IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received March 11, 2018, accepted April 10, 2018, date of publication April 23, 2018, date of current version May 16, 2018.
Digital Object Identifier 10.1109/ACCESS.2018.2829532

UnisonFlow: A Software-Defined Coordination Mechanism for Message-Passing Communication and Computation

KEICHI TAKAHASHI[1], (Student Member, IEEE), SUSUMU DATE[2], (Member, IEEE), DASHDAVAA KHURELTULGA[1], YOSHIYUKI KIDO[2], (Member, IEEE), HIROAKI YAMANAKA[3], (Member, IEEE), EIJI KAWAI[3], (Member, IEEE), AND SHINJI SHIMOJO[2], (Member, IEEE)
[1]Multimedia Engineering Department, Graduate School of Information Science and Technology, Osaka University, Suita 565-0871, Japan
[2]Applied Information Systems Research Division, Cybermedia Center, Osaka University, Suita 567-0047, Japan
[3]ICT Testbed Research and Development Promotion Center, National Institute of Information and Communications Technology, Tokyo 184-8795, Japan

Corresponding author: Keichi Takahashi (takahashi.keichi@ais.cmc.osaka-u.ac.jp)

This work was supported in part by JSPS KAKENHI under Grant JP26330145 and Grant JP17K00168, in part by collaborative research of the National Institute of Information and Communication Technology and Osaka University (Research on high functional network platform technology for large-scale distributed computing), and in part by the Program for Leading Graduate Schools of the Ministry of Education, Culture, Sports, Science and Technology, Japan.

ABSTRACT Message passing interface (MPI) communication performance is becoming one of the key factors heavily affecting the total performance of data-intensive applications running on computer clusters. Our software-defined networking (SDN)-enhanced MPI improves the performance of communication over interconnects by integrating flexible and dynamic network controllability of SDN into MPI. We have demonstrated that the acceleration of individual MPI communication primitives is feasible through our past work on the SDN-enhanced MPI. However, real-world MPI applications have not benefited from such accelerated communication primitives through our research achievements to date, because each of the distinct network control algorithms designed for various MPI communication primitives cannot be activated and coordinated with the execution of the MPI application. Therefore, this paper proposes UnisonFlow, a software-defined coordination mechanism for the SDN-enhanced MPI that performs network control in synchronization with the execution of applications. An experiment conducted on a real-computer cluster verifies that the interconnect control can be successfully performed in synchronization with the execution of the application. Furthermore, the synchronization is performed with a low overhead and its performance penalty is practically negligible.

INDEX TERMS Message passing interface, software defined network, openflow, kernel assistance, interconnects.

I. INTRODUCTION

Recent scientific research has been taking major advantage of computational analysis and simulation. Sustained growth in the volume of data generated by scientific experiments has lead to a rise in the importance of data-intensive computing. For example, approximately 15PB of experimental data is annually generated and processed at the Large Hadron Collider (LHC), an experimental facility for high energy physics [2].

Today, in general, data-intensive computations are performed on high-performance computer clusters. A computer cluster is composed of a set of computing 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 is 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 computing nodes involved in a computation

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

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 [8]. 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 are unable to adaptively reconfigure themselves to match requirements from applications. In fact, in InfiniBand [7], 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 of 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, recent scale-out in the number of computing nodes has revealed two potential shortcomings of over-provisioned designs. First, the cost of 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 computing nodes increases. The second shortcoming is the underutilization 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, we believe that a novel cluster architecture which dynamically controls the traffic flow in the interconnect based on the communication pattern of the application can alleviate the aforementioned two shortcomings of conventional over-provisioned designs. For this reason, Software Defined Networking enhanced Message Passing Interface (*SDN-enhanced MPI*), which is an unconventional MPI framework that incorporates flexible network controllability of SDN into interconnects, was proposed in our past research. Furthermore, in past research towards SDN-enhanced MPI, we have demonstrated that the acceleration of collective MPI communication is feasible.

However, a technical challenge still remained in this research; namely, applying our research achievements to real-world MPI applications. In the preliminary stage of our current research so far, we focused on verifying the feasibility of our idea by investigating whether individual MPI collective communications could be accelerated or not. Therefore, how MPI communication accelerated with SDN could be synchronized with the execution of an MPI application remained a question that required a new technical innovation.

To this end, we propose *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 [18]. 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 paper 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 paper is organized as follows. Section II introduces SDN-enhanced MPI and its key technologies. Subsequently, the challenge to realize SDN-enhanced MPI is derived. Section III describes our proposed mechanism and its implementation. Section IV shows the result of the experiments conducted to demonstrate the feasibility of the proposal. Section V reviews related literature and clarifies the contributions of this paper. Finally, section VI discusses future issues to be tackled and concludes this paper.

