a result, more flow entries that include only L2 header fields can be stored than the flow entries with other header fields.

4.3.2 Architecture

Overview

Figure 4.2 illustrates an overview of UnisonFlow. In this research, it is assumed that a computer cluster executes a single MPI application because the target of this research is the acceleration of inter-node communication in MPI. The operating system of compute nodes is assumed to be Linux.

Three major software modules that constitute this architecture (bold rectangles in Fig. 4.2) have been developed. The first module is the *Interconnect Controller*, which is basically an OpenFlow controller responsible for installing flow entries into OpenFlow

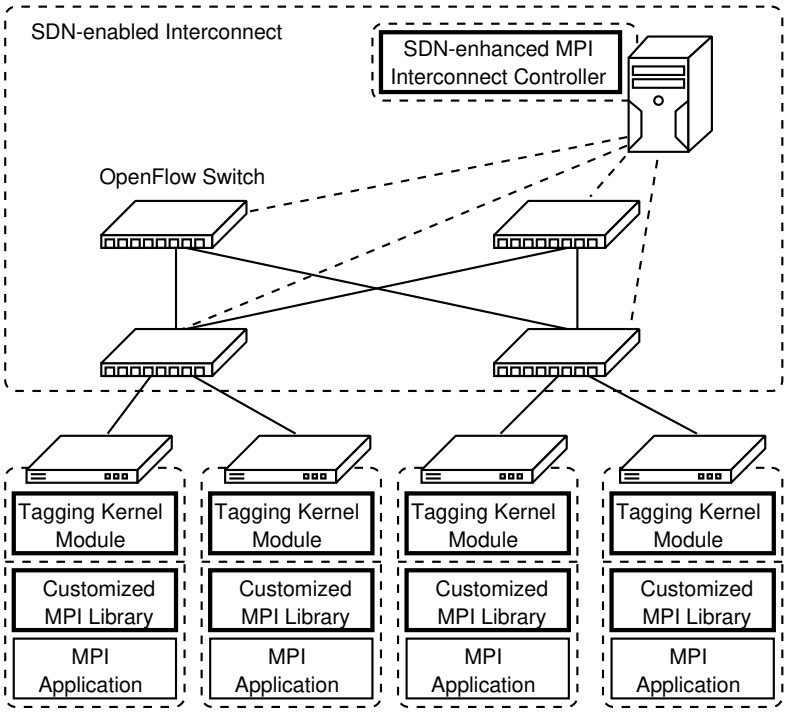


Figure 4.2: Overall Architecture of UnisonFlow

switches. The interconnect controller was developed based on the Ryu SDN controller framework [29]. The second module is the *Tagging Kernel Module*. It was designed and developed to reside in the kernel space of each compute node. The role of the tagging kernel module is to extract MPI context information from each packet emitted by the MPI library, encode this context information as a tag and then apply it to the packet. The third module is the *Customized MPI Library,* which was designed and implemented to be dynamically linked with the MPI application. MPICH [69], an implementation of MPI library, was extended so that it meets our needs. Specifically, it was enhanced to communicate with the tagging kernel module and to send active connection information to the kernel module.

Intra-node architecture

On each compute node, the tagging kernel module and MPI library is deployed to work together to embed MPI context information as a tag into each packet. The kernel module performs the actual tagging procedure, whereas the MPI library provides the kernel module with complementary information used for filtering out non-MPI traffic.

As described in Section 4.3.1, UnisonFlow exploits the destination MAC address field of a packet as a place to store the corresponding tag. To implement this embedding of a tag to the destination mac address field, a functional component that dynamically rewrites MAC address fields of packets is essential.

Three potential technical solutions have been considered for implementing the functional component on the Linux kernel: (1) *ebtables*, (2) *raw socket* and (3) *protocol handler* [35]. Ebtables is a widely adopted L2 packet filter implemented on top of the netfilter framework. It mainly features L2 packet filtering and Network Address Translation (NAT). Raw sockets are special type of sockets that give user space programs access to the whole packet including protocol headers. TCP/UDP sockets only allows user space programs to read or write TCP/UDP payloads, whereas raw sockets allows programs to read and write TCP/UDP, IP and Ethernet protocol headers. In exchange for the high flexibility, the user space program has the full responsibility to handle the network protocol correctly. Protocol handlers are used to implement new network protocols in the Linux kernel. The network stack of Linux kernel is designed to be extensible so that new network protocols can be added relatively easily. New protocols can be implemented by protocol handlers, which are essentially callback functions that are invoked when a packet is sent or received. When implemented in a loadable kernel module, protocol handlers can be added without recompiling the kernel.

Out of these potential solutions, the protocol handler has been adopted because it can achieve both flexibility in rewriting of the packets depending on their payload and minimal alteration to the MPI library. As previously described, ebtables has a MAC NAT feature. However, it has a limitation where the MAC addresses can only be translated to pre-configured addresses. On the other hand, the use of a raw socket results in an extensive modification of the MPI library, since it requires the MPI library to handle the TCP/IP stack. In contrast to these two methods, the use of the protocol handler facilitates the interception of packets in the network stack of the kernel and arbitrary modifications to those packets. For this reason, re-implementing another network stack can be avoided by utilizing the existing network stack of the kernel. Moreover, the whole packet including header and payload can be read and written by the protocol handler for dynamically rewriting the MAC address fields of packets.

Figure 4.3 illustrates how MPI packets are processed on a compute node. The solid arrows represent packet flows generated by an MPI application. The dashed arrow

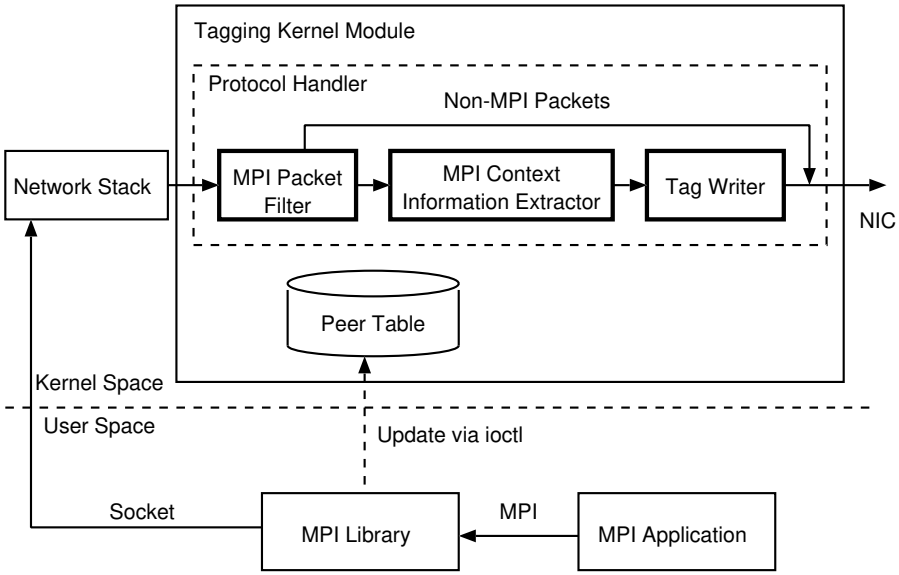


Figure 4.3: Intra-node Packet Flow

represents interaction between software modules.

Once the tagging kernel module is loaded into the kernel space at the boot time of the Linux operating system, the kernel module registers its own protocol handler to the kernel using the `dev_add_pack` API. This protocol handler is called every time a packet is sent out from the network stack to the Network Interface Card (NIC). Intercepted packets sequentially undergo three major phases of packet processing, which are performed by the following three components (bold rectangles in Fig. 4.3), respectively:

1. *MPI packet filter*: Packets generated by SSH, remote file systems, and any other programs other than MPI are immediately forwarded to the NIC. To investigate whether a packet originates from MPI or not, this component looks up the *peer table* maintained by the tagging kernel module and verifies if the packet is a part of the TCP connections opened by the MPI library. The peer table is designed as a hash table of all TCP connections to other processes opened by MPI. The 4-tuple (source IP, destination IP, source port and destination port) of each packet is used to identify a TCP connection.

2. *MPI context information extractor*: This component extracts the MPI context information from packets by reading and parsing their message envelope. The message envelope is essentially a header prepended to every MPI message by the

MPI library for identification. Although the message envelope is prescribed in the MPI specification [19], its actual binary layout is implementation-dependent.

3. *Tag writer*: This component encodes the context information extracted in the previous phase as a virtual MAC address and writes it into the packet. The virtual MAC address is generated by packing the components of MPI context information into the binary format shown in Fig. 4.1. Technically, the MAC addresses of packets can be modified by simply overwriting the specific position of the `sk_buff` structure, which is the internal representation of network packets in the kernel.

