A Study on Acceleration of Staging Operations Using Dynamic Traffic Control on Interconnect

Submitted to
Graduate School of Information Science and Technology
Osaka University

April 2021

Arata ENDO

This work is dedicated to my parents.

List of Publications by the Author

Journal

[1] Arata Endo, Hiroki Ohtsuji, Erika Hayashi, Eiji Yoshida, Chunghan Lee, Susumu Date, and Shinji Shimojo, "Dynamic Traffic Control of Staging Traffic on the Interconnect of the HPC Cluster System", *IEEE Access*, vol. 8, pp. 198 518–198 531, Nov. 2020. DOI: 10.1109/ACCESS.2020.3035158.

International Conference (with review)

[1] Kazuya Ishida, Daiki Asao, Arata Endo, Yoshiyuki Kido, Susumu Date, and Shinji Shimojo, "High-Resolution Streaming Functionality in SAGE2 Screen Sharing", in *Proceedings of the 2nd Future of Information and Communications Conference*, Mar. 2019, pp. 384–399. DOI: 10.1007/978-3-030-12385-7_30.

[2] Arata Endo, Ryoichi Jingai, Susumu Date, Yoshiyuki Kido, and Shinji Shimojo, "Evaluation of SDN-based Conflict Avoidance between Data Staging and Inter-Process Communication", in *Proceedings of the 15th International Conference on High Performance Computing & Simulation*, Jul. 2017, pp. 267–273. DOI: 10.1109/HPCS.2017.48.

[3] Arata Endo, Yoshiyuki Kido, Susumu Date, Yasuhiro Watashiba, Kiyoshi Kiyokawa, Haruo Takemura, and Shinji Shimojo, "Improvement of Scalability in Sharing Visualization Contents for Heterogeneous Display Environments", in *Proceedings of the 13th International Symposium on Grids and Clouds*, Jan. 2017, p. 009. DOI: 10.22323/1.270.0009.

Internal Conference (without review)

[1] Arata Endo, Chunghan Lee, Susumu Date, Yoshiyuki Kido, Yasuhiro Watashiba, and Shinji Shimojo, "Packet-Flow Analysis Tool to Automatically Extract Traffic Collision Derived from Staging Operation", in *the 17th Dependable System Workshop*, (Japanese), Dec. 2019, pp. 1–8.

Oral Presentation

[1] Arata Endo, Chunghan Lee, Susumu Date, Yasuhiro Watashiba, Yoshiyuki Kido, and Shinji Shimojo, "Evaluation of SDN-based Conflict Avoidance between Inter-Node Communication and Staging Communication Based on Packet Monitoring", in *the 36th Pacific Rim Application and Grid Middleware Assembly Workshop*, Apr. 2019.

[2] Arata Endo, Yasuhiro Watashiba, Yoshiyuki Kido, Susumu Date, Kiyoshi Kiyokawa, and Haruo Takemura, "Improvement of Scalability in Sharing Application Screens between Tiled Display Walls", in *the 28th Pacific Rim Application and Grid Middleware Assembly Workshop*, Apr. 2015.

[3] Arata Endo, Yoshiyuki Kido, Susumu Date, Yasuhiro Watashiba, Kiyoshi Kiyokawa, Haruo Takemura, and Shinji Shimojo, "An Implementation of SAGE Bridge for Sharing Visualized Contents on Multiple Tiled Display Wall Systems", in *the 29th Pacific Rim Application and Grid Middleware Assembly Workshop*, Oct. 2015.

Summary

Today's high-performance computing (HPC) systems are mostly built based on cluster architecture where multiple computing nodes perform large-scale computation in parallel. An HPC cluster system is composed of computing nodes connected on an interconnect and a parallel file system (PFS) that provides users with higher I/O performance. Importantly, on a recent HPC cluster system composed of more than thousands of computing nodes, communication performance and I/O performance took on a role of more importance as well as the computing performance itself of the HPC cluster system.

The modern HPC cluster systems often adopt a two-layered file system composed of a PFS (global file system) and a file system constructed on each computing node (local file system). On the HPC cluster systems with a two-layered file system, a staging operation moves input and output data between a local file system and a global file system before and after the computation of a scientific application. The rapid increase in the size of data treated in today's scientific research strongly requires the acceleration of the staging operation for job throughput improvement. From this perspective then, this dissertation tackles the acceleration of the staging operation focused on staging communication, by targeting HPC cluster systems with an oversubscribed fat-tree interconnect shared by computing nodes and storages.

To achieve the acceleration of this staging operation, this dissertation addresses the following three issues:

1. Revealing how the staging operation is slowed down on interconnect,

2. Exploring an approach for accelerating the staging operation, and

3. Verifying the practicality of the explored approach.

To address the first issue, Chapter 2 investigates how the staging operation is slowed down on the interconnect and how slowed staging operation degrades job throughput. For this investigation, this dissertation speculates that traffic collision between inter-node communication traffic and staging traffic prolongs the staging operation time on

the target class of HPC cluster systems. Verifying the correctness of this speculation requires a packet-monitoring that observes how the bandwidth consumed by each type of traffic changes in association with the staging operation. However, each job submission experiment and consumed bandwidth calculation included in the packet-monitoring takes a much longer time. Therefore, this dissertation develops a packet-monitoring tool that automatically performs the packet-monitoring for greater efficiency. This investigation using this packet-monitoring tool showed that the traffic collision increased the staging operation time by 45.2% and degraded the job throughput by 6.4%.

To address the second issue, Chapter 3 presents a dynamic traffic control method in conjunction with the staging operation as an approach to avoid traffic collision between inter-node communication traffic and staging traffic. This chapter first speculates that the static traffic control method, which is usually adopted in traditional interconnects, has difficulty in accelerating the staging operation without performance degradation in inter-node communication. Based on this speculation, as an approach, this dissertation adopts a dynamic traffic control method in conjunction with the staging operation, which switches different types of traffic controls depending on whether a staging operation occurs or not. This approach shows how to accelerate the staging operation with little performance degradation in inter-node communication. To verify the possibility that this approach can be realized, this dissertation proposes a dynamic link-separation collision avoidance method (DLSCA) that separates the links into two groups for each traffic type only while a staging operation occurs. The evaluation result showed that the DLSCA is feasible through an experiment performed in a virtual environment. Also, the evaluation showed that the DLSCA succeeded in shortening the staging operation time by 7.50% and the job execution time by 3.40%.

To address the third issue, Chapter 4 verifies whether the dynamic traffic control method in conjunction with the staging operation contributes to the acceleration of the staging operation and the job throughput improvement or not, by performing a job submission experiment on an actual HPC cluster system. Also, this chapter attempts to reinforce the DLSCA by considering that communication latency and delay in an actual environment are not ideal. As a result, this chapter proposes a progressive switching-based DLSCA (reinforced method), which is reinforced from the DLSCA in terms of the following two points. First, the reinforced method performs the path switching in a progressive way without blocking any traffic to reduce the negative influence of the longer path switching. Second, the reinforced method configures link-separation-based paths on the interconnect with fewer flow entries than the original so that the OpenFlow controller takes less time to

configure link-separation-based paths. The evaluation result showed that the progressive switching-based DLSCA can accelerate the staging operation by 22.0% with negligible overhead of the application and can improve 7.6% of the job throughput. Based on this result, the dynamic traffic control method in conjunction with the staging operation is considered practical for accelerating the staging operation with little performance degradation in inter-node communication and improving the job throughput.

Chapter 5 concludes this dissertation and discusses directions for future study.

Contents

List of Figures

1 Introduction

1.1 Background

1.1.1 High-Performance Computing

High-performance computing (HPC) has occupied an important position in scientific research. Today, in every research field, various scientific phenomena are simulated and analyzed on HPC systems. Recently, researchers have become enthusiastic about applying computationally intensive machine learning techniques to their own large amount of scientific data using HPC systems. In fact, scientific applications like the following examples are performed on HPC systems. First, climate simulations to forecast climate changes such as global warming future are conducted on HPC systems around the world [1, 2]. Second, a variety of molecular dynamics simulations for drug discovery are performed on HPC systems to reveal how chemical compound candidates for drug interact with target proteins causing diseases [3–5]. Third, researchers who work on natural language processing needs HPC systems to analyze a large amount of text data [6, 7]. Finally, researchers who work on materials science now leverage HPC systems to apply machine learning techniques to predict material property [8, 9].

The necessity of HPC can be partly explained from the proliferation and increase of measurement data due to advancements in measurement technology. The advancement of measurement techniques has improved the spatial and temporal resolution of measurement, which leads to a growth in the size of data obtained through such measurement devices. High-resolution measurement data in both time and space enables researchers to perform simulations and analysis in a more accurate and fine-grained manner. However, performing simulations and analysis using high-resolution measurement data simultaneously necessitates a huge amount of computation. For example, radio telescopes coordinated in the Event Horizon Telescope as a virtual telescope generated 2 PB of measurement data every night and the generation of black hole pictures in high resolution from such measurement data necessitated more than 2,000 cores on HPC systems [10, 11].

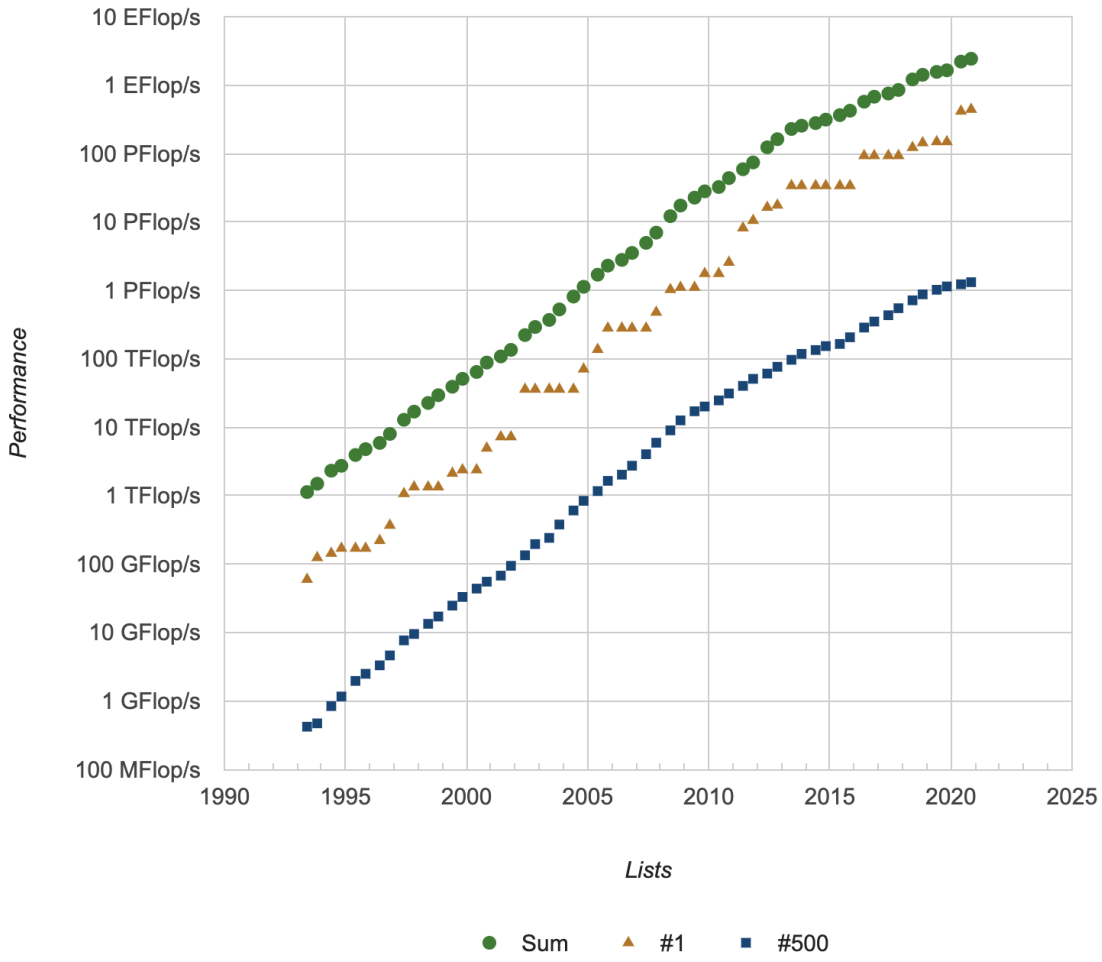


Figure 1.1: Performance development generated by Top500 [12].

The strong demand for higher computing performance from researchers has brought HPC systems into continuous performance improvement. As a result, the computing performance of HPC systems has grown over the past three decades. Figure 1.1 is a graph that illustrates the continuous performance improvement in HPC systems. This graph is generated by Top500 [12], which is a project to list the 500 most powerful HPC systems based on computing performance. This graph shows that the computing performance provided by the HPC systems has grown steadily. However, the graph also shows that improvement in computing performance has gradually slowed down since around 2013. The importance of HPC in scientific research and the rising expectations regarding high performance data analysis such as machine learning and deep learning are essential for the future advancement of science and technology. Therefore, continuous improvement in computing performance provided by HPC systems is crucial.

1.1.2 Cluster Architecture

Today's HPC systems are mostly built based on *cluster architecture* to provide massive computing performance for researchers by aggregating many computers into a single system. According to recent statistics by Top500, 93% of today's HPC systems listed in the Top500 has adopted this cluster architecture. Note that most of the top 35 (top 7%) of the Top500 list has adopted this cluster architecture. In the cluster architecture, multiple computers, referred to as *computing nodes*, are connected on low-latency and high-bandwidth networks. They are configured and coordinated so that large-scale computation is performed in parallel. Specifically, on a *HPC cluster system*, large-scale computation is divided to multiple computing nodes and each computing node performs the assigned computation. In most cases, computing nodes exchange data with each other to complete large-scale computation.

Figure 1.2 shows an overview of a typical HPC cluster system. The typical HPC cluster system is characterized by the following three points. First, this HPC cluster system is mainly composed of computing nodes, each of which is connected on a low latency and high-performance computer network referred to as an *interconnect*. Ethernet [13] and InfiniBand [14] are representative examples of interconnects for the modern HPC cluster systems. Figure 1.3 shows what type of interconnect is used in HPC cluster systems listed in the Top500. Each of the applications (also referred to as *jobs*) is composed of multiple parallel processes. The processes are distributed to multiple computing nodes and *inter-node communication*, which is communication for exchanging data and messages among the processes on each computing node, happens on the interconnect. In applications composed of multiple parallel processes, each process communicates with each other according to a *Message-Passing Interface* (MPI) [15], which is a *de facto* standard communication protocol for exchanging data and messages among the processes. The implementation of MPI has been developed as program libraries such as OpenMPI [16] and MPICH [17] and is used in the development of applications run on HPC cluster systems.