II. 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.

A. MESSAGE PASSING INTERFACE (MPI)

Message Passing Interface (MPI) [13] is a *de facto* standard specification for inter-process communication libraries used to develop parallel distributed applications. MPI defines a suite of communication primitives that helps programmers to develop applications that require complex communications among computing nodes.

The communication primitives defined in MPI can be roughly categorized into point-to-point communication and collective communication. Point-to-point communication is a communication between one sender and one receiver. On the other hand, collective communication involves a group of multiple processes. Table 1 shows some representative examples of MPI primitives.

A remarkable feature of MPI is that it abstracts the underlying network of high-performance computer clusters. This abstraction allows programmers to develop applications

TABLE 1. Examples of MPI primitives.

Name	Category	Description
MPI_Send/MPI_Recv	Point-to-point	Blocking send/receive
MPI_Isend/MPI_Irecv	Point-to-point	Non-blocking send/receive
MPI_Bcast	Collective	Broadcast
MPI_Reduce	Collective	Reduction
MPI_Allreduce	Collective	Broadcast result of reduction
MPI_Gather	Collective	Aggregate pieces of data from processes into a single process
MPI_Alltoall	Collective	Perform MPI_Gather from all processes

without forcing them to study the detailed architecture or structure of the underlying network. In MPI, every process is identified by a *rank* number, a non-negative integer. The mapping between rank numbers and network addresses is automatically handled by the MPI library. Communication can be restricted into a certain group of processes, which is called a *communicator* in MPI. This abstraction makes MPI applications portable and easy to port to different computer clusters.

Until today, countless scientific applications have been developed by utilizing the communication primitives in MPI. The execution time of these MPI primitives is an important performance factor because its impact to the total application performance appears significantly accompanied with the recent scale-out of computer clusters. In other words, the total performance of MPI applications can be improved by optimizing the performance of MPI communications. For this reason, researchers have extensively investigated methods to improve the communication performance of MPI from various aspects.

B. SOFTWARE DEFINED NETWORKING (SDN)

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 has allowed SDN to deliver the following three benefits:

- *Programmable*: The control plane can be handled by a software controller. Network operators can program controllers tailored for their needs.
- *Dynamic*: SDN allows the controller to quickly reconfigure the network. For instance, it is possible to dynamically optimize traffic flow in the network based on the real-time traffic pattern.
- *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.

OpenFlow [11] is a widely accepted open standard of SDN. In an OpenFlow-enabled network, the data plane is handled by OpenFlow switches. Every OpenFlow switch holds a logical construct called *flow table*, which is a collection of *flow entries*. Each flow entry defines what kind of packet control should be performed on what kind of packets (Fig. 1). 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 a matching flow entry is found, the action of the matched flow entry is applied to the packet.

Header Fields			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. An example of a flow table.

The OpenFlow controller is responsible for the control plane. It manages the flow table of switches by adding, removing and modifying flow entries. The controller and switches communicate with each other by asynchronously exchanging messages defined in the OpenFlow protocol. One of the frequently used messages is the *packet-in* message, which is sent out from a switch to the controller when no matching flow entry is found for an incoming packet. In response, the controller can send a *modify flow entry* message to install a new flow entry on the switch.

C. 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 II-A, MPI mainly focuses on hiding the complexity of the underlying network architecture. Therefore, MPI does not provide any functionality for explicitly controlling the network. Integrating SDN into MPI could complement such lack of a network control feature in MPI and allow MPI to optimize the traffic flow in the network in accordance with the communication pattern of applications.

At the time of writing this paper, we have applied the above described basic idea 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 [4], [19]. SDN-enhanced MPI_Bcast [4] accelerates MPI_Bcast by utilizing the hardware multicast functionality of OpenFlow switches. SDN-enhanced MPI_Allreduce [19] dynamically reconfigures the path allocation based on the communication pattern of MPI_Allreduce so that congestion in links is minimized.

D. 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 II-C,

actual MPI applications have not yet taken 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.

In this paper, we realize a software-defined coordination mechanism to perform network control in synchronization with the execution of an application. Furthermore, the following technical requirements must be met 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*: We place 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 existent MPI applications on SDN-enhanced MPI, the existent MPI application 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.

III. PROPOSAL

A. 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 paper, 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 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 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 application-layer information into a header field of packets so that OpenFlow switches can perform application-aware packet flow control.

In detail, 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 the binary layout of a tag are shown in Fig. 2.

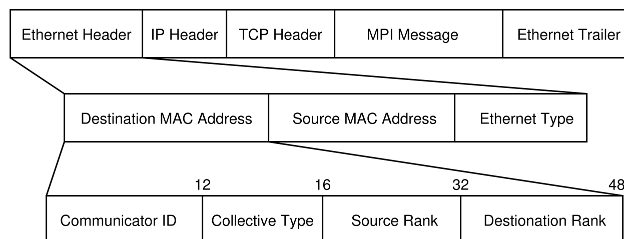


FIGURE 2. Tag information embedded in a packet.