As described, the tagging kernel module maintains the peer table to keep track of all connections opened by the locally-running MPI process to other MPI processes running on remote compute nodes. In order to update the content of the peer table in accordance with the internal information of the MPI library, the MPI library has been enhanced to provide this information to the kernel module. As the communication channel between the MPI library and kernel module, the `ioctl` system call has been leveraged. These modifications have been made so that functional compatibility with the original MPI library is guaranteed.

Inter-node architecture

Switches composing the interconnect forward packets based on their tag value. These forwarding rules are stored in the form of flow entries and managed by the centralized interconnect controller.

The decision on how a packet is forwarded is made by the *MPI primitive module*, which is a pluggable software component integrated into the interconnect controller. A unified interface between the MPI primitive module and the interconnect controller is defined for simplified development and integration of primitive modules. Each MPI primitive module is expected to be designed dedicatedly for a single type of MPI primitive.

Figure 4.4 illustrates an example of the packet flow between two remote compute nodes. When the interconnect controller receives a packet-in message caused by an unmatched packet (step 1), the controller decodes the tag embedded in the packet and extracts the MPI context information (step 2). After that, the responsible MPI primitive module is invoked with the context information as its input (step 3). The MPI primitive module determines how a set of packets carrying the same context information should be treated.

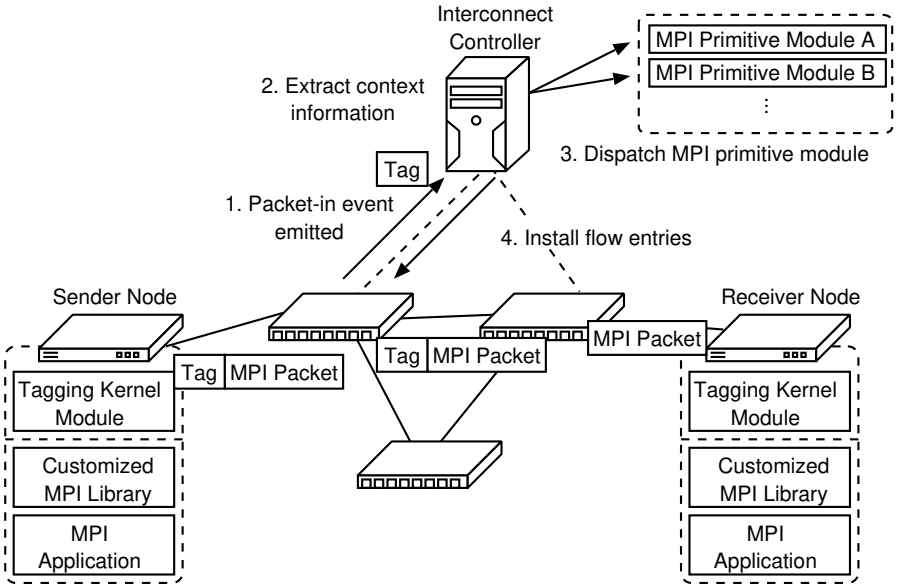


Figure 4.4: Inter-node Packet Flow

Based on this decision, flow entries are generated and then installed to relevant switches (step 4).

Note that NICs drop incoming packets whose destination addresses are not the address of NICs unless they are put into promiscuous mode. Therefore, the destination MAC address of tagged packets needs to be restored to the true MAC address of its receiver node. This restoration is achieved by appending an action for changing the MAC address field to the flow entry installed on the switch adjacent to the receiver node.

4.4 Evaluation

Two experiments are conducted to examine the feasibility of UnisonFlow. In the first experiment, the control of the interconnect is investigated in terms of whether it is properly synchronized with the execution of the application. In the second experiment, the overhead imposed by UnisonFlow is evaluated.

4.4.1 Experimental Environment

Both of the two experiments were conducted on the SDN-enabled computer cluster shown in Fig. 4.5. For the topology of the interconnect, a two-tier fat-tree composed of

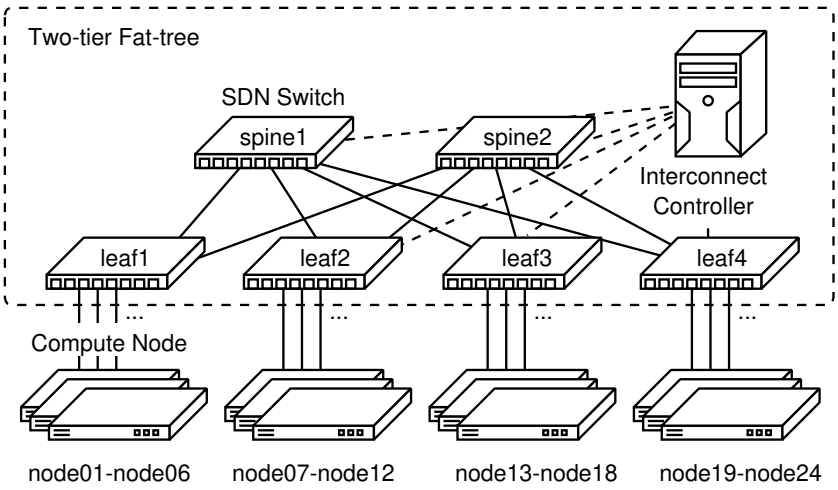


Figure 4.5: Overview of the Experimental Environment

six switches was adopted because a fat-tree is one of the most widely used topologies for today's cluster systems. Note that each of the two physical switches was divided to three logical switches due to the limited number of available OpenFlow switches in our institution. In the following discussion, the two upper layer switches are referred to as spine1 and spine2, whereas the four lower layer switches are referred to as leaf1, leaf2, leaf3 and leaf4, respectively. Spine switches and leaf switches were connected on 4 Gbps links, each of which was an aggregated link of four GbE links. Six compute nodes were connected to a leaf switch; that is, 24 compute nodes in total. These compute nodes are hereinafter referred to as node01 to node24. Leaf switches and compute nodes were interconnected with Gigabit Ethernet. A management node accommodating the interconnect controller was also prepared.

For SDN switches, NEC ProgrammableFlow PF5240 has been adopted. The compute node was a SGI Rackable Half-Depth Server C1001 equipped with the hardware and software as shown in Table 4.1.

4.4.2 Verification of Coordination Mechanism

The first experiment was conducted to verify whether the dynamic control of packet flows on the interconnect was performed in synchronization with the execution of the application. To verify the synchronization between interconnect control and execution of the application, an MPI application which sequentially executes two different MPI