Second, the typical HPC cluster system owns a *management node* and a *job scheduler* such as Slurm [18], Portable Batch System (PBS) [19] and Torque [20] is deployed on the management node for the purpose of load-balancing and distribution of computational workload. By applying such job schedulers, the HPC cluster system provides the computing nodes as computing resources to multiple users in a sharing manner. Specifically, although there is a *time-sharing* manner that allows multiple jobs to share a single computing node at the same time, today's HPC cluster systems commonly often use a *space-sharing*

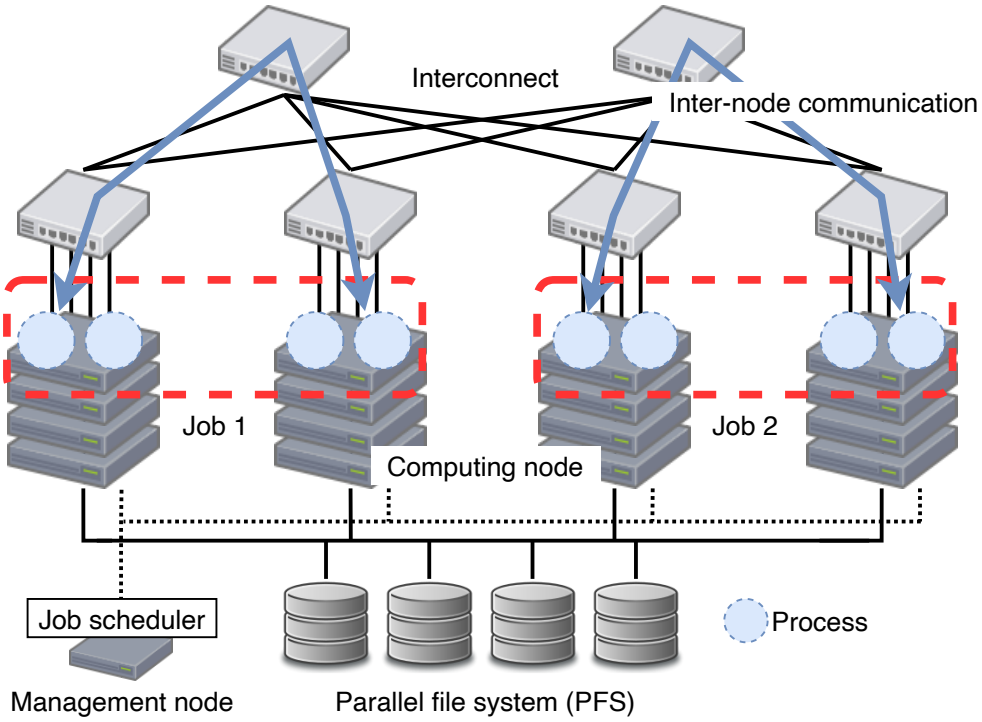


Figure 1.2: An overview of a typical HPC cluster system.

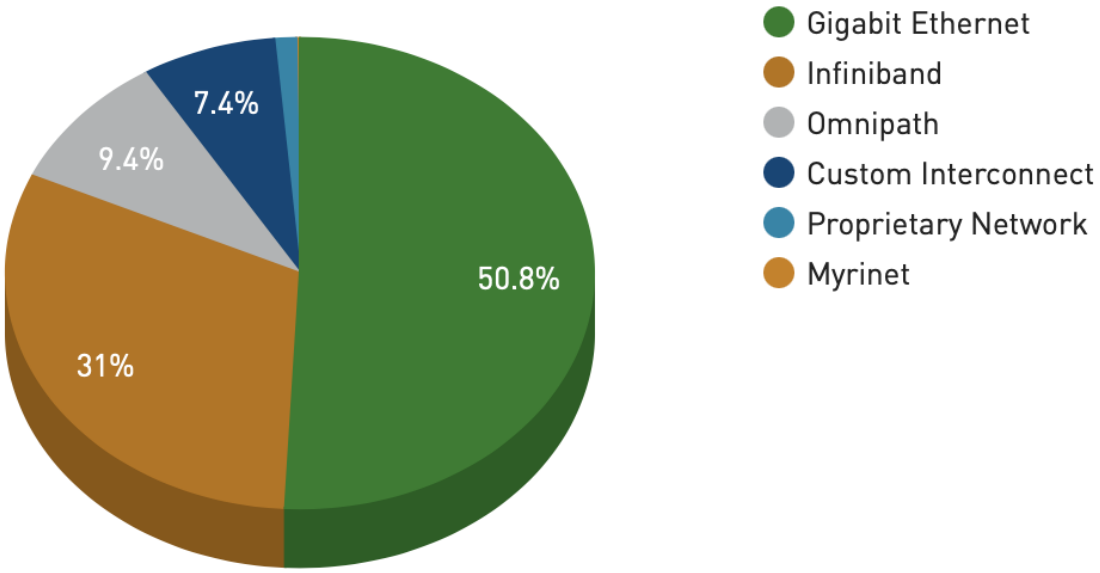


Figure 1.3: The share of interconnect adopted by HPC cluster systems generated by Top500 [12].

manner, where a computing node is used by a single job at the same time [21, 22]. The job scheduler works as follows on the HPC cluster system. When a user wants to run an application on the HPC cluster system, the user first needs to submit a job request to the job scheduler. The job request is written as a shell script, called a *job script*, which specifies how many computing nodes the user wants to use for computation and what application the user wants to perform in general. On receiving the job request, the job scheduler decides which computing nodes composing the HPC cluster system are selected for the job request, and actually allocates the selected computing nodes to the job request, finally launching multiple processes for the job request on the selected computing nodes.

Third, a parallel file system (PFS) is, in general, deployed for providing users with higher *I/O performance*, which is read and write performance to files and IOPS (Input/Output Per Second). Examples of PFS widely prevailing today include Lustre [23], Gluster [24], General Parallel File System (GPFS) [25] and BeeGFS [26]. A remarkable feature of PFS is that files are stored to different storages so that multiple processes running on the HPC cluster system can access PFS in parallel. PFS is configured to be accessed from all of computing nodes so that the user's application can write and read data from any computing node.

To measure the performance delivered by HPC cluster systems, *job throughput* is often used. Job throughput is a criterion to measure how many jobs can be processed in a time period by an HPC cluster system and to guarantee and deliver the highest job throughput. Although making the computing performance of a single processor higher contributes to the job throughput, the performance improvement of a single processor is predicted to be gradually slowed and almost stopped as indicated by the end of Moore's law [27]. Therefore, today's HPC cluster system cannot help increase the number of computing nodes (scale-out) to achieve job throughput improvement. Fugaku, which is the most powerful HPC cluster system in Japan, is a cluster system of 158,976 computing nodes, each of which has a Fujitsu A64 processor whose peak performance is 3.4 TFlop/s, for gaining 514 PFlop/s as the theoretical peak performance of Fugaku [28].

Importantly, on the recent HPC cluster systems, especially HPC cluster systems composed of more than thousands of computing nodes, communication performance and I/O performance took on a role of more importance as well as the computing performance itself of the HPC cluster system. For example, the poor performance of the inter-node communication caused a longer data exchange time among processes over the interconnect. As a result, the total computation time of an application is increased because multiple processes composing the application must be blocked until the data exchange is finished.

Also, the degradation of the I/O performance results in the longer application's I/O operation to PFS. In many cases, an application reads data needed for its computation, or *input data* from PFS to memory, and writes a computation result, or *output data* from memory to PFS. Because the read and write operations are performed during the computation, the application is forced to wait until the completion of the I/O operation, which increases the total computation time. Therefore, the communication and I/O performance have become more important in determining the job throughput performance on today's HPC systems.

1.1.3 Two-Layered File System and Staging Operation

Two-Layered File System

The typical HPC cluster system with PFS does not always satisfy both storage capacity and I/O performance requirements as needed by researchers. As described in Section 1.1.2, today's PFS can provide a petabyte-scale storage capacity and also deliver higher I/O performance, by structuring a lot of hard disk drives (HDD) in redundant arrays of inexpensive disks (RAID). However, specific classes of applications, for example, applications that create or access a large number of small-sized files rather than a small number of large-sized data, require higher I/O performance than the I/O performance delivered by the HPC cluster system with PFS. To meet such requirements to higher I/O performance, it has been appeared in the market to use faster storage media, or solid-state drive (SSD), for PFS than HDD. However, in reality, SSD-based PFS is much more expensive than HDD-based PFS and so it is difficult for SSD-based PFS to satisfy the large storage capacity requirement. For this reason, HPC cluster systems often adopt a *two-layered file system* composed of HDD-based PFS and a file system constructed on SSD attached to each computing node. For example, TSUBAME 3.0 at the Tokyo Institute of Technology [29] and Summit at the Oak Ridge National Laboratory (ORNL) [30] adopt a two-layered file system.

Figure 1.4 shows an example of the HPC cluster system with a two-layered file system. This HPC cluster system deploys PFS as a global file system to provide large storage capacity like the typical HPC cluster system described in Section 1.1.2. Also, in this HPC cluster system, an SSD is deployed on each computing node and a file system constructed on the SSD, called a *local file system*, delivers high I/O performance for applications. Where the input and output data of an application are located is coordinated on this HPC cluster system so that the application can take advantage of these two types of file systems.

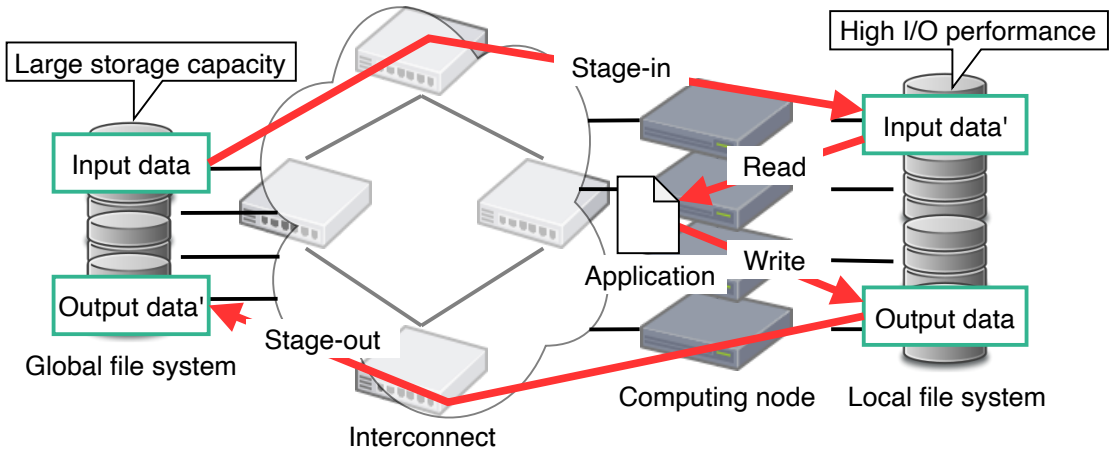


Figure 1.4: An example of the HPC cluster system with a two-layered file system.

To provide a large storage capacity in the global file system, this HPC cluster system places the input and output data on the global file system before and after computation of the application. To deliver high I/O performance, this HPC cluster system places the input and output data on a local file system during computation. As a result, the application can access the input data and create the output data on the local file system at high speed.

Staging Operation

As described above, in the HPC cluster system with a two-layered file system, the input and output data of the application is located on a local file system during the computation of the application. Also, the input data is located on a global file system before the computation and the output data is located on the global file system after the computation. To realize this data placement, the data needs to be moved between local file systems and global file system (*staging operation*). In detail, as shown in Fig. 1.4, the input data is staged from the global file system to a local file system before the computation (*stage-in*) and the output data is staged from a local file system to the global file system after the computation (*stage-out*). A staging operation is generally performed by the job scheduler such as Slurm and PBS [31, 32].

Figure 1.5 diagrams the breakdown of a job execution, which starts when computing nodes are actually allocated to the job, on the HPC cluster system with a two-layered file system. As shown in this figure, job execution is composed of a stage-in operation, stage-out operation, and computation of an application. As explained in the last paragraph, the stage-in operation performs an I/O operation before the computation, and the stage-out

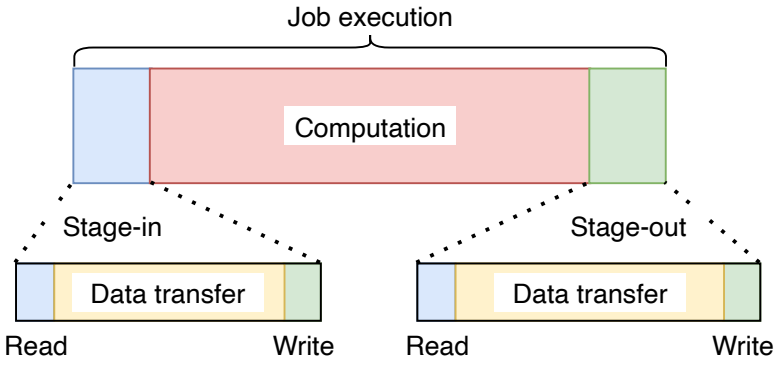


Figure 1.5: The breakdown of job execution in the HPC cluster system with a two-layered file system.

operation performs an I/O operation after the computation. In detail, the stage-in operation in a job execution is performed after computing nodes are actually allocated to the job and before the computation. In the former stage-in operation, input data is read on the global file system, and the input data is transferred to the memory on a computing node and then written to the local file system on the computing node. Likewise, in the latter stage-out operation, output data on a local file system is read to the memory, and the output data is transferred from the memory to the global file system and then written to the global file system.

Two factors decide the execution time of the staging operation described above. The first factor is the performance of read and write operations on the local file system and global file system. The second factor is the performance of the communication generated by the staging operation, called *staging communication*. The staging communication generates traffic called *staging traffic* to transfer data between a local file system and the global file system. When the staging traffic cannot use the bandwidth available on interconnect fully, the time required for the data transfer increases, which results in a longer staging operation.

1.1.4 Acceleration of Staging Operation

To achieve improvement in job throughput on the HPC cluster system with a two-layered file system, the staging operation takes on a role of great importance. As mentioned in Section 1.1.3, a staging operation is executed before and after the computation of the application requested in a job. This fact means that the long stage-in operation results in the application waiting for the completion of the stage-in operation and that the long

stage-out operation results in the job waiting for the completion of the stage-out operation. Although the staging operation has a relatively small impact on job throughput at first glance because the staging operation time is generally shorter than the computation time, the impact of the staging operation on the job throughput becomes larger due to the rapid increase in size of data treated in today's scientific research. This circumstance strongly requires the acceleration of the staging operation for faster job execution.

Also, as described in Section 1.1.3, the acceleration of staging operations needs the performance improvement of I/O operation and communication generated by the staging operation. Although the acceleration of the staging operation is important for the above reason, to the author's knowledge, there are few representative studies that discuss the acceleration of the staging operation on the HPC cluster system with a two-layered file system. Moreover, the few representative studies, such as Burst Buffer [33], GekkoFS [34] and CatWalk-ROMIO [35] focus on the improvement in the I/O performance, and studies to improve the communication performance have not been found. Therefore, this dissertation tackles the acceleration of staging operations by focusing on the staging communication.