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. For this reason, 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.

B. ARCHITECTURE

1) OVERVIEW

Figure 3 illustrates an overview of UnisonFlow. At this stage of research, it is assumed that a computer cluster executes a single MPI application because our research target is the acceleration of inter-node communication in MPI. The operating system of computing nodes is assumed to be Linux.

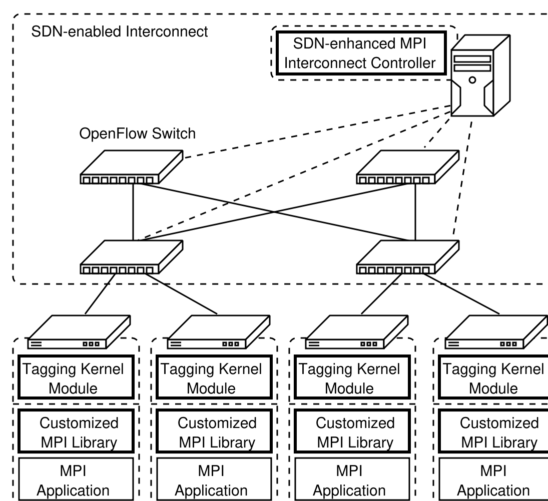


FIGURE 3. Overall architecture of UnisonFlow.

We have developed three major software modules that constitute this architecture (bold rectangles in Fig. 3). The first module is the *Interconnect Controller*, which is

basically an OpenFlow controller responsible for installing flow entries into OpenFlow switches. The interconnect controller was developed based on the Ryu SDN controller framework [17]. The second module is the *Tagging Kernel Module*. It resides in the kernel space of each computing 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 is dynamically linked to the MPI application. MPICH [5], an implementation of MPI, 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.

2) INTRA-NODE ARCHITECTURE

On each computing node, the tagging kernel module and MPI library 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 supplementary information used for filtering out non-MPI traffic.

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

As a possible solution for the functional component on the Linux kernel, we have considered *ebtables*, *raw socket* and *protocol handler* [16]. Ebtables is commonly used as an L2 packet filter that includes Network Address Translation (NAT) for MAC addresses. Raw sockets are special type of sockets that give user space programs access to the whole packet including protocol headers. The protocol handler is a function that can be inserted to the kernel, usually to add new network protocols. Out of these potential solutions, we have adopted the protocol handler because it achieves 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, we can utilize the network stack of the kernel and avoid re-implementing another network stack. 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 illustrates how MPI packets are processed on a computing node. The solid arrows represent packet flows generated by an MPI application. The dashed arrow represents interaction between software modules.

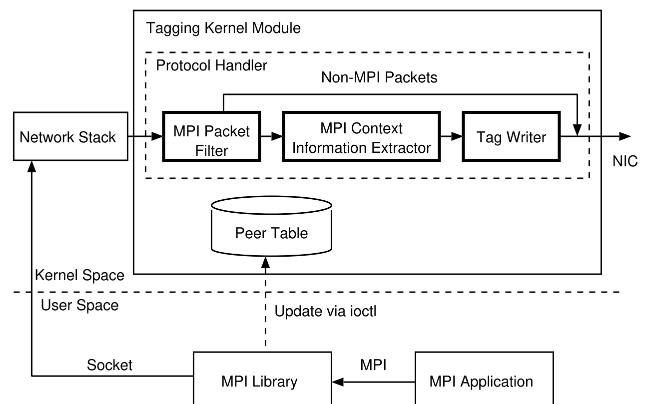


FIGURE 4. **Intra-node packet flow.**

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), respectively:

1. *MPI packet filter*: Packets generated by SSH, remote file systemis, and any 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 that is prepended to every MPI message by the MPI library for identification. Although the message envelope is prescribed in the MPI specification [13], 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. 2. 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 computing 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 was guaranteed.

3) 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 about 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 5 illustrates an example of the packet flow between two remote computing nodes. When the interconnect controller receives a packet-in message caused by an unmatched packet (step 1 in Fig. 5), 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. Based on this decision, flow entries are generated and then installed to relevant switches (step 4).

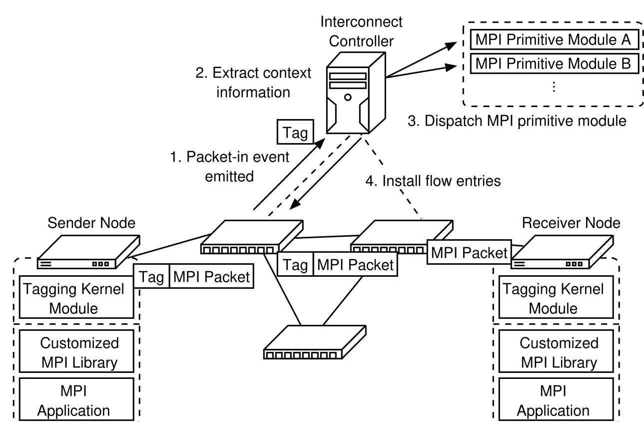


FIGURE 5. Inter-node packet flow.