Table 4.1: Specifications of Compute Nodes

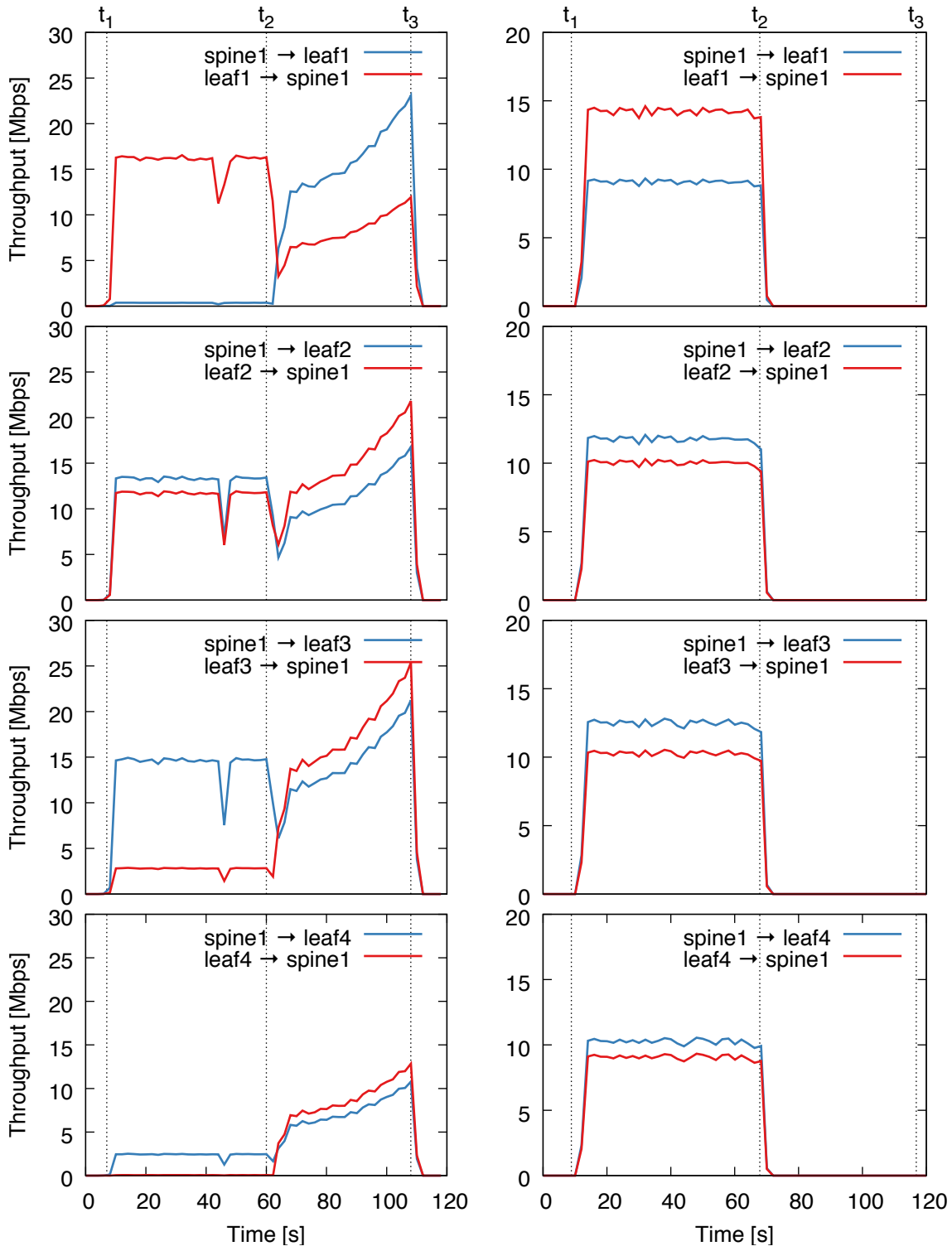
Name	Spec
CPU	Intel Xeon E5-2620 (2.00 GHz, 6 cores) × 2
Memory	64GB (DDR3-1600 8GB × 8)
Network	Gigabit Ethernet
OS	CentOS 7.2
Kernel	Linux 3.10
MPI Library	MPICH 3.1.4

primitives has been developed. The interconnect controller applies different routing strategies for each primitive as MPI primitive modules. The packet flow on the interconnect was observed using the port counters of switches to verify if the interconnect control can successfully switch from one to another when the MPI primitive executed changes.

The detailed experimental setup is as follows. The MPI application executes an iteration of MPI_Bcast followed by another iteration of MPI_Reduce. List 4.1 shows a simplified source code of this application. The process with rank 0 is specified as the root process for both MPI_Bcast and MPI_Reduce. The rank 0 process is configured to run on node01, which is connected to switch leaf1. Furthermore, the MPI application records the time where each of the following three events occurs: the start of the MPI_Bcast iteration (t_1), the start of the MPI_Reduce iteration (t_2) and the finish of the MPI_Reduce iteration (t_3). This timing information is used to investigate the relationship between the execution of the MPI application and the traffic change in the interconnect.

Each MPI primitive is repetitively executed because a single invocation of these primitives completes too quickly to observe the traffic change. The port counters of PF5240 are updated approximately once a second. This update frequency implies that instant traffic changes happening in less than one second cannot be precisely observed. Since a single invocation of MPI_Bcast or MPI_Reduce finishes in the order of milliseconds, each primitive is repeated to make its total execution time longer so that the traffic change can be observed using port counters.

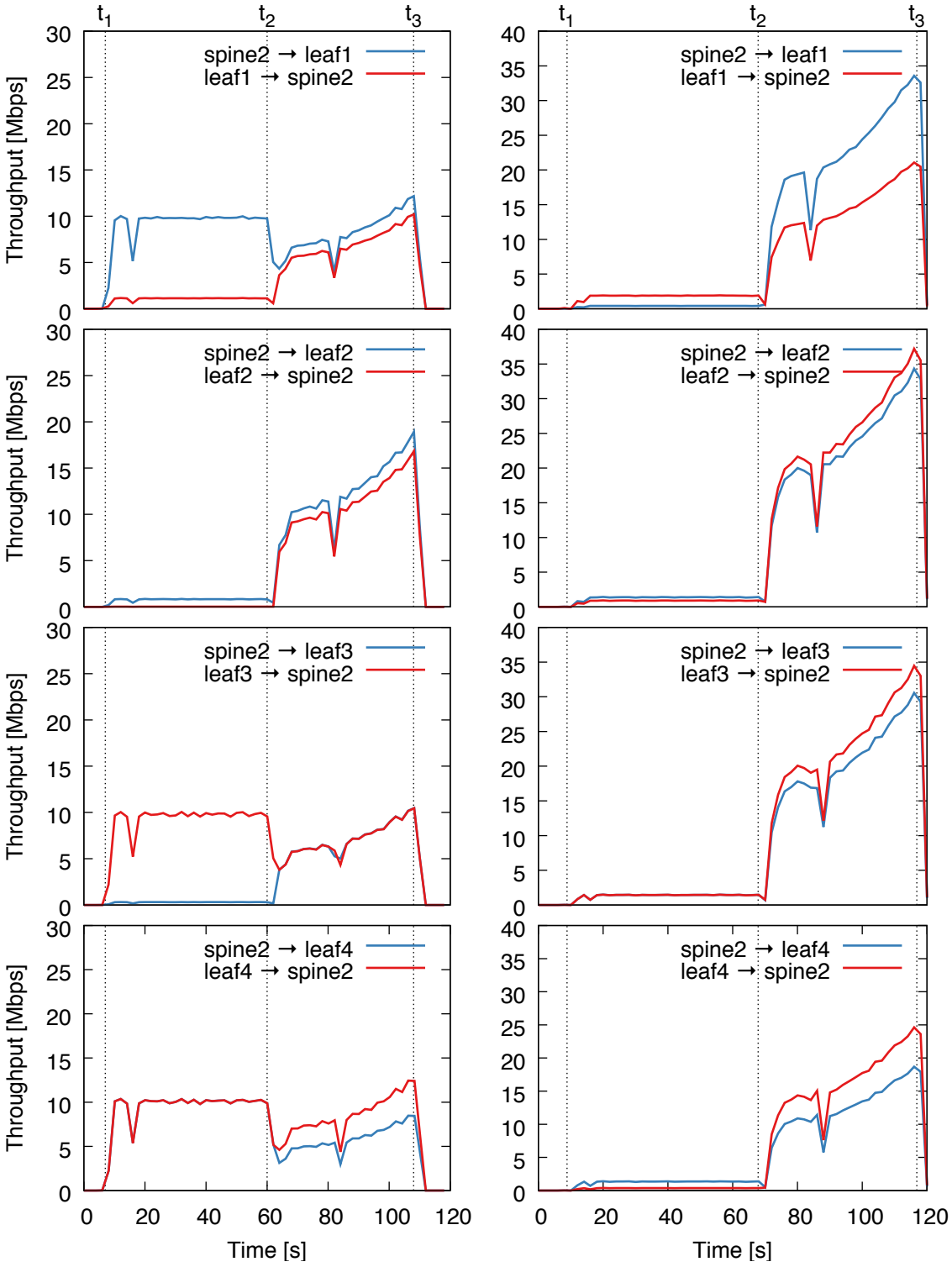
Under the interconnect topology of this experimental environment, there are always two possible paths between any two different leaf switches. One path contains spine1



(a) Conventional

(b) Proposed

Figure 4.6: Throughput Measured at Ports on Switch spine1



(a) Conventional (b) Proposed

Figure 4.7: Throughput Measured at Ports on Switch spine2

Listing 4.1: Source code of MPI application

```
1  #include <mpi.h>
2  #define BUF_SIZE (1000)
3  #define REPEAT_COUNT (10000)
4
5  char send_buf[BUF_SIZE];
6  char recv_buf[BUF_SIZE];
7
8  int main(int argc , char** argv) {
9      MPI_Init(&argc , &argv);
10
11     /* Record current time as t1 */
12
13     /* MPI_Bcast */
14     for (i = 0; i < REPEAT_COUNT; i++) {
15         MPI_Bcast(send_buf, BUF_SIZE, MPI_CHAR, 0,
16                 MPI_COMM_WORLD);
17     }
18
19     /* Record current time as t2 */
20
21     /* MPI_Reduce */
22     for (i = 0; i < REPEAT_COUNT; i++) {
23         MPI_Reduce(send_buf, recv_buf, BUF_SIZE,
24                  MPI_CHAR, MPI_SUM, 0,
25                  MPI_COMM_WORLD);
26     }
27
28     /* Record current time as t3 */
29
30     MPI_Finalize();
31 }
```

(*e.g.* leaf1–spine1–leaf2) and another path contains spine2 (*e.g.* leaf1–spine2–leaf2). The interconnect controller was deployed with a routing strategy that assigned paths utilizing spine1 to the traffic generated by MPI_Bcast. In contrast, the traffic generated by MPI_Reduce was set so that it went through spine2. Note that spine switches are never utilized by traffic between two compute nodes under an identical leaf switch. As a representative implementation of conventional networking architecture, an SDN controller was employed with a Equal Cost Multi Path (ECMP) routing strategy. To observe the traffic change in the interconnect, a measurement module that periodically (every two seconds) gathers and reports transmitted and received bytes of every switch port was integrated into the interconnect controller. Based on these port counter values, the throughput of the transmitted traffic and the received traffic of each port was calculated.