1.2 Target Class of HPC Cluster Systems

This dissertation targets the HPC cluster systems with the oversubscribed fat-tree interconnect shared by computing nodes and storages for the acceleration of staging operations. An example of such a cluster system is shown in Fig. 1.6. In the target class of HPC cluster systems, it is assumed that a two-layered file system is adopted. The two-layered file system is composed of a local file system accessible on a computing node and a PFS accessible from every computing node. It is also assumed that a PFS is constructed on storages and a file system is constructed on a high-performance storage such as SSD on a computing node as a local file system. As to the interconnect, this study assumes an oversubscribed fat-tree where the bisection bandwidth is smaller than half of injection bandwidth. This reason is explained from the fact that the oversubscribed fat-tree is often adopted by today's HPC cluster systems. Note that this study assumes that the job scheduler of the target class of HPC cluster systems adopts a space-sharing manner which is commonly used.

A fat-tree [36] is a major topology adopted for the interconnect of HPC cluster systems. In a fat-tree interconnect, a top-level switch is called a *core switch*. On the other hand, a lowermost switch is called an *edge switch*, and the remains are called *aggregation switches*. In general, as shown in Fig. 1.7, the fat-tree interconnect is classified into two types in terms

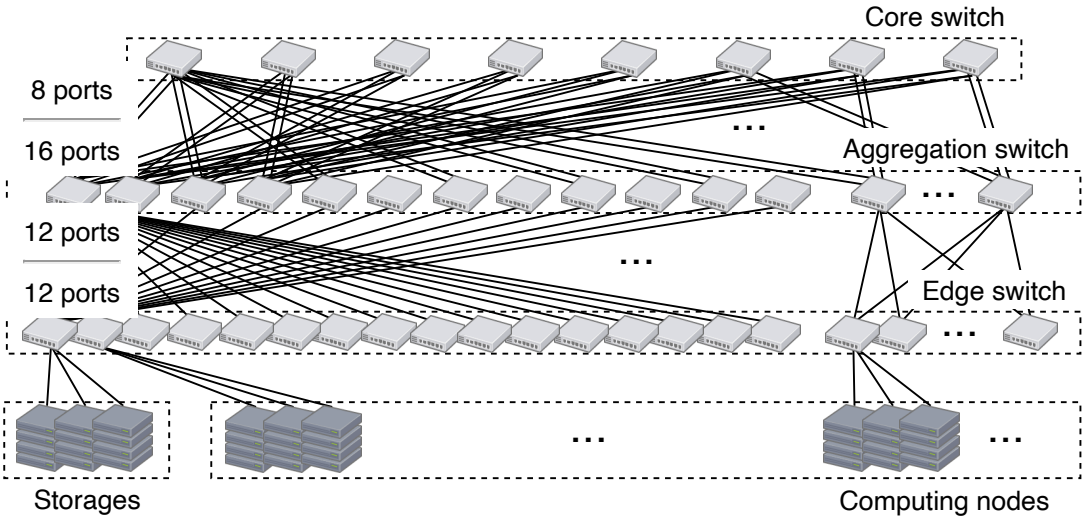
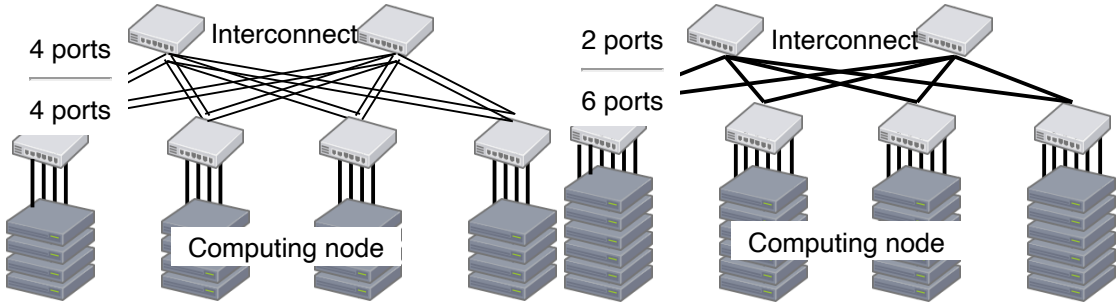


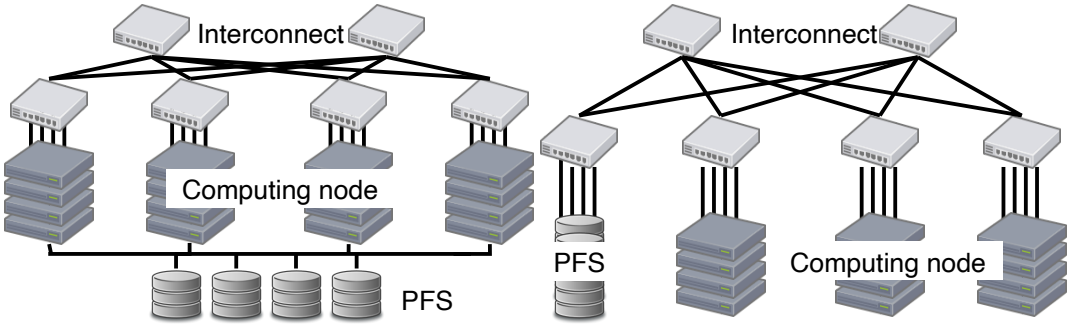
Figure 1.6: The HPC cluster system targeted in this study.

of available bandwidth: full-bisection and oversubscription. In a full-bisection fat-tree interconnect, the bisection bandwidth is half of the injection bandwidth in the cluster system, meaning that half of the processes running on computing nodes can communicate with the other half without network contention. Figure 1.7 (a) shows an example of full-bisection fat-tree interconnects where each switch has 8 ports. In this figure, the number of upper ports, which are ports used for connecting from an edge switch to core switches, is equal to the number of lower ports, which are ports used for connecting from an edge switch to computing nodes. Therefore, even if half of the computing nodes communicate with the other computing nodes, the traffic generated by each communication can use sufficient bandwidth. On the other hand, in the oversubscribed fat-tree interconnect, the bisection bandwidth is smaller than half of the injection bandwidth. Figure 1.7 (b) shows an example of oversubscribed fat-tree interconnects where each switch has 8 ports. In this figure, the ratio of the number of upper ports and the number of lower ports is 1:3. This ratio causes not enough bandwidth to exist for traffic to consume if half of the computing nodes communicate with the other computing nodes. However, this interconnect can hold more computing nodes than full-bisection fat-tree interconnects when both types of fat-tree interconnect are constructed at almost the same cost. In today's HPC cluster systems, the latter fat-tree interconnect is used more frequently than the former although the former can provide higher communication performance on the interconnect. For example, AI Bridging Cloud Infrastructure (ABCI) at the National Institute of Advanced Industrial Science and Technology (AIST) adopts an oversubscribed fat-tree [37]. A reason for this can be explained from the fact that the former costs more than the latter,



(a) A full-bisection fat-tree interconnect. (b) An oversubscribed fat-tree interconnect.

Figure 1.7: Examples of the two types of fat-tree interconnect.



(a) The non-shared-manner. (b) The shared-manner.

Figure 1.8: Two approaches to placing storages composing the PFS.

especially in the case of the larger HPC cluster systems.

There are two approaches to place storages composing the PFS on an HPC cluster system: shared manner and non-shared manner. Figure 1.8 shows HPC cluster systems adopting each approach to placing the storages. In a shared manner, the storages are placed on the same interconnect which connects the computing nodes together and the interconnect is shared by both of the computing nodes and the storages. On the other hand, in the non-shared manner, the storages are placed on a storage network, which is additionally built for the storages and connects between the computing nodes and the storages, and the interconnect is used only by the computing nodes. In today's HPC cluster systems, the former manner is often adopted for storage placement even though the latter manner provides higher I/O performance than the former manner. The reason for this is the same as the reason for the adoption of the oversubscribed fat-tree. The latter costs more than the former.

1.3 Research Objective

As described in Section 1.1.4, this dissertation tackles the acceleration of staging operations by focusing on staging communication to improve the job throughput. For this objective, the following three issues must be achieved:

1. Revealing how the staging operation is slowed down: Problems that degrade the performance in staging communication are not clear because there are few studies that investigate the staging communication in fine-grained situations. Therefore, to consider how to accelerate the staging operation, this study reveals how the staging operation is slowed down by observing the staging communication at the packet-level.

2. Exploring an approach for accelerating the staging operation: Since few existing methods tackle the acceleration of the staging operation as described in Section 1.1.4, this dissertation inevitably needs to explore a new approach for accelerating the staging operation. Also, for the same reason, it is not clear whether there is any approach that can accelerate the staging operation by focusing on staging communication or not. Therefore, with considerations of the cost to use or build an HPC cluster system, in a simulation environment, this dissertation verifies the possibility that the new approach can be realized first. For this purpose, this dissertation takes the strategy of proposing a prototype method, which resolves the problem revealed in the first issue following the new approach.

3. Verifying the practicality of the new approach explored in the second issue: The possibility that this new approach can be realized in a simulation environment is only theoretically verified. To show the practicality of this new approach, verifying whether this new approach can accelerate the staging operation on real HPC cluster systems is still needed.

1.4 Organization of the Dissertation

The organization of this dissertation is as follows. Chapter 2 quantitatively investigates how the staging operation is slowed down to address the first issue listed in Section 1.3. This dissertation speculates that the traffic collision between the inter-node communication traffic and the staging traffic slows the staging operation in an HPC cluster system with an oversubscribed fat-tree interconnect shared by computing nodes and the storages.

To verify the correctness of this speculation, a packet-monitoring is needed to observe how packets flow on the interconnect in a job submission. However, it is difficult to manually perform this verification because the measurement and analysis needed in packet-monitoring on an HPC cluster system take a much longer time. Therefore, this dissertation develops a packet-monitoring tool which automatically measures how each type of traffic consumes bandwidth on the interconnect through a job submission experiment. Using this packet-monitoring tool, this dissertation revealed that the traffic collision between each type of traffic slows the staging operation and slowed staging operation degrades the job throughput through a job submission experiment. Specifically, this investigation showed that the traffic collision increased the staging operation time by 45.2% and degraded the job throughput by 6.4%.

Chapter 3 addresses the second issue. This chapter explores an approach for avoiding traffic collision between inter-node communication traffic and staging traffic. This chapter first discusses whether the static traffic control method that traditional interconnects use can accelerate the staging operation. As a result, this dissertation takes a dynamic traffic control method in conjunction with the staging operation as an approach, which switches traffic control methods depending on whether any staging operation occurs. To verify the possibility that this approach can be realized, this dissertation proposes a prototype method, where the set of dedicated paths for each type of traffic are configured on the interconnect only while the staging operation occurs. The evaluation conducted in this dissertation showed that the proposed method is feasible through an experiment performed in a virtual simulation environment. Also, the evaluation showed that the proposed method succeeded in shortening the staging operation time by 7.50% and the job execution time by 3.40% as compared to a static traffic control method.

Chapter 4 addresses the third issue. This chapter investigates whether the dynamic traffic control method in conjunction with the staging operation can accelerate the staging operation and improve the job throughput on a real HPC cluster system. Because Chapter 3 implements the prototype method based on the dynamic traffic control method in conjunction with the staging operation assuming the simulation environment, this dissertation needs to transplant the prototype method onto a real HPC cluster system. In a real environment, communication delay and latency are not ideal, and may negatively affect the prototype method. Therefore, this dissertation reinforces the prototype method by considering the communication latency and delay. The reinforced prototype method is evaluated on a real HPC cluster system and the evaluation result shows that the reinforced prototype method is capable of avoiding the traffic collision between each type of traffic

and can accelerate the staging operation by 22.0%. Also, the evaluation indicates the overhead of the application incurred by the reinforced method is negligible. Furthermore, it is shown that 7.6% of the job throughput is improved by the reinforced method. Based on this result, the dynamic traffic control method in conjunction with the staging operation is considered practical for accelerating the staging operation with little performance degradation in inter-node communication and improving the job throughput.

Chapter 5 concludes this dissertation and discusses future directions.

2 Investigation on Traffic Collision using a Packet-Monitoring Tool

2.1 Introduction

As described in Chapter 1, this dissertation discusses the acceleration of the staging operation for the job throughput improvement. This chapter investigates how the staging operation is slowed down and how slowed staging operation degrades the job throughput and then clarifies the technical problems and issues to be achieved in this dissertation.

For the investigation, this chapter follows the two procedures below:

1. Speculation: How the staging operation is slowed down is speculated through the observation and review of the staging operation on an HPC cluster system.

2. Verification: The correctness of the speculation is quantitatively verified through an actual measurement experiment.

To investigate and understand how the staging operation is slowed down on an oversubscribed fat-tree interconnect shared by computing nodes and storages, it is essential to observe and analyze how packets flow on the interconnect, or *packet-monitoring*. However, there is no existing packet-monitoring tool that automates packet-monitoring for this study. Therefore, this study first develops a packet-monitoring tool suitable for investigating the staging traffic on the interconnect for this purpose. Next, the author investigates how the staging operation is slowed down on an HPC cluster system by performing a job submission experiment using the developed tool. In a job submission experiment, multiple job requests are submitted to an HPC cluster system to reproduce the situation where the HPC cluster system provides computing nodes for multiple users in a sharing manner.

The structure of this chapter is as follows: Section 2.2 speculates how the staging operation is slowed down on the interconnect. Section 2.3 describes the functionalities required for a packet-monitoring tool after clarifying the requirements to satisfy the

tool suitable for observing and analyzing how packets flow on the interconnect on an HPC cluster system. Section 2.4 explains the packet-monitoring tool developed in this study. Section 2.5 quantitatively verifies the speculation using the packet-monitoring tool. Section 2.6 clarifies the technical problem and issues to be achieved in this dissertation, and discusses future works. Section 2.7 concludes this chapter.

2.2 Speculation

This section speculates how the staging operation is slowed down on the interconnect. As described in Section 1.1.4, this dissertation focuses on staging communication to improve the job throughput on the target class of HPC cluster systems. Therefore, for this purpose, and by taking how traffic flows on the interconnect into consideration, this section reviews and observes how the staging operation works on the HPC cluster system targeted in this study.

To understand how the staging operation is slowed down, the following two facts affecting how traffic flows on the interconnect have to be considered in addition to the premise that this dissertation targets HPC cluster systems with an oversubscribed fat-tree interconnect shared by computing nodes and storages as described in Section 1.2. First, the job scheduler provides computing resources to multiple users in a sharing manner as described in Section 1.2, which results in sharing the interconnect among multiple processes. In short, multiple jobs work simultaneously on an HPC cluster system, meaning that the inter-node communication traffic is supposed to exist almost all the time on the interconnect. Second, job execution proceeds in the order of the stage-in operation, computation and the stage-out operation as described in Section 1.1.3. Considering the execution of multiple jobs on an HPC cluster system, some jobs proceed with computation and other jobs wait for the completion of the staging operation. Under these considerations, inter-node communication traffic and staging traffic co-exist on the interconnect because computing nodes and storages share the same interconnect. Moreover, both types of traffic may interfere with each other, called *traffic collision*, in the target class of HPC cluster systems described in Section 1.2. As a result, the staging communication takes a long time because of insufficient bandwidth on the oversubscribed fat-tree interconnect. This result causes longer job execution which includes the execution of the staging operation and then degrades the job throughput.