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.

IV. EVALUATION

Two experiments were 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.

A. EXPERIMENTAL ENVIRONMENT

Both of the two experiments were conducted on the SDN-enabled computer cluster shown in Fig. 6. For the topology of the interconnect, a two-level fat-tree composed of six switches was adopted because fat-trees are 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 a limited number of available OpenFlow switches in our institution. In the following discussion, we refer to the two upper layer switches 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 computing nodes were connected to a leaf switch; that is, 24 computing nodes in total. These computing nodes are hereinafter referred to as node01 to node24. Leaf switches and computing nodes were interconnected with 1Gbps Ethernet. A management node accommodating the interconnect controller was also prepared.

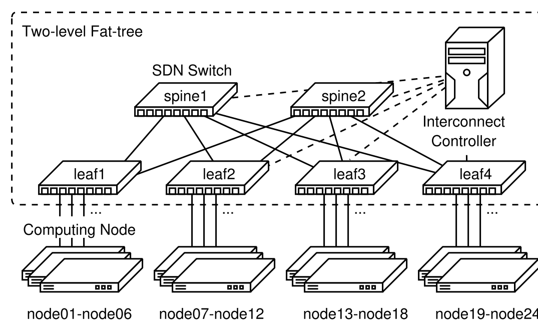


FIGURE 6. Overview of the experimental environment.

For SDN switches, NEC® ProgrammableFlow® PF5240 has been adopted. The computing node was a SGI® Rackable® Half-Depth Server C1001 equipped with the hardware and software as shown in Table 2.

B. 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

TABLE 2. Computing node specifications.

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

control and execution of the application, an MPI application which sequentially executes two different MPI primitives has been developed. The interconnect controller applies different routing strategies for each primitive as MPI primitive modules. We then observe the traffic flow on the interconnect 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 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.

```
#include <mpi.h>
#define BUF_SIZE (1000)
#define REPEAT_COUNT (10000)

char send_buf[BUF_SIZE];
char recv_buf[BUF_SIZE];

int main(int argc , char** argv) {
    MPI_Init(&argc , &argv);

    /* Record current time as t1 */

    /* MPI_Bcast */
    for (i = 0; i < REPEAT_COUNT; i++) {
        MPI_Bcast(send_buf, BUF_SIZE, MPI_CHAR, 0,
                  MPI_COMM_WORLD);
    }

    /* Record current time as t2 */

    /* MPI_Reduce */
    for (i = 0; i < REPEAT_COUNT; i++) {
        MPI_Reduce(send_buf, recv_buf, BUF_SIZE,
                   MPI_CHAR, MPI_SUM, 0,
                   MPI_COMM_WORLD);
    }

    /* Record current time as t3 */

    MPI_Finalize();
}
```

Listing 1. Source code of MPI application.

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 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, we repeat each primitive to make its total execution time longer so that we can observe the traffic change using port counters.

Under the interconnect topology of this experimental environment, there are always two possible paths between any two different leaf switches. One is the path that contains spine1 (*e.g.* leaf1 - spine1 - leaf2) and the other path contains spine2 (*e.g.* leaf1 - spine2 - leaf2). The interconnect controller was deployed with a routing strategy that assigns paths utilizing spine1 to the traffic generated by MPI_Bcast. In contrast, the traffic generated by MPI_Reduce was set so that it goes through spine2. Note that spine switches are never utilized by traffic between two computing 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, we calculated the throughput of the transmitted traffic and the received traffic of each port.

Figures 7 and 8 show the change of throughput observed at the ports of switches spine1 and spine2 when using ECMP. 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 implementation used in UnisonFlow, has optimized implementations for collective communications like other MPI libraries. In particular, under the environment of this experiment, MPI_Bcast uses binomial tree algorithm while MPI_Reduce uses the Rabenseifner's reduce algorithm [15]. 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, Fig. 7a indicates how the traffic between spine1 and leaf1 changes. In detail, TX shows the outgoing traffic from spine1 to leaf1, which is the aggregated traffic from the computing nodes under leaf2, leaf3 and leaf4 to the computing nodes under leaf1. In contrast, RX shows the aggregated traffic from computing nodes under leaf1 to other computing nodes under leaf2, leaf3 and leaf4.