Figure 4.6 shows the change of throughput observerd at the ports of switch spine1. The four plots on the left column (Fig. 4.6 (a)) show the observed throughput when using ECMP, where as the four plots on the right column (Fig. 4.6 (b)) show the throughput when using the proposed mechanism. Each row corresponds to a port of switch spine1. For example, the plots in the first row show the throughput measured at the port connected to switch leaf1. In these plots, the time of event occurrences recorded by the MPI application (t_1–t_3) are marked with vertical dotted black lines. The temporal synchronization between throughput change and event occurrences was made by using timestamps. Figure 4.7 shows the observed throughput at switch spine2 in the same manner as Fig. 4.6.

When using ECMP, Figs. 4.6 (a) and 4.6 (a) indicate that both spine1 and spine2 were utilized during the execution of MPI_Bcast and MPI_Reduce as a result of load balancing. However, there is some inequality in the utilization of two spine switches. This inequality is because ECMP distributes the traffic workload not on the basis of not packets, but on flows.

MPICH, which is the MPI library used in UnisonFlow, has optimized implementations for collective communications like other MPI libraries. In particular, under the environment of this experiment, MPI_Bcast uses the binomial tree algorithm while MPI_Reduce uses the Rabenseifner's reduce algorithm [66]. As a result, MPI_Bcast is not a simple repeated point-to-point communication from the root process to other processes, but involves communication between non-root processes. For instance, the first plot in Fig. 4.6 (a) indicates how the traffic between spine1 and leaf1 changes. In detail, spine1→leaf1 shows the outgoing traffic from spine1 to leaf1, which is the aggregated

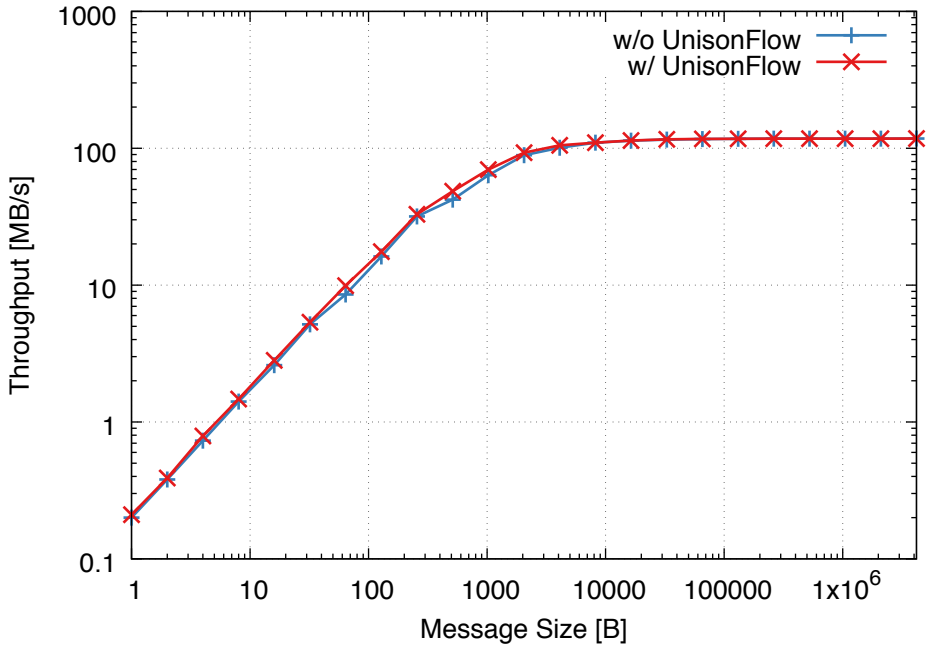


Figure 4.8: Throughput of Point-to-point Communication

traffic from the compute nodes under leaf2–leaf4 to the compute nodes under leaf1. In contrast, leaf1→spine1 shows the incoming traffic to spine1, which is the aggregated traffic from compute nodes under leaf1 to other compute nodes under leaf2, leaf3 and leaf4.

Figures 4.6 (b) and 4.7 (b) reveal the change of throughput when using the proposed mechanism. At t_1 where MPI_Bcast started, both incoming and outgoing traffic observed at the ports of spine1 rise steeply, while there is no clear growth of throughput at the ports of spine2. This result indicates that only spine1 was utilized during the execution of MPI_Bcast. When MPI_Bcast finished and then MPI_Reduce started at t_2, a sharp fall of throughput at spine1 was observed, whereas a rapid uptake in the throughput at spine2 was observed. After that, a sharp drop of throughput at the ports of spine2 was observed immediately when MPI_Reduce finished (t_3). This measurement result indicates that only spine2 was utilized during the execution of MPI_Reduce. Based on these observations, it is confirmed and verified that the network control is successfully synchronized with the execution of the MPI application.