Figure 2.1 illustrates an example of these situations where a traffic collision can possibly happen on an HPC cluster system. The interconnect of the HPC cluster system is

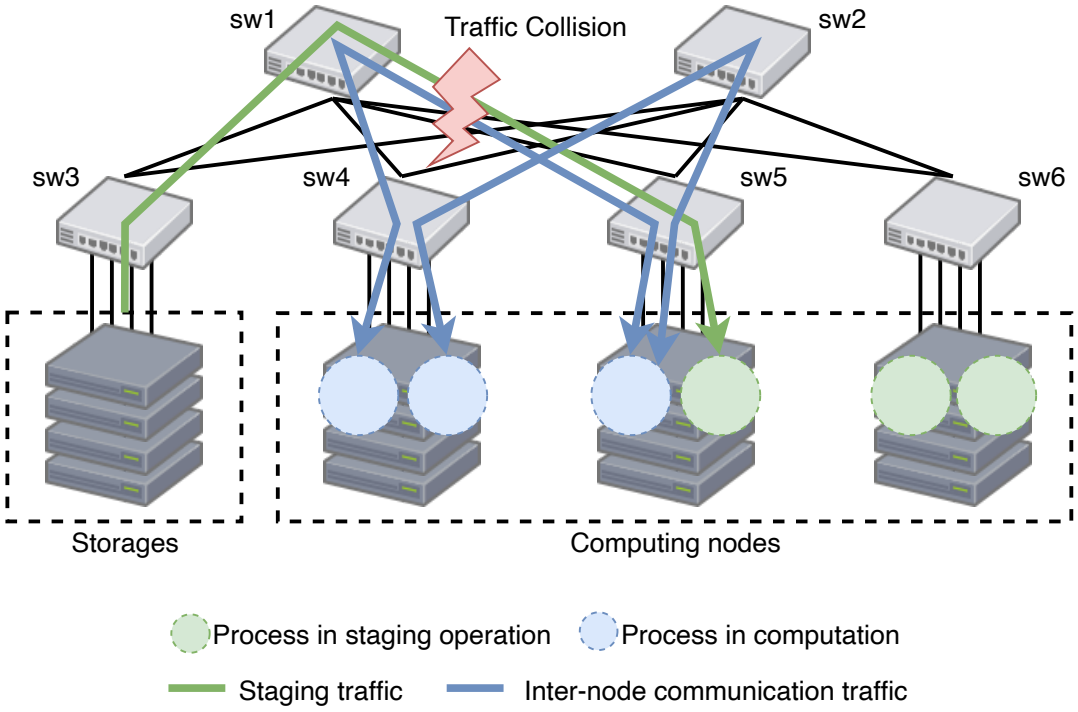


Figure 2.1: An example of a traffic collision between inter-node communication traffic and staging traffic on the target HPC cluster system.

composed of *sw1*-*sw6*. It is assumed that the job scheduler distributes the three processes (blue processes) for a job to the computing nodes connected to sw4 and sw5 and distributes the three processes (green processes) for the other to the computing nodes connected to sw5 and sw6. Under the assumption of these process distributions, now suppose that the former processes perform inter-node communication, and the latter ones wait for the completion of a staging operation. Then, the inter-node communication traffic among the former ones follows the path through sw4, sw1 and sw5, and one through sw4, sw2 and sw5. On the other hand, the staging traffic generated by the latter processes passes through sw3, sw1 and sw5. In this situation, both types of traffic share the link between sw1 and sw5. As a result, both types of traffic may block each other on the link between sw1 and sw5.

2.3 Functionalities Required for the Packet-Monitoring Tool

To investigate and understand how the staging operation is slowed down on the interconnect, the author conducts a job submission experiment, which requires a packet-monitoring tool

that allows us to observe the traffic on the interconnect on a per-packet-flow basis. For this purpose, this section summarizes what suitable functionalities need to be equipped on the packet-monitoring tool for this investigation.

2.3.1 Requirements for the Packet-Monitoring Tool

The packet-monitoring tool required in this investigation has to provide us with a way to observe how the traffic collision between each type of traffic happens. An essential piece of information for the observation is the bandwidth consumed by each type of traffic while a staging operation is running. To allow us to precisely measure and learn how much bandwidth each type of traffic consumes as a network resource, the packet-monitoring tool must facilitate three procedures: *packet-capture*, *consumed bandwidth calculation* and *visualization*.

1. Packet-capture procedure: Packet flowing on links connected to core switches, and not packet flowing on network interfaces of the computing nodes, have to be captured during a job submission experiment. This is because the traffic collision can happen on the links connecting to a core switch on the HPC cluster system targeted in this dissertation. Also, this packet-capture procedure must be performed to all links connected to core switches in parallel since in this investigation bandwidth is consumed on all links where the traffic collision can occur.

2. Consumed bandwidth calculation procedure: The tool must categorize a series of packets captured by the packet-capture procedure into two traffic types, inter-node communication traffic and staging traffic, and then calculate the consumed bandwidth for each traffic type. In this study, a series of packets captured by the packet-capture procedure is called *packet data*.

3. Visualization procedure: To understand and learn how the traffic changes in terms of bandwidth over time, some intuitive visualization of the consumed bandwidth is required. In this visualization, the execution time of a staging operation has to be associated with the information on how the staging traffic generated by the staging operation flows on the interconnect. For this purpose, it is needed to distinguish which staging traffic is generated by a staging operation because multiple staging operations generate staging traffic during a job submission experiment. However, this distinction is difficult because multiple jobs use the same computing nodes at different times and consequently, packets with almost the same packet header are

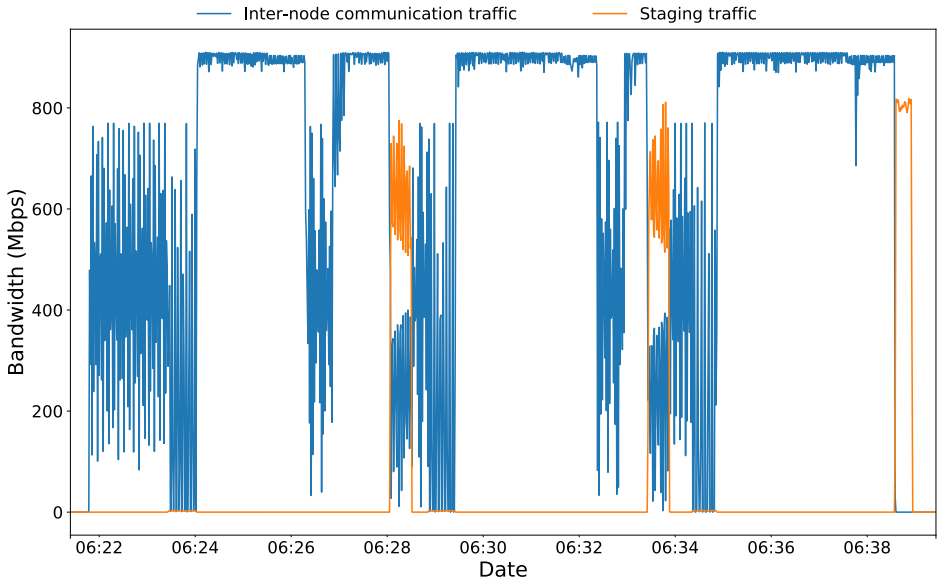


Figure 2.2: An example of a simple visualization that is a line chart of the consumed bandwidth given by the consumed bandwidth calculation procedure.

generated by different staging operations. For this reason, a simple visualization where the consumed bandwidth given by the consumed bandwidth calculation procedure is simply plotted is less useful for us. For example, Fig. 2.2 is an example of such a simple visualization. In this graph, it is difficult to observe which staging traffic is generated by a staging operation. Therefore, this visualization procedure has to provide a way to allow us to easily observe staging traffic generated by a staging operation.

Furthermore, automation of these three procedures becomes essential in making this investigation efficient. It is almost impossible for a researcher to manually perform these three procedures because of the enormous amount of data captured by the tool. To capture a set of packets generated by staging communication and other types of communication, the packet-capture procedure must be performed during a job submission experiment. A job submission experiment takes a long time, for example, for at least 20 minutes in this study. Such a job submission experiment has to be conducted repeatedly to ensure its validity. Furthermore, the consumed bandwidth calculation procedure is error-prone and time-consuming because researchers themselves have to find the packet data related to their focused traffic from an enormous amount of packet-capture data. For example, the amount of packet data captured reaches 150 GB even when performing a 20-minute

packet-capture on a 1 Gbps link.

2.3.2 Functionality

The functionalities required for the packet-monitoring tool to automate the three procedures are summarized as follows.

1. Job experiment management functionality: As described above, to ensure the validity of a job submission experiment, the tool has to repeatedly conduct the job submission experiment, which starts after job requests are submitted to the job scheduler and finishes when all the job requests complete. Therefore, the packet-monitoring tool should have the functionality of automatically repeating a set of job submission experiments. To automate the repeat of the job submission experiments, this functionality needs to cooperate with the job scheduler so that the packet-monitoring tool can submit job requests and wait for all the job requests to complete.

2. Parallel packet-capture management functionality: In the packet-capture procedure, the tool needs to perform packet-captures to multiple links on core switches in parallel. The tool needs multiple computers to simultaneously perform each packet-capture on a single computer because a packet-capture performed to a link connected to core switches requires enough I/O performance to record a large amount of captured packets into files. Therefore, to automate the packet-captures, keeping the packet-captures working on multiple computers in parallel is needed. For this reason, the tool needs the functionality to manage the multiple computers so that the tool can launch and stop a process for a packet-capture on the respective computers. Also, this functionality is required to allow a packet-capture to avoid packet drops due to insufficient I/O performance when writing packet data to the HDD of each computer.

3. High-throughput bandwidth calculation functionality: In the packet-monitoring tool, the consumed bandwidth calculation procedure has to calculate consumed bandwidth from packet data. This consumed bandwidth calculation necessitates a high-throughput calculation method because the tool has to treat more than 150 GB just for the input of this calculation and thus, the consumed bandwidth calculation procedure becomes time-consuming. Researchers want the consumed bandwidth calculation to finish quickly to shorten experiment time. For this reason, the tool

needs a functionality which provides a high-throughput calculation method to automate the consumed bandwidth calculation procedure in an efficient way. Note that this calculation requires a large memory capacity because the tool has to treat a large amount of packet data. Because the size of the packet data becomes larger in the case of a long-lasting job submission experiment, it is needed to consider if the size of the packet data is larger than the memory capacity.

4. Staging event collection functionality: The visualization procedure is responsible for visualizing the consumed bandwidth associated with a staging operation. To perform this procedure, the packet-monitoring tool must know the timing when a staging operation starts and finishes, called a *staging event*, for respective staging operations. Therefore, for the purpose of automating this procedure, the tool has to automatically collect staging events.

The technical issues in developing a packet-monitoring tool with these functionalities are summarized briefly as follows:

1. The smooth interaction of the job experiment management functionality and the job scheduler.

2. The avoidance of packet drops in packet-captures managed by the parallel packet-capture management functionality.

3. The acceleration of the consumed bandwidth calculation in the high-throughput bandwidth calculation functionality.

4. The collection of staging events in the staging event collection functionality.

There are several packet-monitoring tools potentially available for the investigation in this study. Wireshark [38] is a graphical user interface (GUI) application, which has a suite of functions for capturing a set of packets and visualizing the consumed bandwidth. However, Wireshark is unsuitable for automating the packet-monitoring required in this investigation because Wireshark needs GUI-based interactive operations with itself for users. For example, Wireshark requires GUI-based click operations for users to use the functions and this fact prevents the packet-monitoring from being automated. Tshark [39] is a command-line interface (CLI) based utility of Wireshark, which has the function for capturing a set of packets and analyzing packet data. Although Tshark cannot visualize the consumed bandwidth, Tshark is useful for parsing packet data to develop the packet-monitoring tool. Tcpdump [40] is a CLI-based packet-capture tool, which

provides rich features for the packet-capture. Therefore, tcpdump is useful for developing a packet-monitoring tool. Netflow [41] is a function implemented on a network device, which analyzes a set of packets flowing on the network device. Although Netflow can calculate the consumed bandwidth for each traffic type from a set of packets flowing on the network device, Netflow cannot always be implemented on any network device. Actually, some of the network devices used in this study do not support Netflow. For a function similar to Netflow, there is sFlow [42], which is available on almost any network device. However, sFlow is unsuitable for observing how traffic changes over time in a short cycle, such as 1 second.

2.4 Design and Implementation of the Packet-Monitoring Tool

2.4.1 Design

Figure 2.3 shows the configuration of the packet-monitoring tool on an HPC cluster system. This tool is composed of two types of processes: *master process* and *monitoring process*. The master process is deployed on the management node and the monitoring processes are deployed on multiple computers, called *monitoring nodes*, on an HPC cluster system. These monitoring nodes are connected to core switches for the purpose of allowing the tool to capture packets. This placement of the master and monitoring processes allows the tool to perform packet-captures to all links on core switches. Under this configuration, the port-mirroring function is configured on core switches so that the packets passing through the ports of core switches are copied and are transferred to the monitoring nodes.