Figures 9 and 10 show the change of throughput when using the proposed mechanism. In these plots, the time of event occurrences recorded by the MPI application are marked with vertical solid black lines. The time

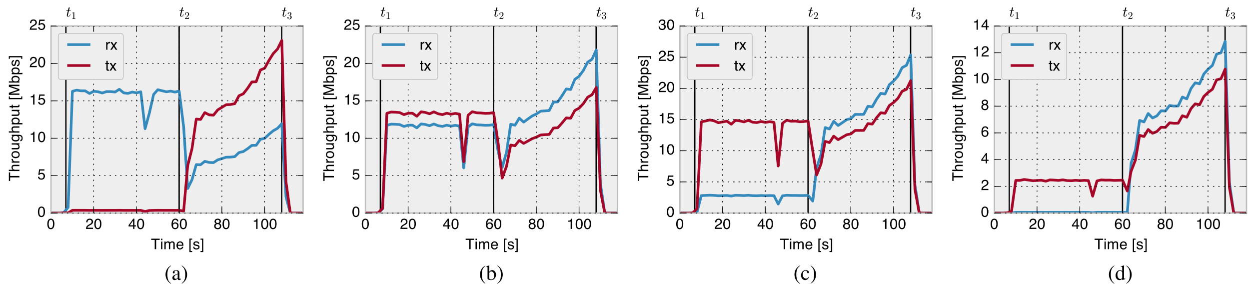


FIGURE 7. **Throughput measured at ports on switch spine1 (conventional). (a) At port 0xa00 (spine1) towards port 0xa00 (leaf1). (b) At port 0xb00 (spine1) towards port 0xa00 (leaf2). (c) At port 0xc00 (spine1) towards port 0xa00 (leaf3). (d) At port 0xd00 (spine1) towards port 0xa00 (leaf4).**

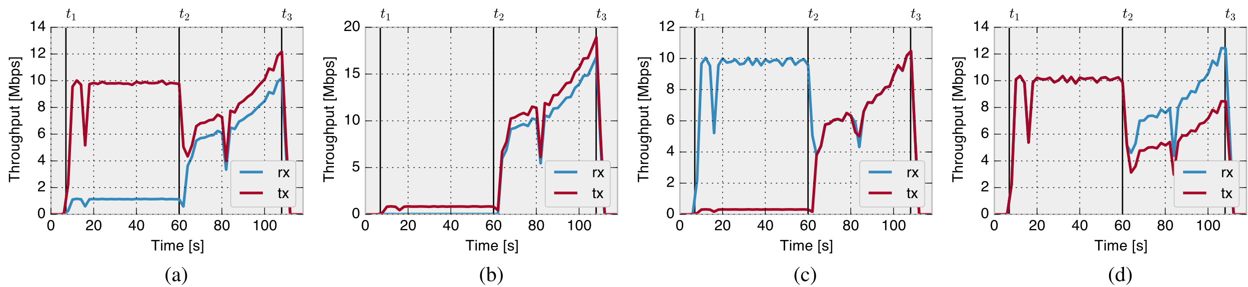


FIGURE 8. **Throughput measured at ports on switch spine2 (conventional). (a) At port 0xa00 (spine2) towards port 0xb00 (leaf1). (b) At port 0xb00 (spine2) towards port 0xb00 (leaf2). (c) At port 0xc00 (spine2) towards port 0xb00 (leaf3). (d) At port 0xd00 (spine2) towards port 0xb00 (leaf4).**

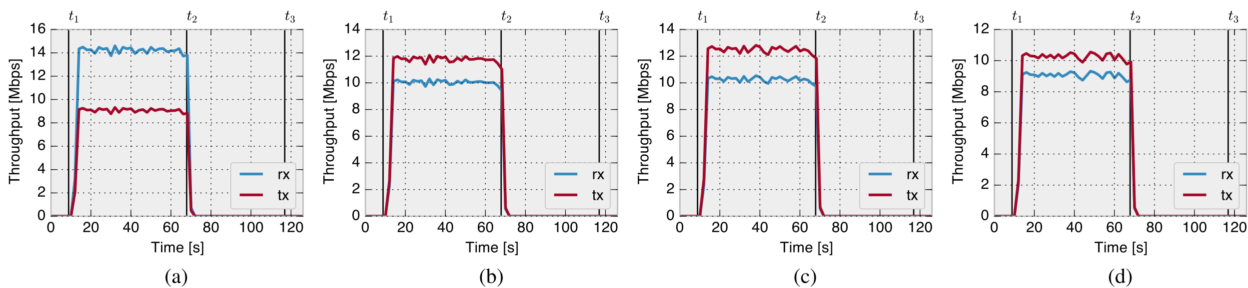


FIGURE 9. **Throughput measured at ports on switch spine1 (proposed). (a) At port 0xa00 (spine1) towards port 0xa00 (leaf1). (b) At port 0xb00 (spine1) towards port 0xa00 (leaf2). (c) At port 0xc00 (spine1) towards port 0xa00 (leaf3). (d) At port 0xd00 (spine1) towards port 0xa00 (leaf4).**