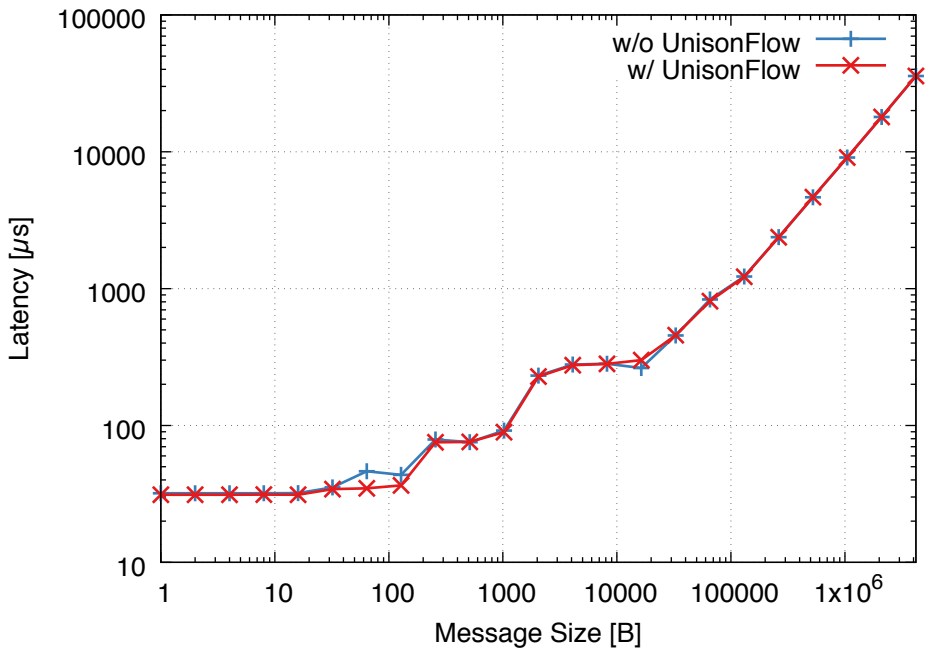


Figure 4.9: Latency of Point-to-point Communication

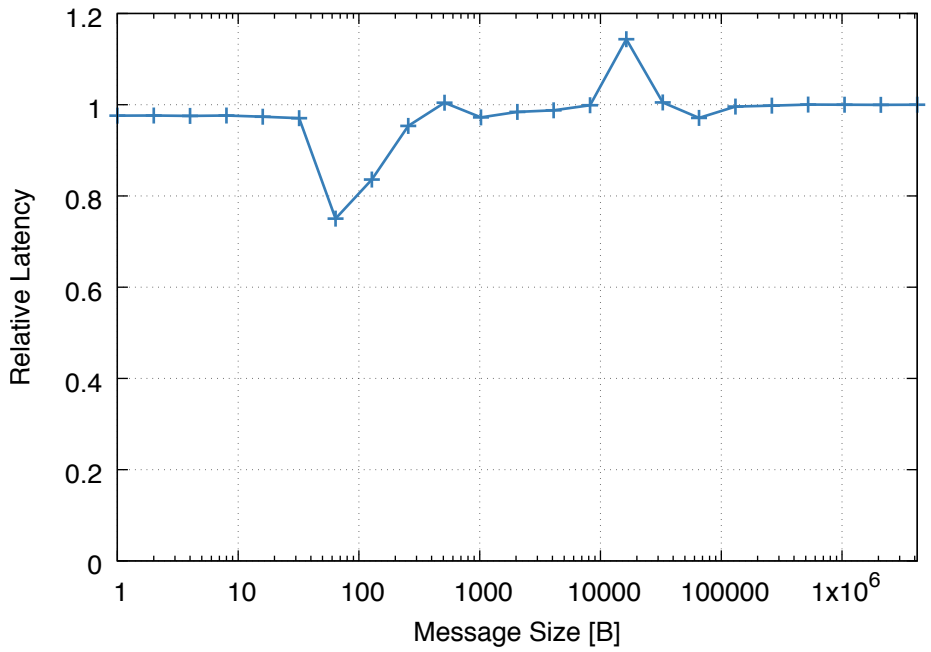


Figure 4.10: Relative Latency of Point-to-point Communication

4.4.3 Evaluation of Overhead

The primary source of the overhead incurred by the proposed mechanism is considered to be the tagging kernel module, because it requires per-packet inspection and modification over all packets emitted from a compute node. Additionally, rewriting the destination MAC address header field in the switches to restore the true MAC address can also be a source of overhead. In order to evaluate the total overhead caused by the proposal, the communication performance of point-to-point MPI primitives between node01 and node02 was measured using the OSU Micro-Benchmark Suite 5.3 [4], a widely adopted micro-benchmark for evaluating MPI communication performance. The result were compared with and without the proposed mechanism. The reason for measuring the performance of not collective communication but point-to-point communication is to remove unwanted influence from complex algorithms and communication patterns of collective communications. The osu_bw benchmark and osu_latency benchmark included in the OSU Micro-Benchmark suite were used to measure the bandwidth and latency, respectively.

Figure 4.8 shows a comparison of the throughput observed between node01 and node02. Figure 4.9 shows the comparison of latency for the same compute node pair. The plots in Figs. 4.8 and 4.9 represent the average of 500 measurements and 50,000 measurements, respectively. Figure 4.10 shows the relative latency when using the proposed mechanism compared to the latency without using the proposed mechanism. These plots indicate that performance degradation imposed by the proposed mechanism is practically negligible for both bandwidth and latency.

It should be noted, however, that Fig. 4.10 reveals a fluctuation in the latency. In particular, using the proposed mechanism resulted in a smaller latency than not using the proposed mechanism when the message size was 50–100 bytes. This is considered to be arising from the high jitter (*i.e.* variation of latency) caused by the network stack. Unlike InfiniBand that employs aggressive hardware offloading and kernel bypassing to reduce the latency and jitter of communication, TCP/IP over Ethernet is mostly implemented in the kernel. Therefore, the latency of TCP/IP communication is affected by the context switching and task scheduling of the kernel, which results in high jitter.

This experiment only evaluated the performance of point-to-point primitives. However, the fact that collective primitives are often implemented using multiple point-to-point

primitives [59, 62] implies that the overhead of the proposed mechanism is negligible for collective primitives as well.

4.5 Related Work

Several studies [16, 28] have been carried out to incorporate application-awareness into SDN. An extension to Open vSwitch and OpenFlow has been proposed [28] to realize an application-aware data plane. This extension adds flow tables with application-specific actions to the packet processing pipeline of Open vSwitch. Although this method covers most network applications, it is not able to efficiently read and process the payload of a packet because the flow matching mechanism has not been modified from the plain OpenFlow design. Thus, only pre-defined header fields can be used as matching criteria. Moreover, applying this method to bare-metal computer clusters is challenging because the hardware switches are out of the focus. The packet processing pipelines of commercial hardware switches cannot be modified since they are implemented using unmodifiable hardware. In contrast, the research summarized in this dissertation supports the existing hardware switches and per-packet inspection.

An application-aware routing scheme for big data applications has been presented in [16]. This routing scheme maintains a global view of the network topology and link usage. Based on this global view, the network controller dynamically allocates a path for each Hadoop network flow so that congestion is avoided and network utilization is increased. Experiments demonstrated that the application-aware SDN routing significantly improved the speed of the shuffle phase in Hadoop, in comparison with conventional routing mechanisms such as ECMP and Spanning Tree. The concept of optimizing the packet flow in the interconnect based on application-layer information is similar to SDN-enhanced MPI. However, the scheme to synchronize flow installation and a Hadoop job has not been clarified in this research.

Hybrid Flexibly Assignable Switch Topology (HFAST) [56] interconnect architecture tailors the interconnect topology to meet the communication requirements of different applications. This is achieved by utilizing reconfigurable optical circuit switches to dynamically provide the connection between packet switches. Additionally, a process allocation algorithm optimized for HFAST architecture is also presented. HFAST architecture can reduce required hardware resources compared to conventional fat-tree

interconnects. The research summarized in this dissertation is different from this research in terms that a technical design to extract communication patterns from applications and convey such information to the network controller in real-time is exhibited.

A software-defined multicasting mechanism for MPI has been presented in [21]. This mechanism offloads collective MPI primitives to programmable NICs and OpenFlow switches. This method heavily depends on specialized hardware such as NetFPGA, whereas the proposal in this dissertation is software-based.

Multi-Protocol Label Switching (MPLS) and UnisonFlow share the similar idea of eliminating the need to examine packet payloads by encoding the upper layer information into fixed-length tags that are processable by hardware. However, to the best of the author's knowledge, no work has tackled to integrate MPI with label switching networks.

4.6 Conclusion

This chapter proposed UnisonFlow, a software-defined coordination mechanism for SDN-enhanced MPI that performs network control in synchronization with the execution of an application. The proposed mechanism is characterized by a kernel-assisted approach to tag packets that are emitted from compute nodes with the MPI context information of each packet. Experiments conducted on a computer cluster have verified the synchronization between network control and the execution of the application. Moreover, evaluation experiments have indicated that the overhead incurred by the coordination mechanism is practically negligible.

There are still issues to be addressed in the future. First, SDN-enhanced MPI primitives developed in our previous work [23, 30] need to be adjusted as MPI primitive modules on the interconnect controller and tested to see if they are accelerated compared to conventional MPI primitives. Second, performance evaluation using real-world MPI applications is necessary. Although some individual SDN-enhanced MPI primitives and UnisonFlow as the coordination mechanism of message-passing communication and computation have been developed, it is still unclear how these elements can accelerate a practical application as a whole. Finally, how this architecture can be adopted to a computer cluster simultaneously running multiple jobs, which is common in practical deployments, needs to be investigated. Since the current implementation of the UnisonFlow assumes only one job running at the same time on a node, it needs to be enhanced to support

multiple concurrent jobs. This enhancement might involve an integration with the job scheduler.

5 Conclusion

5.1 Concluding Remark

The inter-process communication of applications running on cluster systems show distinctive patterns. However, in contrast to the application-dependent communication pattern, the interconnect is inherently designed in an application-agnostic manner because a real-world cluster is usually shared by many users and each user runs various applications. As a result, the imbalance in the packet flow on the interconnect can take place under particular combinations of communication pattern and interconnect. This imbalance can lead to traffic congestion on links in the interconnect, which lowers the throughput of communication and degrades the total application performance as a result.

This dissertation tackled this imbalance problem by taking the strategy of adapting the interconnect to the communication pattern of applications. Traditionally, such dynamic adaptation of the interconnect has been deemed infeasible due to the lack of a networking architecture, technology, or technique that allows flexible and dynamic reconfiguration. However, the recent emergence of programmable networking architectures exemplified by Software-Defined Networking (SDN) has opened up the possibility to realize such adaptation. This dissertation aimed to overcome this shortcoming of conventional application-agnostic interconnects by establishing a programmable interconnect control that dynamically controls the packet flow in the interconnect based on the communication pattern of applications.

The following three challenges have been tackled to achieve the goal described above: (1) analyzing the packet flow in the interconnect, (2) accelerating MPI communication by dynamically controlling the packet flow in the interconnect, and (3) coordinating the execution of application and interconnect control.