Figure 2.4 shows the interaction among the four functionalities listed in Section 2.3.2 to automate the packet-monitoring tool. First, the interaction between the master process and the job scheduler is the operation of the job experiment management functionality that allows the tool to start a job submission experiment and to wait until the completion of the job submission experiment. Second, as a parallel packet-capture management functionality, the master process has a function to allow the monitoring processes on the monitoring nodes to capture packets during a job submission experiment. Third, the high-throughput bandwidth calculation functionality is designed to achieve high-throughput computation so that monitoring processes calculate consumed bandwidth individually. Fourth, as the staging event collection functionality, the master process collects output

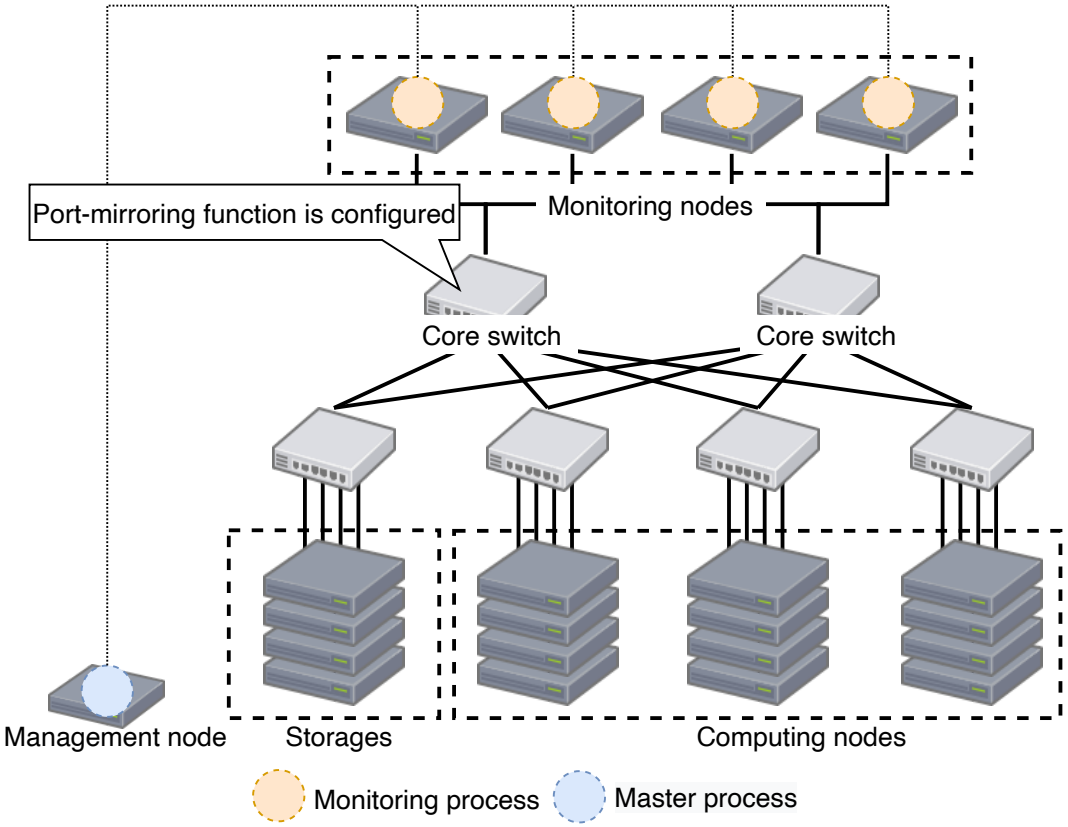


Figure 2.3: An example of deploying the packet-monitoring tool on an HPC cluster system.

logs of jobs to obtain the staging events required in the visualization procedure from the output logs. For this functionality, the staging operation is configured so that each staging operation records its staging event on the output log of a job.

Figure 2.5 shows how the packet-monitoring tool with these functionalities works. This work is described as the following five steps:

1. The master process launches the monitoring processes on the monitoring nodes and then submits job requests to the job scheduler to conduct a single job submission experiment. At this time, the monitoring processes start the packet-capture procedure in parallel.

2. The master process stops the packet-capture procedure performed by the monitoring processes when the master process finds that the job submission experiment finishes. After that, each monitoring process starts the consumed bandwidth calculation procedure for the packet data captured by itself. Also, the master process simultaneously collects output logs of jobs in the job submission experiment.

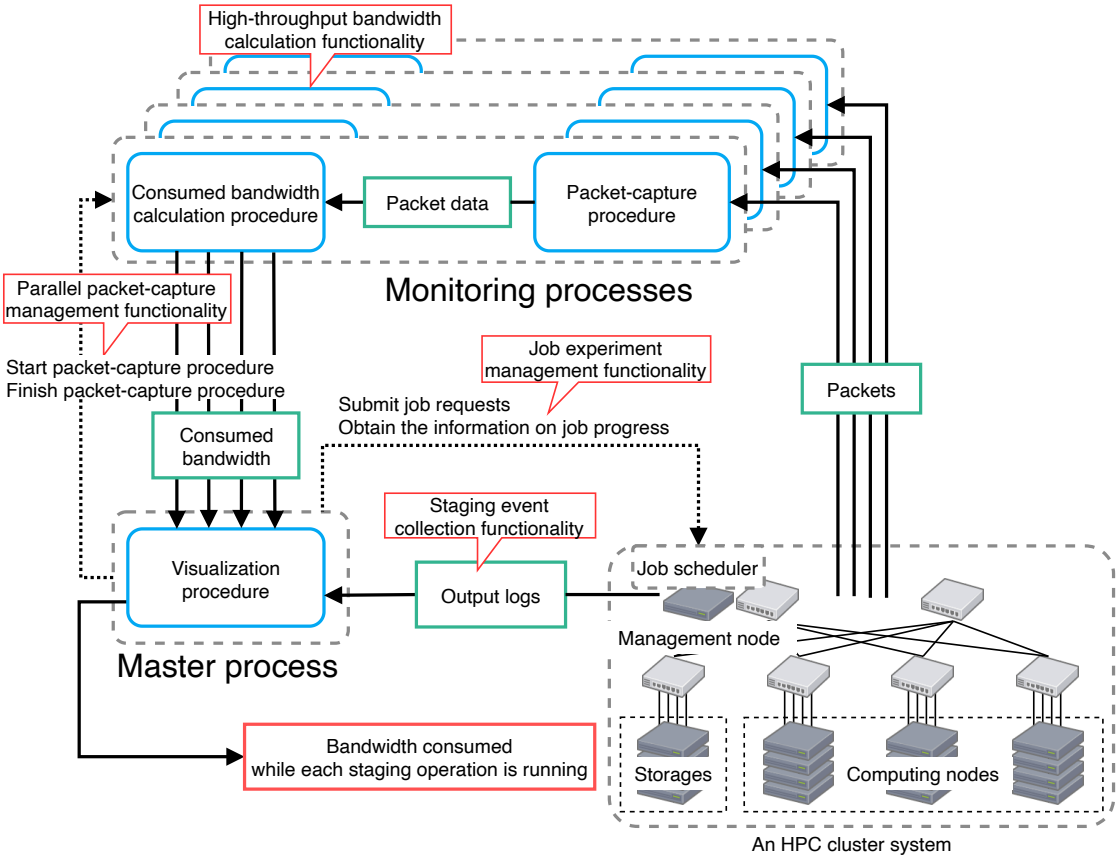


Figure 2.4: The workflow of the packet-monitoring tool.

3. When the consumed bandwidth calculation procedure finishes, the monitoring processes send the consumed bandwidth to the master process.

4. The master process returns to the first step unless all job submission experiments finish.

5. When all job submission experiments finish, the master process performs the visualization procedure.

Next, detail of the design of functionalities is explained.

Job Experiment Management Functionality

This functionality works on the master process to automatically repeat job submission experiments. To start a single job submission, the master process is designed to submit job requests. Also, to find that a single job submission experiment finishes, the master process is designed to periodically obtain the information on job progress from the job scheduler (Fig. 2.4). The reason why this design is adopted is explained from the fact that

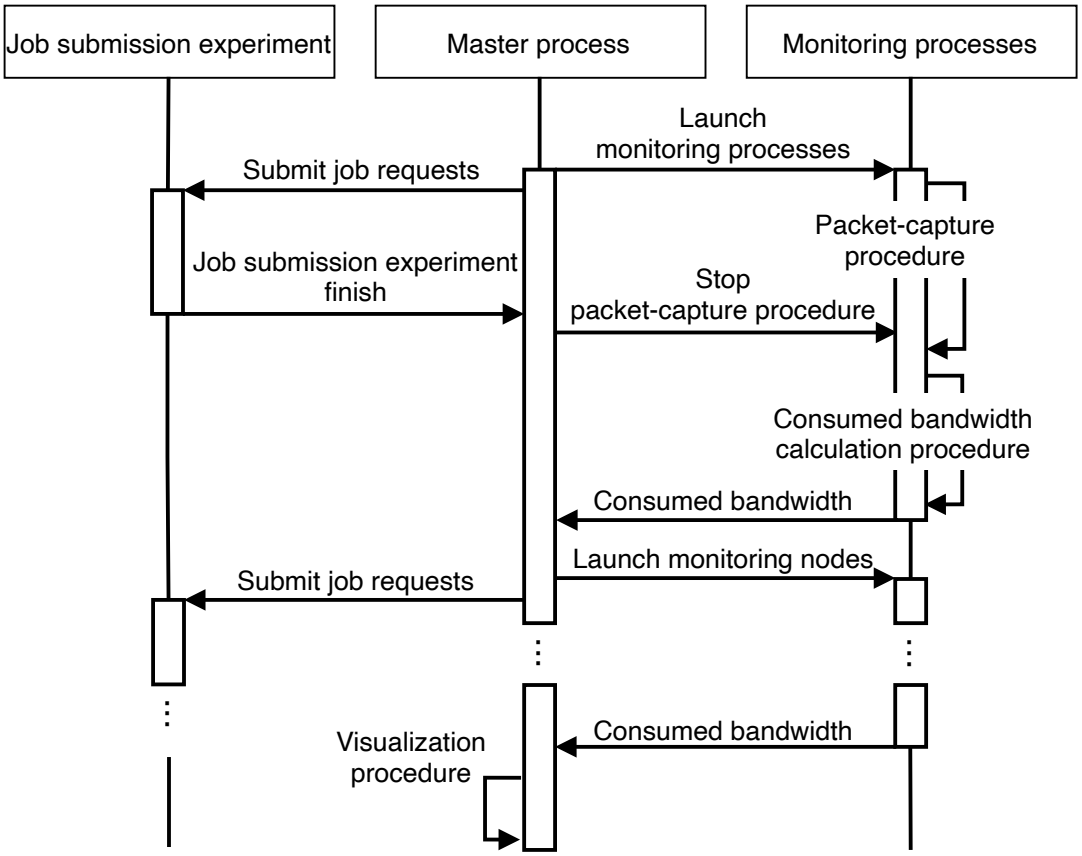


Figure 2.5: A sequential diagram of the packet-monitoring tool.

the job scheduler cannot recognize whether a set of submitted job requests is processed (a single job submission experiment finishes) or not and thus this functionality has to monitor the progress of job executions. When the master process obtains the information, the master process does nothing if this information indicates that one or more jobs are running. On the contrary, if this information indicates that no job is running, the master process stops the packet-capture procedure performed on the monitoring processes, which results in the monitoring processes proceeding to the consumed bandwidth calculation procedure. After the consumed bandwidth calculation procedure finishes, the master process submits new job requests if one or more job submission experiments remain.

Parallel Packet-Capture Management Functionality

This functionality works on the master process to manage the multiple monitoring processes. The monitoring processes have to start the packet-capture procedure before a job submission experiment starts. Also, they have to finish after the job submission experiment finishes. For this reason, the master process is designed to instruct the

monitoring processes to start and finish the packet-capture procedure, as shown in Fig. 2.4. These instructions control the monitoring processes so that they keep their packet-captures running during a job submission experiment.

High-Throughput Bandwidth Calculation Functionality

This functionality is designed to allow the consumed bandwidth calculation procedure to be performed in a high-throughput way. The consumed bandwidth calculation procedure is time-consuming due to the large size of packet data. The consumed bandwidth calculation is designed to be performed on either the master process or monitoring process. In the case of the master process, the consumed bandwidth calculation procedure can use only the management node as a computing resource. On the other hand, in the case of the monitoring processes, this procedure can use multiple monitoring nodes as computing resources. Therefore, to process the packet data in a high-throughput way, each monitoring process is designed to perform this procedure for packet data captured by itself on its monitoring node in parallel, as shown in Fig. 2.4.

Staging Event Collection Functionality

This functionality is designed to offer staging events to the visualization procedure on the master process. Although it is typical to obtain staging events from the job scheduler directly, this functionality utilized output logs of the staging operation for simplicity in the implementation of the packet-monitoring tool. For this purpose, the staging operation needs to be configured so that each staging operation writes its staging event on the output log of a job in this study. In this configuration, every time a job submission experiment finishes, the master process collects the output logs of jobs from the management node (Fig. 2.4) and then reads staging events from the output logs. After all job submission experiments finish, the master process performs the visualization procedure using these staging events.

2.4.2 Implementation

Figure 2.6 shows the implementation detail of the packet-monitoring tool. It was implemented based on the design described in Section 2.4.1 as a python program. The master process in this implementation is composed of a *management* module and a *visualization* module. The monitoring process in this implementation is composed of a *packet-capture* module and a *consumed bandwidth calculation* module. The tool

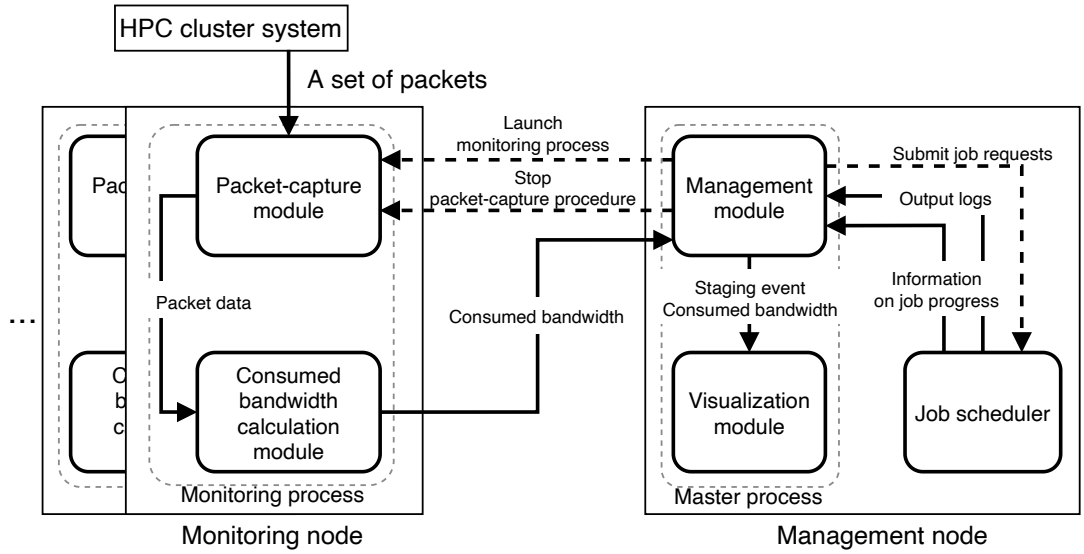


Figure 2.6: The architecture of the implemented packet-monitoring tool.

Table 2.1: Software used for implementing the packet-monitoring tool.

Software	Version
Python	3.7
Tcpdump	4.9.1
Tshark	2.6.2
Dask	2.30.0
Matplotlib	3.3.2

automatically performs the three procedures, the packet-capture, consumed bandwidth calculation and visualization procedures through the interaction of these modules. The management module provides the job experiment management, parallel packet-capture management and staging event collection functionalities. On the other hand, the consumed bandwidth calculation module provides the high-throughput bandwidth calculation functionality.

Table 2.1 summarizes software used for implementing the packet-monitoring tool.