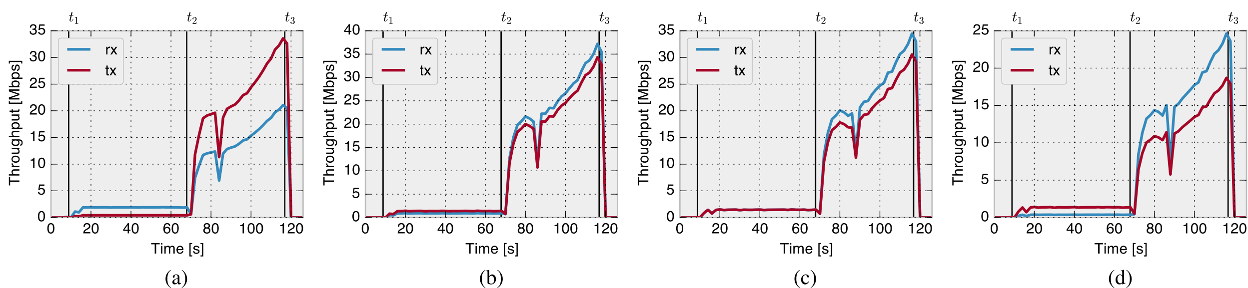


FIGURE 10. **Throughput measured at ports on switch spine2 (proposed). (a) At port 0xa00 (spine2) towards port 0xb00 (leaf1). (b) At port 0xb00 (spine2) towards port 0xb00 (leaf2). (c) At port 0xc00 (spine2) towards port 0xb00 (leaf3). (d) At port 0xd00 (spine2) towards port 0xb00 (leaf4).**

synchronization between throughput change and event occurrences was made on the basis of the timestamp. At t_1 where MPI_Bcast started, both RX and TX throughput observed at the ports of spine1 rise steeply and then maintain the amount of approximately 14Mbps and 9Mbps, respectively, while there is no clear growth of throughput at the ports

of spine2. This 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 synchronized with the execution of the MPI application.

C. 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 computing 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, we measured the communication performance of point-to-point MPI primitives using the OSU Micro-Benchmark Suite 5.3 [14] between node01 and node02 and compared the result with and without the proposed mechanism. The reason to measure the performance of point-to-point communication and not collective 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 11 shows a comparison of the throughput observed between node01 and node02. Fig. 12 shows the comparison result of latency for the same computing node pair. The plots in Fig. 11 and Fig. 12 represent the average of 500 measurements and 50,000 measurements, respectively. Fig. 13 shows the overhead imposed to latency by the proposed mechanism. These plots indicate that performance degradation imposed by the proposed mechanism is practically negligible for both bandwidth and latency.

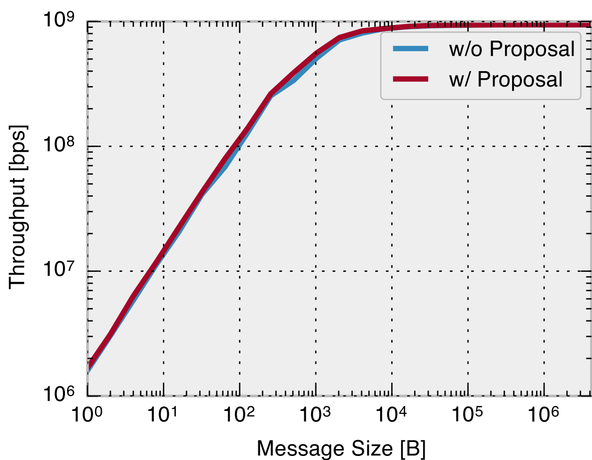


FIGURE 11. Comparison of bandwidth.

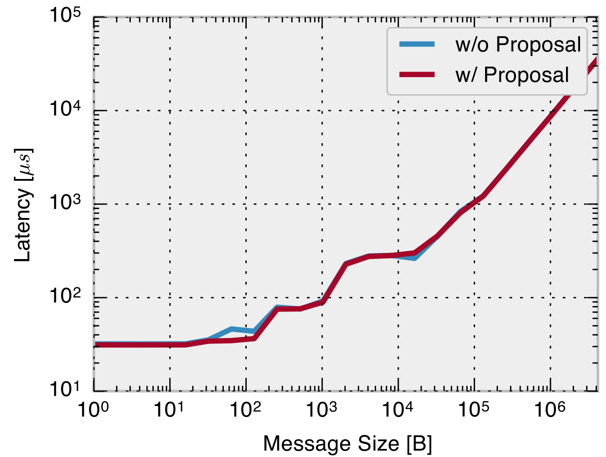


FIGURE 12. Comparison of latency.