To address the first challenge, Chapter 2 proposed PFAnalyzer, a toolset for analyzing the packet flow in the interconnect. When designing and implementing an efficient

programmable interconnect control, researchers need to conduct a systematic analysis over many combinations of applications and interconnects. Since performing such an analysis on a physical cluster is time-consuming, this dissertation has chosen the strategy of using simulation to facilitate the analysis. The proposed toolset is a pair of tools: an interconnect simulator specialized for programmable interconnects, and a profiler to collect communication pattern from applications. PFSim has allowed researchers and designers working on interconnects to investigate possible congestion in the interconnect for an arbitrary cluster configuration and a set of communication patterns extracted by PFProf. In the evaluation, the accuracy of the simulation results obtained from PFSim was assessed. Furthermore, how PFAnalyzer can be used to analyze the effect of programmable interconnect control was demonstrated.

To address the second challenge, Chapter 3 proposed a framework to accelerate MPI collectives by dynamically controlling the packet flow in the interconnect. Message Passing Interface (MPI) is a standardized inter-process communication library widely used to develop parallel distributed applications for clusters. Out of the communication primitives provided by MPI, this dissertation focused on accelerating collective communication because it usually occupies a significant fraction of the execution time of applications. The network programmability provided by Software-Defined Networking was integrated into MPI collectives in such a way that MPI collectives were able to effectively utilize the bandwidth of the interconnect. In particular, this dissertation aimed to reduce the execution time of MPI_Allreduce, which is a frequently used MPI collective communication in many simulation codes. The speedup of MPI_Allreduce when using the proposed collective acceleration framework was evaluated.

To address the third challenge, Chapter 4 proposed UnisonFlow, a software-defined coordination mechanism that performs interconnect control in synchronization with the execution of applications. In real-world applications, the communication pattern changes with the execution of application. For the reason, a mechanism to coordinate packet flow control and execution of application is essential for the adaptation of the interconnect to the time-varying communication pattern of real-world applications. UnisonFlow was proposed as a kernel-assisted mechanism that realizes such coordination on a per-packet basis while maintaining significantly low overhead. Evaluation shown in Chapter 4 verified that the interconnect control was successfully performed in synchronization with the execution of the application and the overhead imposed by the coordination mechanism

was small.

5.2 Future Work

In this dissertation, it was assumed that only a single job is executed on the cluster. However, a production cluster usually executes multiple jobs simultaneously. Therefore, the proposed programmable interconnect control should be enhanced to support multiple concurrent jobs on a cluster. This enhancement is a challenging task because of the following two reasons. First, the interconnect control needs to be coordinated with the scheduling of jobs. In other words, the interconnect control needs to be triggered each time a job starts and a job exits. Such coordination could be realized by integrating the job scheduler with the interconnect controller. Second, inter-job interference of packet flow needs to be considered. The coexistence of multiple jobs on a single cluster implies that the packet flow generated by different jobs may share a single link. Under such situation, the packet flow generated by a communication-intensive job could occupy the interconnect and degrade the communication performance of other jobs. In fact, researchers have reported a significant performance variability of jobs on production clusters caused by the interference of packet flow between different jobs [32]. Therefore, the programmable interconnect control should globally optimize the packet flow in the interconnect while considering the communication pattern of each job and the interference between jobs, so that all jobs can equally benefit from the interconnect control.

Due to the limited scale of the cluster that was available for the experiments, the scalability of the proposed programmable interconnect control has not been thoroughly investigated yet. There are mainly two challenges in applying the programmable interconnect control to large-scale clusters composing of large number of compute nodes. The first challenge is the concentration of load on the interconnect controller. As described in Section 1.1.6, a centralized controller oversees the entire network in SDN. This design inherently poses a limit in the scale of the cluster since the work that needs to be performed by the controller, such as monitoring the state of the interconnect and exchanging control messages with the switches, dramatically increases with the number of compute nodes in the cluster. To overcome this limitation, multi-controller architecture [6] may be utilized. Under the multi-controller architecture, the network is managed by multiple controllers working cooperatively. The second challenge is the limit of flow entries that a switch can

handle. The number of flow entries that need to be installed on each switch increases rapidly with the number of compute nodes composing the cluster. However, the number of flow entries that can be stored in the TCAM of an SDN switch is limited. This problem might be solved by merging redundant flow entries or evicting rarely matched flow entries from switches.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor Professor Shinji Shimojo at the Cybermedia Center, Osaka University. He continuously and convincingly conveyed a spirit of adventure and excitement to this study. His excellent advices based on his deep knowledge always provided a unique perspective and new meaning to this study.

I would like to thank Professor Makoto Onizuka and Professor Yasuyuki Matsushita at the Department of Multimedia Engineering, the Graduate School of Information Science and Technology, Osaka University for serving as dissertation committee members and providing me valuable advice and guidance throughout my doctoral research work.

Sincere thanks also goes to Professor Takahiro Hara, Professor Toru Fujiwara, and Professor Norihiro Hagita at the Department of Multimedia Engineering, the Graduate School of Information Science and Technology, Osaka University for their appropriate guidance.

I would like to express my deepest appreciation to Associate Professor Susumu Date at the Cybermedia Center, Osaka University, for his patient and enthusiastic guidance along my studies. His exceptional ability to analyze, solve and present scientific problems has constantly impressed me. Without his support and encouragement, this dissertation would not have been possible.

I also would like to thank Dr. Eiji Kawai and Dr. Hiroaki Yamanaka at the National Institute of Information and Communications Technology, Associate Professor Hirotake Abe at the University of Tsukuba, Associate Professor Kohei Ichikawa at the Nara Institute of Science and Technology, Associate Professor Yoshiyuki Kido and Specially Appointed Associate Professor Yasuhiro Watashiba at the Cybermedia Center, Osaka University, for their encouragement, insightful comments, and hard questions.

I am grateful to Specially Appointed Associate Professor Kazufumi Hosoda and Specially Appointed Associate Professor Suyong Eum at the Institute for Transdisciplinary

Graduate Degree Programs, Osaka University, who always showed me daring ideas and unique approaches to look at a problem through discussions. I also would like to express my gratitude to the student advisory committee members of the Humanware Innovation Program, particularly Kaoru Amano at the National Institute of Information and Communications Technology, and Akira Sakakibara at Microsoft Japan, for their invaluable feedbacks. This research was supported in part by the scholarship and research funding provided by the Humanware Innovation Program.

This study greatly benefited from the comments and advices from Associate Professor Kazuhide Kojima at the Cybermedia Center, Osaka University, Professor Kaname Harumoto at the Institute for Datability Science, Osaka University, Associate Professor Hiroki Kashiwazaki, Guest Professor Takashi Yoshikawa, and Specially Appointed Associate Professor Chonho Lee at the Cybermedia Center, Osaka University. Their suggestions improved the quality of this thesis much further.

I would like to thank all the students at Shimojo laboratory for the sleepless nights we worked together before deadlines, and for all the fun we had. It was a great pleasure working with you all.

I would like express my heartfelt gratitude to my parents, Toshihito Takahashi and Naoko Takahashi, for their continuous support, encouragement and care. They made me the person who I am today.

Finally, but most importantly, I would like to express my deepest gratitude and appreciation to my wife Ziyan Xu for her unconditional love and support throughout my studies and my personal life.

References

[1] *List Statistics | TOP500 Supercomputer Sites*. [Online]. Available: http://www.top500.org/statistics/list/.

[2] *Mellanox Products: Fabric Collective Accelerator (FCA)*. [Online]. Available: http://www.mellanox.com/products/fca/.

[3] MIMD Lattice Computation Collaboration, *MIMD Lattice Computation (MILC) Collaboration Home Page*. [Online]. Available: http://physics.indiana.edu/~sg/milc.html.

[4] Ohio State University, *OSU Micro Benchmark*. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/.

[5] *Trema Full-Stack OpenFlow Framework in Ruby and C*. [Online]. Available: http://trema.github.io/trema/.

[6] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, "Multi-controller Based Software-Defined Networking : A Survey", *IEEE Access*, vol. 6, pp. 15 980–15 996, Mar. 2018. DOI: 10.1109/ACCESS.2018.2814738.

[7] J. Liu, A. Vishnu, and D. K. Panda, "Characterization of MPI Usage on a Production Supercomputer", in *2018 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'18)*, Nov. 2018.

[8] J. Y. Choi, J. Logan, M. Wolf, G. Ostrouchov, T. Kurc, Q. Liu, N. Podhorszki, S. Klasky, M. Romanus, Q. Sun, M. Parashar, R. M. Churchill, and C.-S. Chang, "TGE: Machine Learning Based Task Graph Embedding for Large-Scale Topology Mapping", in *2017 International Conference on Cluster Computing (CLUSTER 2017)*, Sep. 2017, pp. 587–591. DOI: 10.1109/CLUSTER.2017.67.