Management Module

As described, this module is responsible for the three functionalities (the job experiment management, parallel packet-capture management and the staging event collection functionalities). For the job experiment management functionality, this module submits

job requests specified in advance on behalf of the users when the master process is launched to start a single job submission experiment. Also, this module periodically executes *squeue* [43], which is a command provided by Slurm adopted as a job scheduler in this investigation, to obtain the information on job progress. For the parallel packet-capture management functionality, this module launches monitoring processes to start the packet-capture procedure. Moreover, this module stops the packet-capture procedure performed on these monitoring processes when a job submission experiment finishes. For the staging event collection, this module obtains staging events from output logs of jobs and gives them to the visualization module as input.

Packet-Capture Module

This module captures a set of packets from the start of the monitoring processes until the management module stops this module, as shown in Fig. 2.5. To capture a set of packets, this module uses tcpdump. While tcpdump captures packets, tcpdump records them into files as packet data. After the job submission experiment finishes, this module passes the packet data to the consumed bandwidth calculation module as shown in Fig. 2.6.

To reduce the workloads of the monitoring processes, the following three setting items were configured on tcpdump. The first setting item is the file-rotation function, which divides a pcap file (packet data) into multiple pcap files by fixed size. The reason why the file-rotation function was configured is that it is a prerequisite for activating the second setting item. The second setting item is the post-rotation command function, which performs a command specified by a user, typically a *gzip* command, after the file-rotation function is performed. The author configured the post-rotation command function so that packet data outputted by tcpdump is compressed with gzip. These two setting items were configured to reduce the size of packet data stored on the monitoring nodes. The third setting item is snapshot length, which decides how many bytes tcpdump extracts data from each packet. To reduce loads of I/O operation derived from tcpdump, tcpdump was configured so that tcpdump discards all but the header of a packet when tcpdump captures the packet. Otherwise, tcpdump fails to capture packets (packet drops) because the I/O operation is not in time.

Consumed Bandwidth Calculation Module

This module calculates the bandwidth consumed by each type of traffic from the packet data given by the packet-capture module. For this calculation, this module follows the three steps below. In the following steps, the packet-monitoring tool uses Tshark for

parsing the packet data and Dask [44], which is a python-based analysis library, for the packet categorization and the consumed bandwidth calculation. After this calculation, this module sends the consumed bandwidth to the management module, as shown in Fig. 2.6.

1. Packet data parsing: The packet data outputted by tcpdump is not readable for Dask. Therefore, this module has to convert the packet data to a readable file for packet categorization and consumed bandwidth calculation with Dask. For this purpose, this module uses Tshark to convert packet data to a CSV file readable for Dask.

2. Packet categorization: This module needs to categorize the series of packets recorded in the CSV file into traffic types. This study adopted the MAC address recorded on their packet headers to categorize the packets. When the source and destination MAC addresses recorded on a packet indicates computing nodes, this module classifies the packet into inter-node communication traffic. On the other hand, when the source and destination MAC address are a pair of MAC addresses indicating a computing node and storage, this module classifies the packet into staging traffic.

3. Consumed bandwidth calculation: This module needs to calculate the consumed bandwidth for each traffic type from the categorized packet data. For this calculation, this study uses the packet length written in the header of each packet. For each classified group of packets, this module calculates the bandwidth consumed at time t on a port of the core switches by summing the packet lengths of the packets captured between time t and time $t + 1$. The consumed bandwidth calculated in this step is written to a CSV file as a result.

This module has been implemented so that memory is not exhausted because of the huge size of packet data. For this purpose, Dask has been adopted rather than pandas [45], which is a representative python-based analysis library. The reason why Dask can avoid the memory exhaustion is that Dask allows incremental access to data, while pandas tries to read it into memory all at once.

Visualization Module

This module performs the visualization procedure using the consumed bandwidth and the staging events as input given by the management module as shown in Fig. 2.6. For input data, this module outputs a set of line charts each of which plots consumed bandwidth

for each traffic type focusing on a single staging operation. In detail, this module draws the line chart of consumed bandwidth in the range from the start to the end of each staging operation. To obtain the timing when a staging operation starts and finishes, this module uses the staging events which record the timing. This module uses matplotlib [46], which is a representative python library for drawing graphs, for drawing line charts of the consumed bandwidth.

2.5 Investigation

This section verifies the speculation described in Section 2.2 by using the packet-monitoring tool developed in Section 2.4. In detail, this section quantitatively investigates how traffic collision between inter-node communication traffic and staging traffic slows the staging operation and degrades the job throughput. In the investigation, the author observed how the two types of traffic flow on the interconnect and how the traffic collision increases staging operation time by focusing on the case when the traffic collision happens on the target class of HPC cluster systems. From these observations, an HPC cluster system with the oversubscribed fat-tree interconnect was built, and then a job submission experiment was conducted on the cluster system so that the traffic collision can happen on the interconnect. In detail, the job submission experiment reproduced the situation where the two types of the traffic co-exist on the interconnect where the representative traffic load-balancing method often used in the fat-tree interconnect is applied. Also, in the job submission experiment, the author measured the bandwidth consumed by each type of traffic and staging operation time using the developed packet-monitoring tool for these observations.

2.5.1 Experimental Environment

Figure 2.7 shows the HPC cluster system built for this investigation. The HPC cluster system is composed of six computing nodes, one storage node, four switches, one management node and four monitoring nodes (*mnt1* to *mnt4*). The specification of the

Table 2.2: The spec of each node.

OS	Fedora release 27
CPU	Intel® Xeon® E5520 (2.27 GHz)

Table 2.3: The spec of each OpenFlow switch.

Switch name	Product name
ofs1, ofs2	Nexus9372PX
ofs3	UNIVERGE PF5240
ofs4	Arista 7050T-36

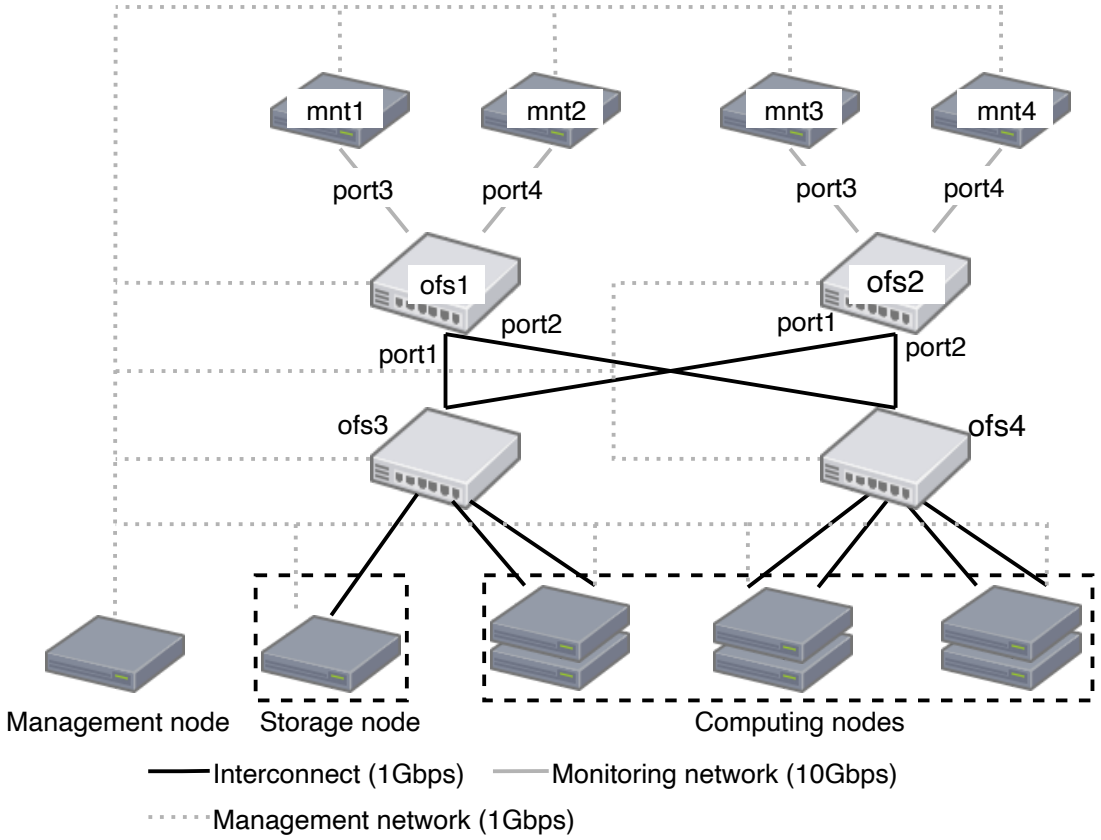


Figure 2.7: The experimental environment.

nodes used for this investigation is shown in Table 2.2. The information on switches used for this investigation is summarized in Table 2.3. The storage node stores data to be moved in staging operations as the global file system of this HPC cluster system. The management node manages the computing nodes with Slurm, which is a job scheduler widely used in modern HPC cluster systems.

Two core switches (*ofs1*, *ofs2*) and two edge switches (*ofs3*, *ofs4*) form a two-level oversubscribed fat-tree (2:3 blocking on ofs3 and ofs4) interconnect, on which the computing nodes and the storage node are connected. Equal-cost multipath (ECMP) [47], which is a traffic load-balancing method widely used for fat-tree interconnects, was

Table 2.4: Software used for this investigation.

Software	Version
Ryu	4.15
Slurm	17.02.10
GNU C Compiler	7.3.1
OpenMPI	3.0.0rc5

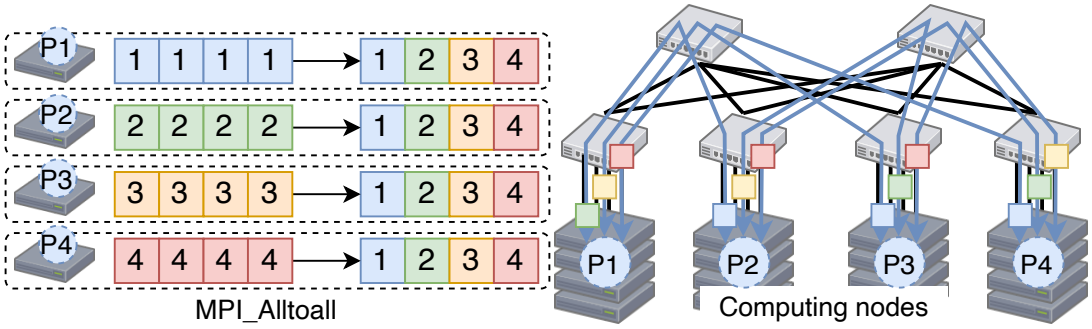
Table 2.5: The port mirroring configuration.

The target port	The destination port
port1 on ofs1	port3 on ofs1
port2 on ofs1	port4 on ofs1
port1 on ofs2	port3 on ofs2
port2 on ofs2	port4 on ofs2

adopted. For a pair of the nodes, it selects one of the candidate shortest paths searched by a routing protocol such as Open Shortest Path First (OSPF) [48]. In this investigation, ECMP-based traffic control method was implemented with the Ryu [49] framework, which is a software framework to develop an OpenFlow controller. In addition, all of the switches, the computing nodes, the storage node, and the monitoring nodes were connected on a management network.

OpenFlow [50] is the *de facto* standard implementation of Software-Defined Networking (SDN), which is a network architecture to realize network programmability and a centralized control of switches. It allows us to apply different routing algorithms and methods on the interconnect in a software programming manner. In this investigation, an OpenFlow controller was deployed on the management node to control the OpenFlow switches via the management network. Table 2.4 summarizes the software used for this investigation.

Also, the four monitoring nodes were configured so that the master process of the packet-monitoring tool can launch the monitoring processes on these monitoring nodes. On these monitoring nodes, the monitoring processes captures all outgoing packets passing through on the core switches ofs1 and ofs2. For this configuration, the port mirroring function on the core switches was used. Table 2.5 shows the port mirroring configuration.



(a) A data exchange in MPI_Alltoall. (b) A communication pattern in MPI_Alltoall.

Figure 2.8: An example of MPI_Alltoall.

2.5.2 Job Submission Experiment

In this investigation, the author submitted a set of six job requests, each of which requests MPI_Alltoall collective communication [51] to be executed repeatedly, fifty times. Each job performs the stage-in and the stage-out operation before and after computation. For the staging operation, the author set up a simple function that uses SCP to move 2 GB data generated by the *dd* command between *tmpfs* [52], which is a memory-based file system, on the storage node and tmpfs on a computing node that executes a job. Each job request is set to request three computing nodes so that two jobs are executed simultaneously on six computing nodes and then the traffic collision between the two types of the traffic is observed. Also, this configuration is applied to exclude the situation leading the performance degradation of the staging operation, where multiple processes share such resources in a computing node as ALU, cache, and NIC. For the purpose of reproducing the traffic collision on the interconnect, this study adopts MPI_Alltoall collective communication, which is one of the most important MPI communication primitives widely used by scientific applications [53]. Note that data is exchanged among all processes composing a single job with MPI_Alltoall collective communication as shown in Fig. 2.8 (a) and thus the inter-node communication traffic uses the whole of the interconnect as shown in Fig. 2.8 (b). Furthermore, to investigate how the traffic collision causes a longer staging operation, the author adjusted the job scheduler configuration so that the processes of a job were not allocated to a set of computing nodes linked to the same edge switch.

To observe how the two types of the traffic flow on the interconnect, the bandwidth consumed by the two types of the traffic on the links from the core switches to the edge switches have to be measured. For this purpose, the author launched the master process

of the packet-monitoring tool on the management node. After that, the packet-monitoring tool measured the consumed bandwidth automatically as described in Section 2.4.

To observe how the staging operation time and the job throughput is changed with and without the traffic collision on the interconnect, the staging operation times and the job throughput with and without traffic collision in the job submission experiments were measured. To measure the staging operation time without the traffic collision, the author configured staging traffic to use not the interconnect, but the management network. To measure the staging operation time both when staging traffic uses the interconnect and when it uses the management network, the author configured jobs to write output logs including staging events as described in Section 2.4.1.

2.5.3 Experimental Result

Figure 2.9 shows how each type of the traffic consumed the bandwidth on port2 on ofs1 when a stage-in operation occurred under the situation where two jobs out of six jobs are executed as shown in Fig. 2.10. In Fig. 2.9, the Y axis indicates the bandwidth consumed by each type of the traffic, and the X axis indicates time. From this graph, it is supposed that the following situation happened. From 02:24:45 to 02:25:31, only the inter-node communication traffic flowed on port2 on ofs1. The stage-in operation started at 02:25:31 and finished at 02:26:07. During this stage-in operation, the staging traffic and the inter-node communication traffic co-existed on the link between ofs1 and ofs4 and the link between ofs3 and ofs1. Also, it was observed that the bandwidth used by the inter-node communication traffic was lowered to 424 Mbps from 894 Mbps. In addition, the bandwidth used by the staging traffic was 484 Mbps. After that, only inter-node communication traffic flowed on port2 on ofs1 again. Although this result can be predicted in advance, this result is important from the point of view to verify that the traffic collision actually happens on the interconnect. Note that a similar traffic pattern was observed on the other ports (port1 on ofs1, port1 on ofs2 and port2 on ofs2) and during different time periods.