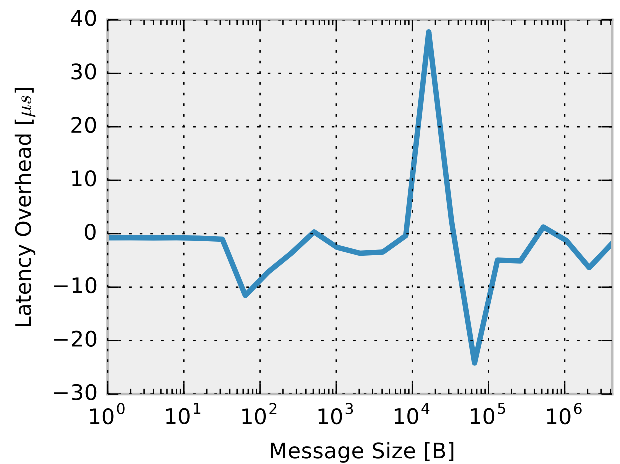


FIGURE 13. Absolute overhead to latency.

In this experiment, we only evaluated the performance of point-to-point primitives. However, the fact that collective primitives are often implemented using multiple point-to-point primitives [6], [10] implies that the overhead of the proposed mechanism is negligible for collective primitives as well.

V. RELATED WORK

Several studies [3], [12] have been carried out to incorporate application-awareness into SDN. An extension to Open vSwitch and OpenFlow has been proposed [12] 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 were out of the focus. The packet processing pipelines of commercial hardware switches cannot be modified since they are implemented using unmodifiable hardware. In contrast, our research supports existing hardware switches and per-packet inspection.

An application-aware routing scheme for big data applications has been presented in [3]. 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 traffic 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) [9] 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. Our research 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 [1]. This mechanism offloads collective MPI primitives to programmable NICs and OpenFlow switches. This method heavily depends on specialized hardware such as NetFPGA, whereas our proposal is software-based.

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

VI. CONCLUSION

In this paper, we have 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 computing 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 [4], [19] 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 we have developed some individual SDN-enhanced MPI primitives and UnisonFlow as the coordination mechanism of message-passing communication and computation, it is still unclear how these elements can accelerate a practical application as a whole. Finally, we need to investigate how this architecture can be adopted to a computer cluster simultaneously running multiple jobs, which is common in practical deployments. Since our current implementation of the UnisonFlow assumes only one job running at the same time on a node, we need to enhance it to support multiple concurrent jobs. This enhancement might involve an integration with the job scheduler.

REFERENCES

[1] O. Arap, G. Brown, B. Himebaugh, and M. Swany, "Software defined multicasting for MPI collective operation offloading with the NetFPGA," in *Euro-Par 2014 Parallel Processing* (Lecture Notes in Computer Science), vol. 8632. Cham, Switzerland: Springer, 2014, pp. 632–643.

[2] I. Bird, "Computing for the large hadron collider," *Annu. Rev. Nucl. Particle Sci.*, vol. 61, no. 1, pp. 99–118, 2011.

[3] L.-W. Cheng and S.-Y. Wang, "Application-aware SDN routing for big data networking," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2015, pp. 1–6.

[4] K. Dashdavaa *et al.*, "Architecture of a high-speed MPI_bcast leveraging software-defined network," in *Euro-Par 2013: Parallel Processing Workshops* (Lecture Notes in Computer Science), vol. 8374. Berlin, Germany: Springer, 2014, pp. 885–894.

[5] W. Gropp, "MPICH2: A new start for MPI implementations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface* (Lecture Notes in Computer Science), vol. 2474. Berlin, Germany: Springer, 2002, p. 7.

[6] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda, "Design of high performance MVAPICH2: MPI2 over InfiniBand," in *Proc. 6th IEEE Int. Symp. Cluster Comput. Grid (CCGRID)*, May 2006, pp. 43–48.

[7] *InfiniBand Architecture Specification Release 1.3*, InfiniBand Trade Association, Beaverton, OR, USA, 2015.

[8] S. Kamil, L. Oliker, A. Pinar, and J. Shalf, "Communication requirements and interconnect optimization for high-end scientific applications," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 2, pp. 188–202, Feb. 2010.

[9] S. Kamil, A. Pinar, D. Gunter, M. Lijewski, L. Oliker, and J. Shalf, "Reconfigurable hybrid interconnection for static and dynamic scientific applications," in *Proc. 4th Int. Conf. Comput. Frontiers (CF)*, May 2007, pp. 183–194.

[10] 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*. Norwell, MA, USA: Kluwer, 2005, pp. 167–185.

[11] N. McKeown *et al.*, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Apr. 2008.

[12] H. Mekky, F. Hao, S. Mukherjee, Z.-L. Zhang, and T. V. Lakshman, "Application-aware data plane processing in SDN," in *Proc. 3rd Workshop Hot Topics Softw. Defined Netw. (HotSDN)*, 2014, pp. 13–18.