[9] E. A. Leon, I. Karlin, A. Bhatele, S. H. Langer, C. Chambreau, L. H. Howell, T. D'Hooge, and M. L. Leininger, "Characterizing Parallel Scientific Applications on Commodity Clusters: An Empirical Study of a Tapered Fat-Tree", in *2016 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, Nov. 2017, pp. 909–920. DOI: 10.1109/SC.2016.77.

[10] G. Michelogiannakis, K. Z. Ibrahim, J. Shalf, J. J. Wilke, S. Knight, and J. P. Kenny, "APHiD: Hierarchical Task Placement to Enable a Tapered Fat Tree Topology for Lower Power and Cost in HPC Networks", in *17th International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2017)*, May 2017, pp. 228–237. DOI: 10.1109/CCGRID.2017.33.

[11] M. Ruefenacht, M. Bull, and S. Booth, "Generalisation of Recursive Doubling for AllReduce: Now with Simulation", *Parallel Computing*, vol. 69, pp. 24–44, Nov. 2017. DOI: 10.1016/j.parco.2017.08.004.

[12] L. Alawneh, A. Hamou-Lhadj, and J. Hassine, "Segmenting Large Traces of Inter-process Communication with a Focus on High Performance Computing Systems", *Journal of Systems and Software*, vol. 120, pp. 1–16, Oct. 2016. DOI: 10.1016/j.jss.2016.06.067.

[13] S. Date, H. Abe, D. Khureltulga, K. Takahashi, Y. Kido, Y. Watashiba, P. Uchupala, K. Ichikawa, H. Yamanaka, E. Kawai, and S. Shimojo, "SDN-accelerated HPC Infrastructure for Scientific Research", *International Journal of Information Technology*, vol. 22, no. 1, 2016.

[14] S. Kumar, S. S. Sharkawi, and K. A. N. Jan, "Optimization and Analysis of MPI Collective Communication on Fat-Tree Networks", in *30th International Parallel and Distributed Processing Symposium (IPDPS 2016)*, May 2016, pp. 1031–1040. DOI: 10.1109/IPDPS.2016.85.

[15] H. Subramoni, A. M. Augustine, M. Arnold, J. Perkins, X. Lu, K. Hamidouche, and D. K. Panda, "INAM2: InfiniBand Network Analysis and Monitoring with MPI", in *2016 International Conference on High Performance Computing (ISC)*, Jun. 2016, pp. 300–320. DOI: 10.1007/978-3-319-41321-1_16.

[16] L.-w. Cheng and S.-y. Wang, "Application-Aware SDN Routing for Big Data Networking", in *2015 Global Communications Conference (GLOBECOM 2015)*, Dec. 2015, pp. 1–6. DOI: 10.1109/GLOCOM.2014.7417577.

[17] Y. Gong, B. He, and J. Zhong, "Network Performance Aware MPI Collective Communication Operations in the Cloud", *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 11, pp. 3079–3089, Nov. 2015. DOI: 10.1109/TPDS.2013.96.

[18] E. Jo, D. Pan, J. Liu, and L. Butler, "A Simulation and Emulation Study of SDN-based Multipath Routing for Fat-tree Data Center Networks", in *2014 Winter Simulation Conference (WSC 2014)*, Dec. 2015, pp. 3072–3083. DOI: 10.1109/WSC.2014.7020145.

[19] Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, 2015. [Online]. Available: http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[20] K. Takahashi, D. Khureltulga, B. Munkhdorj, Y. Kido, S. Date, H. Yamanaka, E. Kawai, and S. Shimojo, "Concept and Design of SDN-Enhanced MPI Framework", in *Fourth European Workshop on Software Defined Networks (EWSDN 2015)*, Sep. 2015, pp. 109–110. DOI: 10.1109/EWSDN.2015.72.

[21] O. Arap, G. Brown, B. Himebaugh, and M. Swany, "Software Defined Multicasting for MPI Collective Operation Offloading with the NetFPGA", in *20th European Conference on Parallel Processing (Euro-Par 2014)*, ser. Lecture Notes in Computer Science, vol. 8632, Springer International Publishing, Aug. 2014, pp. 632–643. DOI: 10.1007/978-3-319-09873-9_53.

[22] P. Berde, W. Snow, G. Parulkar, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, and P. Radoslavov, "ONOS: Towards an Open, Distributed SDN OS", in *Third Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, Aug. 2014, pp. 1–6. DOI: 10.1145/2620728.2620744.

[23] K. Dashdavaa, S. Date, H. Yamanaka, E. Kawai, Y. Watashiba, K. Ichikawa, H. Abe, and S. Shimojo, "Architecture of a High-Speed MPI_Bcast Leveraging Software-Defined Network", in *19th European Conference on Parallel Processing (Euro-Par 2014) Workshops*, ser. Lecture Notes in Computer Science, vol. 8374, Springer

Berlin Heidelberg, Aug. 2014, pp. 885–894. DOI: 10.1007/978-3-642-54420-0_86.

[24] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface, Third Edition*. The MIT Press, 2014, ISBN: 9780262527392.

[25] *IEEE Standard for Local and Metropolitan Area Networks - Media Access Control (MAC) Bridges and Virtual Bridged Local Area Networks - Amendment 22: Equal Cost Multiple Path (ECMP)*, Apr. 2014. DOI: 10.1109/IEEESTD.2014.6783684.

[26] S. Kumar, A. Mamidala, P. Heidelberger, D. Chen, and D. Faraj, "Optimization of MPI Collective Operations on the IBM Blue Gene/Q Supercomputer", *The International Journal of High Performance Computing Applications*, vol. 28, no. 4, pp. 450–464, Nov. 2014. DOI: 10.1177/1094342014552086.

[27] J. Medved, R. Varga, A. Tkacik, and K. Gray, "OpenDaylight: Towards a Model-Driven SDN Controller Architecture", in *15th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM 2014)*, Jun. 2014, pp. 1–6. DOI: 10.1109/WoWMoM.2014.6918985.

[28] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. Lakshman, "Application-aware Data Plane Processing in SDN", in *Third Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*, 2014, pp. 13–18. DOI: 10.1145/2620728.2620735.

[29] Ryu SDN Framework Community, *Ryu SDN Framework*, 2014. [Online]. Available: https://osrg.github.io/ryu/.

[30] K. Takahashi, D. Khureltulga, Y. Watashiba, Y. Kido, S. Date, and S. Shimojo, "Performance Evaluation of SDN-enhanced MPI_Allreduce on a Cluster System with Fat-tree Interconnect", in *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Jul. 2014, pp. 784–792. DOI: 10.1109/HPCSim.2014.6903768.

[31] T. Adachi, N. Shida, K. Miura, S. Sumimoto, A. Uno, M. Kurokawa, F. Shoji, and M. Yokokawa, "The Design of Ultra Scalable MPI Collective Communication on the K Computer", *Computer Science - Research and Development*, vol. 28, no. 2-3, pp. 147–155, May 2013. DOI: 10.1007/s00450-012-0211-7.

[32] A. Bhatele, K. Mohror, S. H. Langer, and K. E. Isaacs, "There Goes the Neighborhood: Performance Degradation Due to Nearby Jobs", in *2013 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, Nov. 2013, pp. 1–12. DOI: 10.1145/2503210.2503247.

[33] U. Brandes, M. Eiglsperger, J. Lerner, and C. Pich, "Handbook of graph drawing and visualization", in, R. Tamassia, Ed. CRC Press, Oct. 2013, ch. Graph Markup Language (GraphML), pp. 517–541, ISBN: 9781138034242.

[34] B. Goglin and S. Moreaud, "KNEM: a Generic and Scalable Kernel-Assisted Intra-node MPI Communication Framework", *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176–188, Feb. 2013.

[35] R. Rosen, *Linux Kernel Networking: Implementation and Theory*. Apress, Dec. 2013, ISBN: 978-1-4302-6196-4.

[36] Y. Ajima, T. Inoue, S. Hiramoto, Y. Takagi, and T. Shimizu, "The Tofu Interconnect", in *IEEE Micro*, vol. 32, Nov. 2012, pp. 21–31. DOI: 10.1109/MM.2011.98.

[37] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray Cascade: A Scalable HPC System Based on a Dragonfly Network", in *2012 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'12)*, Nov. 2012, pp. 1–9. DOI: 10.1109/SC.2012.39.