Figure 2.11 shows the result of job submission experiments. The rightmost bar shows the average execution time of staging operations performed in 300 jobs (six jobs are submitted in each of fifty job submission experiments). Also, the average job execution time and the average computation time are plotted in this graph. The result indicates that the average execution times under the situation where the traffic collision occurs were larger than the average execution time without the traffic collision. In detail, the staging operation time was increased by 45.2%, the job execution time was increased by

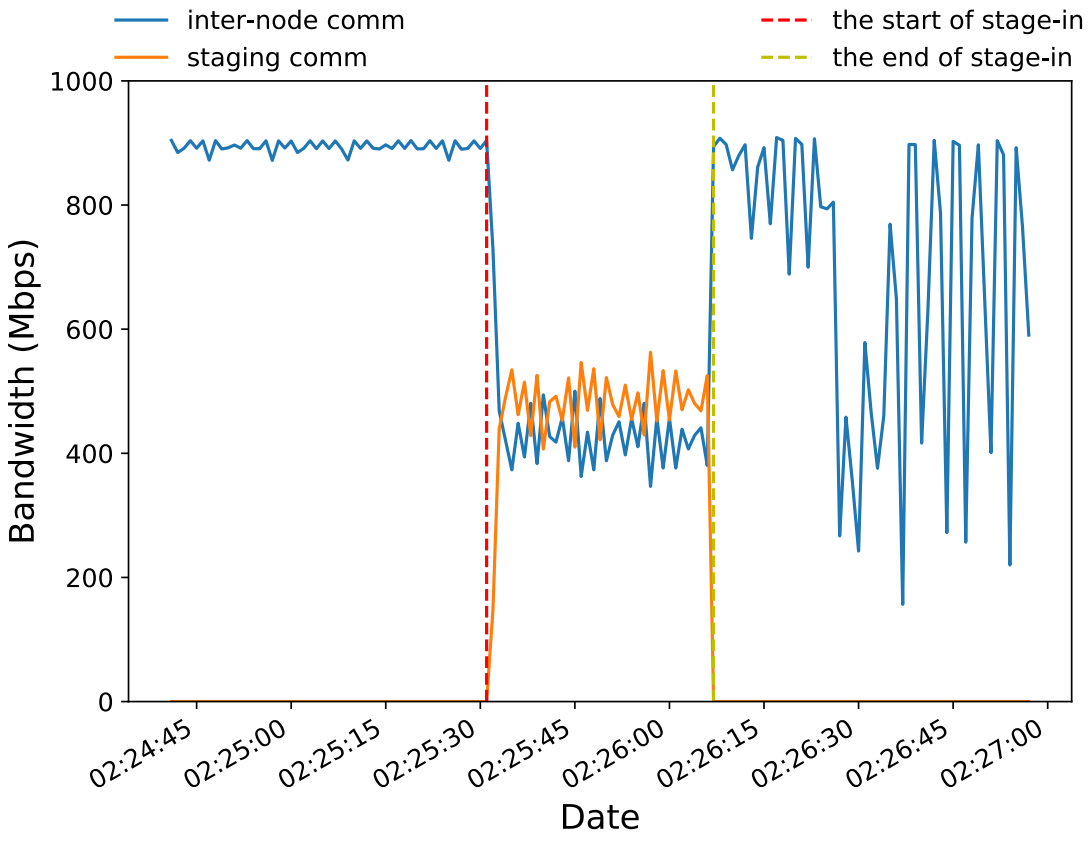


Figure 2.9: The bandwidth consumed by each traffic on port2 on ofs1.

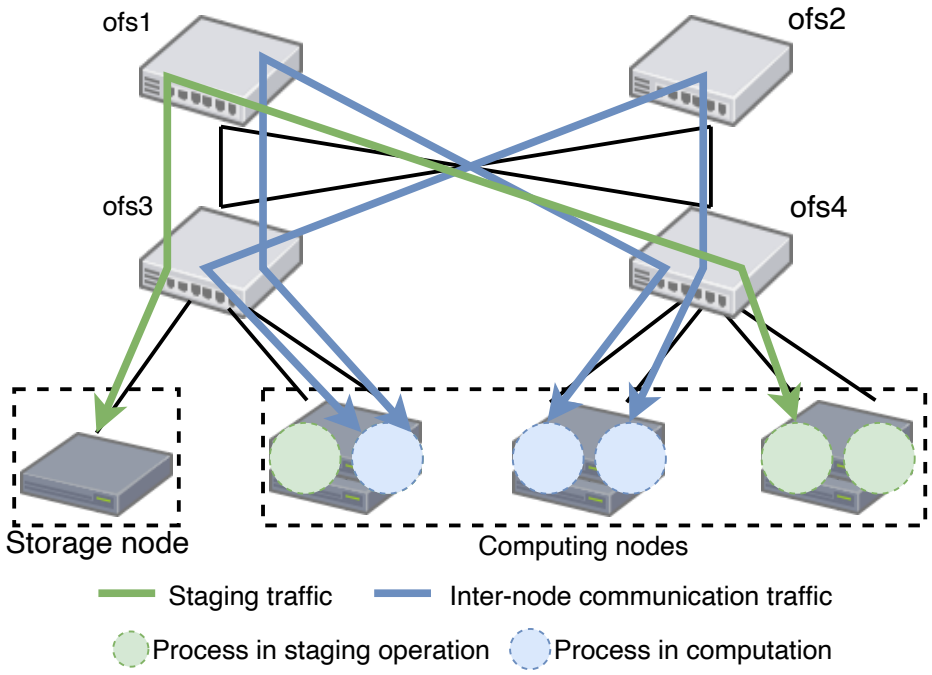


Figure 2.10: The process allocation during the stage-in operation.

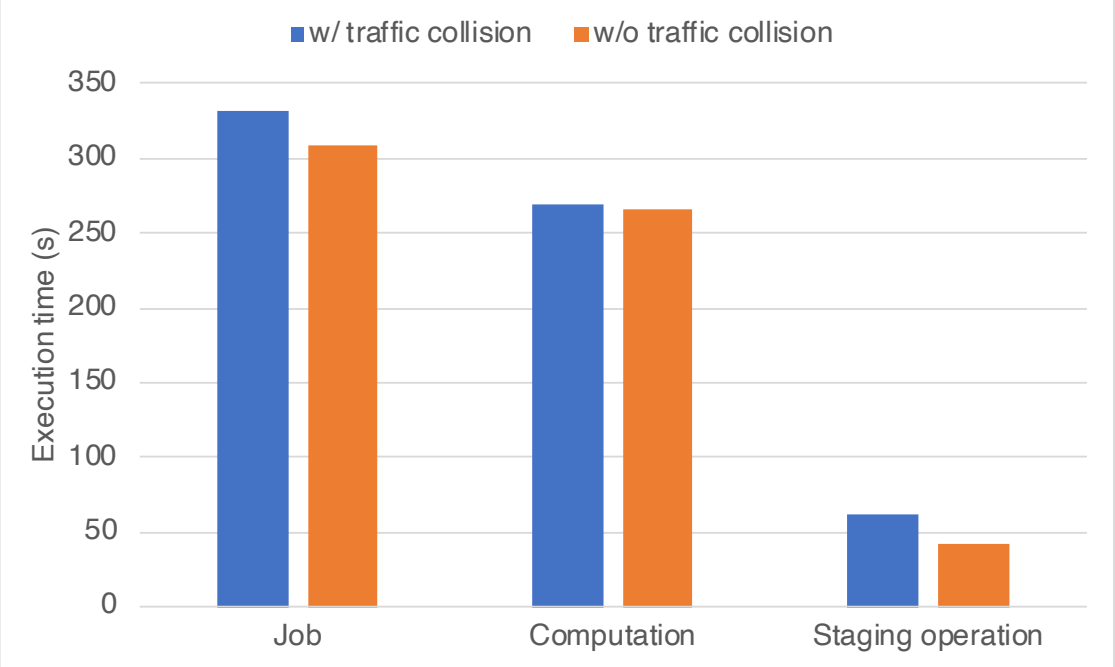


Figure 2.11: The average execution time with and without traffic collision.

Table 2.6: The average of the job throughput with and without traffic collision.

	w/ traffic collision	w/o traffic collision
Job throughput (jobs per hour)	19.50	20.83

7.7%, and the computation time was increased by 1.7%. Note that the 45.2% increase in the staging operation time means that there is room to reduce the staging operation time under the ECMP-based traffic control method by 31.1%.

Moreover, the average of the job throughput with and without traffic collision is shown in Table 2.6. The average of the job throughput was calculated from the job throughput measured in each of the fifty job submission experiments. The job throughput ($job\_throughput$) measured in each job submission experiment was calculated from the time taken for the job submission experiment ($experiment\_time$) and the number of submitted jobs ($jobs$) according to Equation (2.1).

$$job\_throughput = \frac{jobs}{experiment\_time}. \tag{2.1}$$

The right side of this equation shows the number of jobs processed in a time period and is designed based on the definition of job throughput as described in Section 1.1.2. This result indicates that the job throughput under the situation where the traffic collision

occurs was lower than the job throughput without traffic collision. Specifically, the job throughput was degraded by 6.4%.

2.6 Insight and Future Work

From the result of this investigation described in Section 2.5, it is obvious that the traffic collision between inter-node communication traffic and staging traffic causes longer staging operations and consequently, degrades the job throughput. In detail, the bandwidth consumed by staging traffic (e.g., 484 Mbps) did not reach the bandwidth provided by 1 Gbps links due to the traffic collision and therefore, the staging operation time became longer by 45.2%. Note that the possibility that I/O performance might causes such bandwidth consumption of staging traffic can be denied because tmpfs provided sufficient I/O performance for staging operations. Although this result can be predicted in advance, this result is important from the standpoint to quantitatively reveal what problem has to be resolved for the acceleration of the staging operation under the situation where there are few studies for the acceleration of the staging operation.

Based on the above insight, this dissertation discusses the traffic collision between inter-node communication traffic and staging traffic as a problem to be solved for the acceleration of the staging operation. Specifically, this dissertation aims to increase the bandwidth available to staging traffic by avoiding the traffic collision, and consequently to reduce the increase in the staging operation time by the traffic collision. For this purpose, the following two issues have to be tackled:

1. This study has to explore an approach for avoiding the traffic collision between each type of traffic and has to verify the possibility that this approach can be realized in a simulation environment.

2. Whether the above approach can actually accelerate the staging operation on a real HPC cluster system or not has to be investigated to verify the practicality and usefulness of the potential research achievement.

Future work is necessary to expand the developed packet-monitoring tool so that this tool can calculate the consumed bandwidth faster on larger-scale HPC cluster systems. Moreover, how a traffic collision between each type of traffic happens on the interconnect should be investigated through a job submission experiment using more realistic HPC applications as jobs. Although it became clear that the traffic collision causes a longer

staging operation in this study, how the traffic collision happens under the situations that various applications perform communication is not clear yet.

2.7 Conclusion

This chapter developed a packet-monitoring tool suitable for investigating how the staging operation is slowed down on the interconnect. The packet-monitoring tool is composed of a packet-capture, consumed bandwidth calculation, and visualization procedure to produce the essential information on the consumed bandwidth for each traffic type while a staging operation is running. Also, these three procedures were automated to make this investigation efficient. Using this packet-monitoring tool, this chapter investigated how the traffic collision between inter-node communication traffic and staging traffic slows the staging operation on a target class of HPC cluster systems. In this investigation, the author observed and confirmed that the traffic collision happened on the interconnect on an actual HPC cluster system built for the purpose of this investigation. Furthermore, this investigation clearly showed that the traffic collision increased the staging operation time by 45.2%. From these results, it is obvious that the traffic collision causes a longer staging operation on the target HPC cluster system, which results in the 6.4% degradation of job throughput.

This dissertation therefore targets the traffic collision between each type of traffic for a problem to be solved in achieving the acceleration of the staging operation. To resolve this problem, the author explores an approach to avoid traffic collision between each type of traffic and to verify the practicality and usefulness of the approach.

3 Acceleration of the Staging Operation Using Software-Defined Networking

3.1 Introduction

As discussed in Section 2.7, the traffic collision between inter-node communication traffic and staging traffic causes a longer staging operation and the degradation in job throughput. For this reason, this study explores an effective approach to avoid traffic collision for accelerating staging operations. To this end, the following issues have to be achieved in this study. The first issue is to find an effective approach to avoid traffic collision between each type of traffic and then investigate the possibility the approach can be realized. The second issue is to verify the practicality of the approach. In other words, from an implementation standpoint, this dissertation must investigate whether this approach can be used to accelerate the staging operation and improve the job throughput on the HPC cluster systems targeted in this study or not. This chapter tackles the first issue and the next chapter tackles the second issue.

The rest of this chapter is organized as follows: Section 3.2 explores an approach to avoid traffic collision between each type of traffic. Section 3.3 proposes a prototype method following the approach and the implementation of the prototype method. Section 3.4 verifies that the prototype method works properly and accelerates the staging operation in a simulation environment. Section 3.5 reviews the related work. Section 3.6 concludes this chapter with future work.

3.2 Approach Exploration

This section explores *traffic control* methods, with which packet flows on the interconnect are controlled, in terms of what type of traffic control method can avoid traffic collision between inter-node communication traffic and staging traffic. For this exploration, the author investigates whether *static traffic control* methods, which control packets on the

interconnect based on rules configured in advance are effective for the avoidance of the traffic collision between each type of traffic, called *traffic collision avoidance*, or not.

3.2.1 Applicability of Static Traffic Control for Traffic Collision Avoidance

Today's HPC cluster systems have adopted static traffic control methods, in which how to handle incoming packets on a switch is defined in advance by the administrator. There could be two possible strategies to avoid traffic collision between each type of traffic on the interconnect using static traffic control methods. The first possible strategy is to take advantage of traffic load-balancing functionality as provided by ECMP, which is a static traffic control method widely used in HPC cluster systems. Traffic load-balancing functionalities provided by static traffic control methods fairly distribute traffic loads onto redundant routes on the interconnect. Through an advanced use of this method, it may be possible for each type of traffic to use sufficient bandwidth. The second possible strategy is to reinforce static traffic control methods with some bandwidth assurance functionality for staging traffic. This strategy is useful for eliminating the mutual influence between each type of traffic and thus, traffic collision can be avoided.