[13] M. P. Forum. (2012). *MPI: A Message-Passing Interface Standard*. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

[14] OOhio State University. (2016). *MVAPICH :: Benchmarks*. [Online]. Available: http://mvapich.cse.ohio-state.edu/benchmarks/

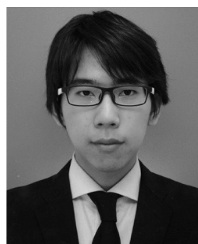
[15] R. Rabenseifner, "Optimization of collective reduction operations," in *Computational Science—ICCS* (Lecture Notes in Computer Science), vol. 3036. Berlin, Germany: Springer, 2004, pp. 1–9.

[16] R. Rosen, *Linux Kernel Networking: Implementation and Theory*. New York, NY, USA: Apress, 2013.

[17] Ryu SDN Framework Community. (2014). *Ryu SDN Framework*. [Online]. Available: https://osrg.github.io/ryu/

[18] K. Takahashi *et al.*, "Concept and design of SDN-enhanced MPI framework," in *Proc. 4th Eur. Workshop Softw. Defined Netw. (EWSDN)*, Sep./Oct. 2015, pp. 109–110.

[19] 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 *Proc. Int. Conf. High Perform. Comput. Simulation (HPCS)*, Jul. 2014, pp. 784–792.



KEICHI TAKAHASHI (S'15) received the M.Eng. degree from Osaka University in 2016, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Science and Technology. His main research interests include high-performance computing and parallel distributed computing. He is a Student Member of ACM and IPSJ.

SUSUMU DATE (M'08) received the B.E., M.E., and Ph.D. degrees from Osaka University in 1997, 2000, and 2002, respectively. He was an Assistant Professor with the Graduate School of Information Science and Technology, Osaka University, from 2002 to 2005. He was also a Visiting Scholar with the University of California at San Diego, San Diego, in 2005. From 2005 to 2008, he was involved in the internationalization of education with the Graduate School of Information Science and Technology, Osaka University, as a specially appointed Associate Professor. Since 2008, he has been an Associate Professor with the Cybermedia Center, Osaka University. He is currently an Associate Professor with the Cybermedia Center, Osaka University. His research field is computer science, and his current research interests include cloud, cluster, grid, high-performance computing, and their applications. He is a member of IPSJ.

DASHDAVAA KHURELTULGA received the M.Eng. degree from Osaka University in 2015, where he is currently pursuing the Ph.D. degree with the Graduate School of Information Science and Technology.

YOSHIYUKI KIDO (M'06) received the Ph.D. degree from Osaka University in 2008. He was a specially appointed Assistant Professor with the Center for Advanced Medical Engineering and Informatics, Osaka University, from 2008 to 2011, and with the Graduate School of Information Science and Technology, Osaka University, from 2011 to 2012. He was involved in RIKEN, the HPCI Program for Computational Life Science from 2012 to 2013. He became a specially-appointed Associate Professor with the Cybermedia Center, Osaka University, in 2013, where he is currently an Associate Professor. His research field is computer science with bioscience, and his current research interests include visualization, software-defined networking, cluster computing, and related information technologies. He is a member of CS and IPSJ.

HIROAKI YAMANAKA (M'17) received the M.E. and Ph.D. degrees from Osaka University in 2008 and 2011, respectively. Since 2011, he has been a Researcher with the National Institute of Information and Communications Technology. He was involved in the research of network virtualization and SDN technologies. His current research interests include edge computing and IoT technologies.

EIJI KAWAI (M'99) received the Ph.D. degree in information systems from the Nara Institute of Science and Technology (NAIST) in 2001. From 2000 to 2003, he was an Awarded Researcher with the Japan Science and Technology Corporation. From 2003 to 2009, he was with the Graduate School of Information Science, NAIST, as an Assistant Professor and an Associate Professor. In 2009, he joined the National Institute of Information and Communications Technology, where he is currently the Director of the ICT Testbed Research, Development and Operations Laboratory.

SHINJI SHIMOJO (M'03) received the M.E. and Ph.D. degrees from Osaka University in 1983 and 1986, respectively. He was an Assistant Professor with the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, in1986, where he was an Associate Professor with the Computation Center from 1991 to 1998. From 1991 to 1998, he was also a Visiting Researcher with the University of California at Irvine, Irvine, for a year. He has been a Professor with the Cybermedia Center (then the Computation Center), Osaka University, since 1998. From 2008 to 2011, he was an Executive Researcher and the Director of the Network Testbed Research and Development Promotion Center, National Institute of Information and Communications Technology. He is currently the Director of the Cybermedia Center. His current research interests include a wide variety of multimedia applications, peer-to-peer communication networks, ubiquitous network systems, and IoT systems. He is a fellow of IEICE and IPSJ. He is a Founding Member of PRAGMA and CENTRA. He received the Osaka Science Prize in 2005 and an award by the Minister of Internal affair on 2017.