[38] K. Kandalla, U. Yang, J. Keasler, T. Kolev, A. Moody, H. Subramoni, K. Tomko, J. Vienne, B. R. de Supinski, and D. K. Panda, "Designing Non-blocking Allreduce with Collective Offload on InfiniBand Clusters: A Case Study with Conjugate Gradient Solvers", in *26th International Parallel and Distributed Processing Symposium (IPDPS 2012)*, May 2012, pp. 1156–1167. DOI: 10.1109/IPDPS.2012.106.

[39] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, W. E. Nagel, Y. Oleynik, P. Philippen, P. Saviankou, D. Schmidl, S. Shende, R. Tschüter, M. Wagner, B. Wesarg, and F. Wolf, "Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir", in *Tools for High Performance Computing*, Jan. 2012, pp. 79–91. DOI: 10.1007/978-3-642-31476-6_7.

[40] Open Networking Foundation, *Software-Defined Networking: The New Norm for Networks*, 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[41] I. Bird, "Computing for the Large Hadron Collider", *Annual Review of Nuclear and Particle Science*, vol. 61, no. 1, pp. 99–118, Nov. 2011. DOI: 10.1146/annurev-nucl-102010-130059.

[42] T. Hoefler and M. Snir, "Generic Topology Mapping Strategies for Large-scale Parallel Architectures", in *International Conference on Supercomputing (ICS '11)*, Jun. 2011, pp. 75–84. DOI: 10.1145/1995896.1995909.

[43] S. Sur, S. Potluri, K. Kandalla, H. Subramoni, K. Tomko, and D. K. Panda, "Co-Designing MPI Library and Applications for InfiniBand Clusters", *IEEE Computer*, Nov. 2011.

[44] M. Yokokawa, F. Shoji, A. Uno, M. Kurokawa, and T. Watanabe, "The K Computer: Japanese Next-Generation Supercomputer Development Project", in *17th International Symposium on Low Power Electronics and Design (ISLPED'11)*, Aug. 2011, pp. 371–372. DOI: 10.1109/ISLPED.2011.5993668.

[45] R. Alverson and D. Roweth, "The Gemini System Interconnect", in *18th Symposium on High Performance Interconnects (HOTI 2010)*, Aug. 2010, pp. 83–87. DOI: 10.1109/HOTI.2010.23.

[46] T. Hoefler, T. Schneider, and A. Lumsdaine, "LogGOPSim: Simulating Large-scale Applications in the LogGOPS Model", in *19th International Symposium on High Performance Distributed Computing (HPDC '10)*, Jun. 2010, pp. 597–604. DOI: 10.1145/1851476.1851564.

[47] S. Kamil, L. Oliker, A. Pinar, and J. Shalf, "Communication Requirements and Interconnect Optimization for High-End Scientific Applications", *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 188–202, Feb. 2010. DOI: 10.1109/TPDS.2009.61.

[48] *IEEE Standard for Local and Metropolitan Area Networks– Station and Media Access Control Connectivity Discovery*, Sep. 2009. DOI: 10.1109/IEEESTD.2009.5251812.

[49] T. Schneider and T. Hoefler, "ORCS: An Oblivious Routing Congestion Simulator", *Indiana University, Computer*, no. 675, 2009.

[50] M. M. Tikir, M. A. Laurenzano, L. Carrington, and A. Snavely, "PSINS: An Open Source Event Tracer and Execution Simulator", in *Department of Defense High Performance Computing Modernization Program - Users Group Conference (HPCMP-UGC)*, Jun. 2009, pp. 444–449. DOI: 10.1109/HPCMP-UGC.2009.73.

[51] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, Aug. 2008. DOI: 10.1145/1402946.1402967.

[52] P. Geoffray and T. Hoefler, "Adaptive Routing Strategies for Modern High Performance Networks", in *16th Symposium on High Performance Interconnects (HOTI 2008)*, Aug. 2008, pp. 165–172. DOI: 10.1109/HOTI.2008.21.

[53] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, Highly-scalable Dragonfly Topology", in *International Symposium on Computer Architecture (ISCA)*, vol. 36, Jun. 2008, pp. 77–88. DOI: 10.1109/ISCA.2008.19.

[54] A. Knüpfer, H. Brunst, J. Doleschal, M. Jurenz, M. Lieber, H. Mickler, M. S. Müller, and W. E. Nagel, "The Vampir Performance Analysis Tool-Set", in *Tools for High Performance Computing*, Jul. 2008, pp. 139–155. DOI: 10.1007/978-3-540-68564-7_9.

[55] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks", *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008. DOI: 10.1145/1355734.1355746.

[56] S. Kamil, A. Pinar, D. Gunter, M. Lijewski, L. Oliker, and J. Shalf, "Reconfigurable Hybrid Interconnection for Static and Dynamic Scientific Applications", in *4th International Conference on Computing Frontiers (CF'07)*, Jan. 2007, pp. 183–194. DOI: 10.1145/1242531.1242559.

[57] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance Analysis of MPI Collective Operations", *Cluster Computing*, vol. 10, no. 2, pp. 127–143, Apr. 2007. DOI: 10.1007/s10586-007-0012-0.

[58] S. Trowbridge, "High-Speed Ethernet Transport", *IEEE Communications Magazine*, vol. 45, no. 12, pp. 120–125, Dec. 2007. DOI: 10.1109/MCOM.2007.4395376.

[59] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda, "Design of High Performance MVAPICH2: MPI2 over InfiniBand", in *Sixth International Symposium on Cluster Computing and the Grid (CCGrid'06)*, May 2006, pp. 43–48. DOI: 10.1109/CCGRID.2006.32.

[60] T. Jones, *MPI PERUSE: An MPI Extension for Revealing Unexposed Implementation Information*, 2006.

[61] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System", *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, May 2006. DOI: 10.1177/1094342006064482.

[62] J. M. Squyres and A. Lumsdaine, "The Component Architecture of Open MPI: Enabling Third-Party Collective Algorithms", in *Component Models and Systems for Grid Applications*, Kluwer Academic Publishers, Jun. 2005, pp. 167–185. DOI: 10.1007/0-387-23352-0_11.

[63] R. Thakur, R. Rabenseifner, and G. William, "Optimization of Collective Communication Operations in MPICH", *International Journal of High Performance Computing Applications*, vol. 19, pp. 49–66, Feb. 2005. DOI: 10.1177/1094342005051521.

[64] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science,

vol. 3241, Springer Berlin Heidelberg, Sep. 2004, pp. 97–104. DOI: 10.1007/978-3-540-30218-6_19.

[65] J. Liu, A. Vishnu, and D. K. Panda, "Building Multirail InfiniBand Clusters: MPI-Level Design and Performance Evaluation", in *2004 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'04)*, Nov. 2004, pp. 33–33. DOI: 10.1109/SC.2004.15.

[66] R. Rabenseifner, "Optimization of Collective Reduction Operations", in *Computational Science (ICCS 2004)*, ser. Lecture Notes in Computer Science, vol. 3036, Springer Berlin Heidelberg, Jun. 2004, pp. 1–9. DOI: 10.1007/978-3-540-24685-5_1.

[67] W. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, Dec. 2003, ISBN: 0122007514.

[68] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management", in *Job Scheduling Strategies for Parallel Processing (JSSPP 2003)*, ser. Lecture Notes in Computer Science, vol. 2862, Springer Berlin Heidelberg, Jun. 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[69] W. Gropp, "MPICH2: A New Start for MPI Implementations", in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, vol. 2474, Springer Berlin Heidelberg, Sep. 2002, p. 7. DOI: 10.1007/3-540-45825-5_5.

[70] R. Buyya, T. Cortes, and H. Jin, "An Introduction to the InfiniBand Architecture", in *High Performance Mass Storage and Parallel I/O*, Wiley-IEEE Press, Dec. 2001, p. 688, ISBN: 9780470544839. DOI: 10.1109/9780470544839.ch42.

[71] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient Algorithms for All-to-all Communications in Multiport Message-passing Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 11, pp. 1143–1156, Nov. 1997. DOI: 10.1109/71.642949.

[72] G. Iannello, "Efficient Algorithms for the Reduce-scatter Operation in LogGP", *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 970–982, Sep. 1997. DOI: 10.1109/71.615442.

[73] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks", *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, Sep. 1991. DOI: 10.1177/10943420910050030 6.

[74] C. E. Leiserson, "Fat-trees: Universal Networks for Hardware-efficient Supercomputing", *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 892–901, Oct. 1985. DOI: 10.1109/TC.1985.6312192.