The first strategy of benefiting from a traffic load-balancing functionality enables the efficient distribution of traffic loads on the interconnect. Importantly, however, a traffic collision between each type of traffic can take place on the oversubscribed fat-tree interconnect even if a traffic load-balancing functionality available today with static traffic control methods is applied. The reason for this is explained as follows. Inherently, the traffic load-balancing functionality distributes traffic loads without distinguishing the traffic type of packets. In short, inter-node communication traffic and staging traffic still have a chance to share the same interconnect links. Furthermore, the oversubscribed fat-tree interconnect targeted in this study cannot always provide sufficient bandwidth for each traffic, even when traffic loads are distributed perfectly. Therefore, this strategy has difficulty in contributing to the acceleration of the staging operation.

The second strategy for utilizing a bandwidth assurance functionality with static traffic control methods is also considered effective for traffic collision avoidance. In reality, however, it is difficult and almost impossible to predict how much bandwidth can be assured for each type of traffic in advance when the administrator configures bandwidth assurance on the switches composing the interconnect. Thus, in the case that the administrator overestimates the bandwidth assured for a type of traffic, there are

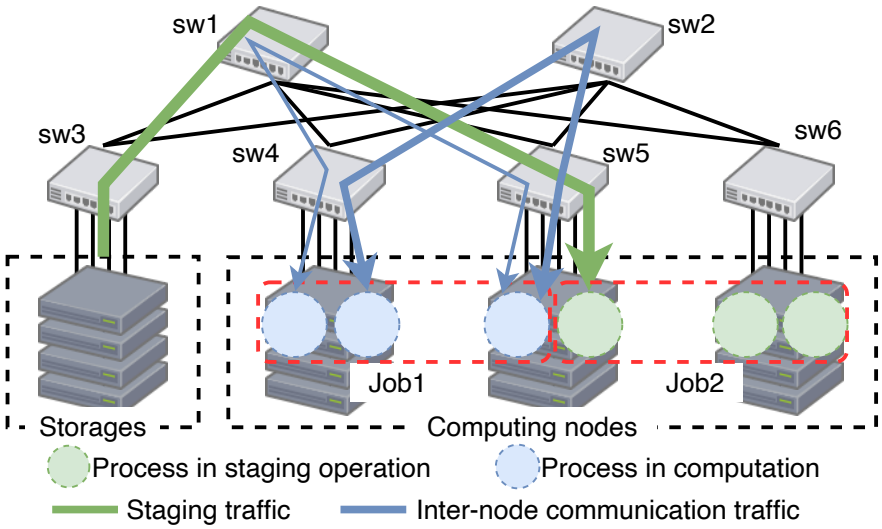


Figure 3.1: An example of the performance degradation in inter-node communication due to the bandwidth assurance for staging traffic on the interconnect.

situations which happens where the other traffic is slowed down because it cannot fully use the bandwidth on the interconnect.

Figure 3.1 illustrates in detail the problem that occurred happened in the second strategy. The interconnect of the HPC cluster system is composed of *sw1-sw6*. The administrator needs to manually configure the bandwidth assurance for staging traffic when the administrator implements the interconnect. Now assume that the administrator has configured the interconnect links between core switches and edge switches so that sufficient bandwidth is assured for staging traffic. Under this assumption, when a staging operation occurs on *job2*, staging traffic occupies the link between sw1 and sw5 due to the bandwidth assurance, and thus a traffic collision can be avoided. On the other hand, however, inter-node communication traffic passing through the link between sw1 and sw5 can use little bandwidth while staging traffic passes through the link. At this time, the bandwidth consumption of inter-node communication passing through the link between sw1 and sw5 generated by *job1* is much less than that of the inter-node communication passing through the link between sw2 and sw5 generated by job1. The computation of job1 has to wait for the slow inter-node communication passing through the link between sw1 and sw5 even if the other inter-node communication passing through the link between sw2 and sw5 is completed. Therefore, the second strategy of using bandwidth assurance methods with static traffic control methods is not practical and almost impossible to achieve for acceleration of the staging operation without performance degradation in inter-node communication.

From these considerations, the author has come up with the conclusion that the static traffic control methods widely used on the modern HPC cluster systems have difficulty in avoiding traffic collisions without the performance degradation in inter-node communication.

3.2.2 Approach

To achieve traffic collision avoidance, this study adopts a *dynamic traffic control* method, which allows us to dynamically control the packet flows on the interconnect in a programming manner, depending on the traffic situation on the interconnect. This reason is explained from the above conclusion and the fact that dynamic traffic control can change traffic controls according to an off-network event, occurring outside the interconnect, such as the commencement of a staging operation. Specifically, this study adopts a dynamic traffic control method in conjunction with the staging operation as an approach. A challenging feature of this approach is that it switches different types of traffic controls depending on whether any staging operation occurs or not. In more detail, when any staging operation occurs, this approach applies a traffic control specialized for traffic collision avoidance onto the interconnect. On the other hand, when no staging operation occurs, a traffic control based on load-balancing for the inter-node communication traffic is applied to the interconnect. To realize this versatile switching of multiple traffic controls, this approach requires the dynamic traffic control method to interact with the job scheduler to obtain staging event information about the commencement and completion of staging operations.

This approach is in line with the concept of software-defined networking (SDN), which is a concept of network architecture to realize network programmability and centralized control of switches. Therefore, to realize this approach on HPC cluster systems, this study adopts an OpenFlow network, an implementation of SDN for the interconnect. Note that several studies have discussed using SDN for the interconnects on HPC cluster systems as will be mentioned later in Section 3.5, although the OpenFlow network has not been generally adopted in HPC cluster systems.

Software-Defined Networking

SDN is a concept of networking architecture where the ordinary functionalities of traffic control and data forwarding are separated into a control plane and a data plane [54]. In a conventional network architecture, a switch composing the network usually has two

functionalities. Under this conventional network architecture, network administrators have difficulty changing the configuration pertaining to how packets flow on the interconnect because the administrators have to configure multiple switches composing the network, each of which may have its cooperating vendor-oriented interfaces. On the other hand, SDN can wipe out this administrative difficulty because it provides administrators with a way to dynamically configure switches in a software programming manner using a common set of interfaces. As a *de facto* standard implementation of SDN, OpenFlow has prevailed.

OpenFlow Network

An OpenFlow network is composed of OpenFlow switches as a data plane and an OpenFlow controller as a control plane. In the OpenFlow network, the OpenFlow controller centrally controls how the OpenFlow switches forward packets. The OpenFlow switches only forward packets, while the control and the forwarding are performed on each switch independently in a conventional network. Additionally, the OpenFlow controller is developed by an administrator with a development framework such as Ryu, Floodlight [55], Open Daylight [56] and Trema [57]. Therefore, OpenFlow provides programmable traffic control and the OpenFlow controller can control the traffic according to an off-network event.

Each OpenFlow switch has a logical structure called a *flow table* which has a set of *flow entries*. A flow entry is a rule to decide how to forward packets and it is comprised of the following items:

- Match field: The match field of a flow entry defines what kind of packets this flow entry is applied to. This flow entry is applied to the packets whose packet header has the same information (MAC address, IP address, port number, and so on) described on the match field.

- Action: The action defines how to handle a packet and it generally decides which port to forward the packet from. *Output* and *Modify* are examples of actions used in general. When *Output* is defined as action on the flow entry matched to the packet, a matched packet is transferred to a port specified by this action. The header of the matched packet is modified when *Modify* is defined as action. If the action is undefined, the matched packet is discarded.

- Priority: Priority is set for when multiple flow entries are matched to a packet. The

flow entry with the higher priority is applied to the packet even if there are other flow entries matched to the packet.

- Counter: The counter of a flow entry records how many bytes of packets this flow entry is matched to.

Based on this constitution of OpenFlow switches, each OpenFlow switch works as follows. When an OpenFlow switch receives a packet, it searches a flow entry whose match field is matched to the packet header. The OpenFlow switch forwards the matched packets according to the action set in advance on the matched flow entry.

How the OpenFlow controller controls the OpenFlow switches is explained as follows. The OpenFlow controller communicates with the OpenFlow switches through the protocol defined by the OpenFlow implementation [58]. Through this communication, the OpenFlow controller installs flow entries to OpenFlow switches and the OpenFlow switches inform network events, which occur in network, to the OpenFlow controller. As an example, a *Flow-Mod* message to create, delete, and update a flow entry is sent to the OpenFlow switch by the OpenFlow controller. As another example, when an OpenFlow switch receives a packet matched to no flow entry in its flow table, the *Packet-In* message is sent to the OpenFlow controller as a network event.

3.3 Dynamic Link-Separation Collision Avoidance

In this chapter, whether the dynamic traffic control method in conjunction with staging operation can be realized or not is investigated. For this investigation, this section proposes a *dynamic link-separation collision avoidance* method (DLSCA) as a dynamic traffic control method that interacts with the staging operation. Specifically, this section first presents the design of the proposed method. Next, the implementation of the proposed method is described in detail.

3.3.1 Design

Figure 3.2 shows an overview of DLSCA. The DLSCA is designed so that staging traffic benefits from bandwidth assurance for itself, and the bandwidth assurance barely degrades inter-node communication performance. In detail, the DLSCA switches two types of traffic controls depending on whether a staging operation occurs or not. This type of switching is called *path switching*, and leverages the network programmability brought by SDN. When no staging operation occurs, the DLSCA applies the traffic control

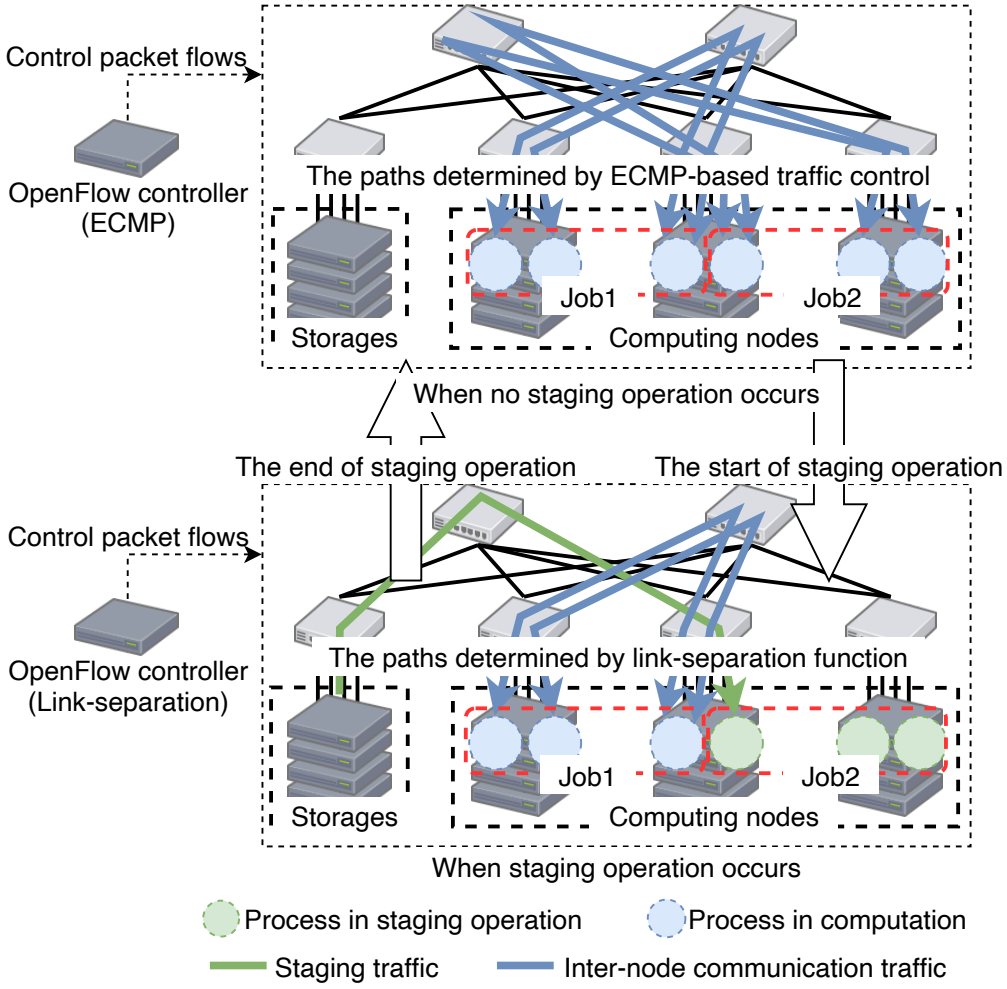


Figure 3.2: An overview of DLSCA.

using a load-balancing functionality to the interconnect. For the traffic control using the load-balancing functionality, the DLSCA uses ECMP-based traffic control as a simple traffic control in this stage of the research. On the other hand, when any staging operation occurs, the DLSCA attempts to separate the interconnect links into the links used by staging traffic and the ones used by inter-node communication traffic, or *link-separation*, for traffic collision avoidance. This link-separation function excludes any mutual impact between each type of traffic and thus each type of traffic occupies bandwidth on paths used by itself. The path switching function of the DLSCA is performed in conjunction with staging events. In detail, when a staging operation starts, the DLSCA switches the paths determined by ECMP-based traffic control to the paths determined by the link-separation function. When all staging operations finish, the DLSCA switches the paths determined by the link-separation function back to the paths determined by ECMP-based traffic control.

Path Switching Function

The path switching function is designed to interact with the job scheduler so that this function monitors whether any staging operation is running or not. It is assumed that ECMP-based traffic control works when no staging operation occurs, while on the other hand, the link-separation function works when any staging operation occurs. The path switching function starts when this function is notified by the job scheduler that the staging operation occurs. In more detail, this function switches the paths determined by ECMP-based traffic control to the paths determined by the link-separation function for each type of traffic, immediately after being notified of the commencement of the staging operation. Then it switches the paths by the link-separation function back to the paths by ECMP-based traffic control right after being notified of the end of the staging operation. After each path switching is finished, this function notifies the job scheduler that the path switching has finished, and then the job scheduler starts to perform the staging operation.

Link-Separation Function

In the link-separation function, multiple sub-trees are formed on a fat-tree interconnect and then divided into two sub-tree groups. This function allocates the two sub-tree groups to staging traffic and inter-node communication traffic, respectively, before a staging operation starts. Each type of traffic dedicatedly uses all the links in the allocated sub-tree group. Therefore, a different set of links between core switches and aggregation switches are used by either the staging communication or the inter-node communication, which results in exclusion of mutual impact between the two types of traffic. Figure 3.3 is an example of sub-trees forming on a fat-tree interconnect. This interconnect is composed of core switches (*core1* and *core2*), aggregation switches (*aggr1-aggr4*) and edge switches (*edge1-edge4*). In this example, *sub-tree A* is composed of all links connecting core1 and aggregation switches, and *sub-tree B* is composed of all links connecting core2 and aggregation switches. If sub-tree A is allocated to staging traffic and sub-tree B is allocated to inter-node communication traffic, each type of traffic can pass between core switches and aggregation switches without sharing paths between each type of traffic.

Based on this idea, the link-separation function determines paths on the interconnect for each type of traffic using the following four steps. This path determination is designed so that inter-node communication is less affected by the path switching.

1. Sub-tree formation

2. Determination of the number of sub-trees allocated to each sub-tree group

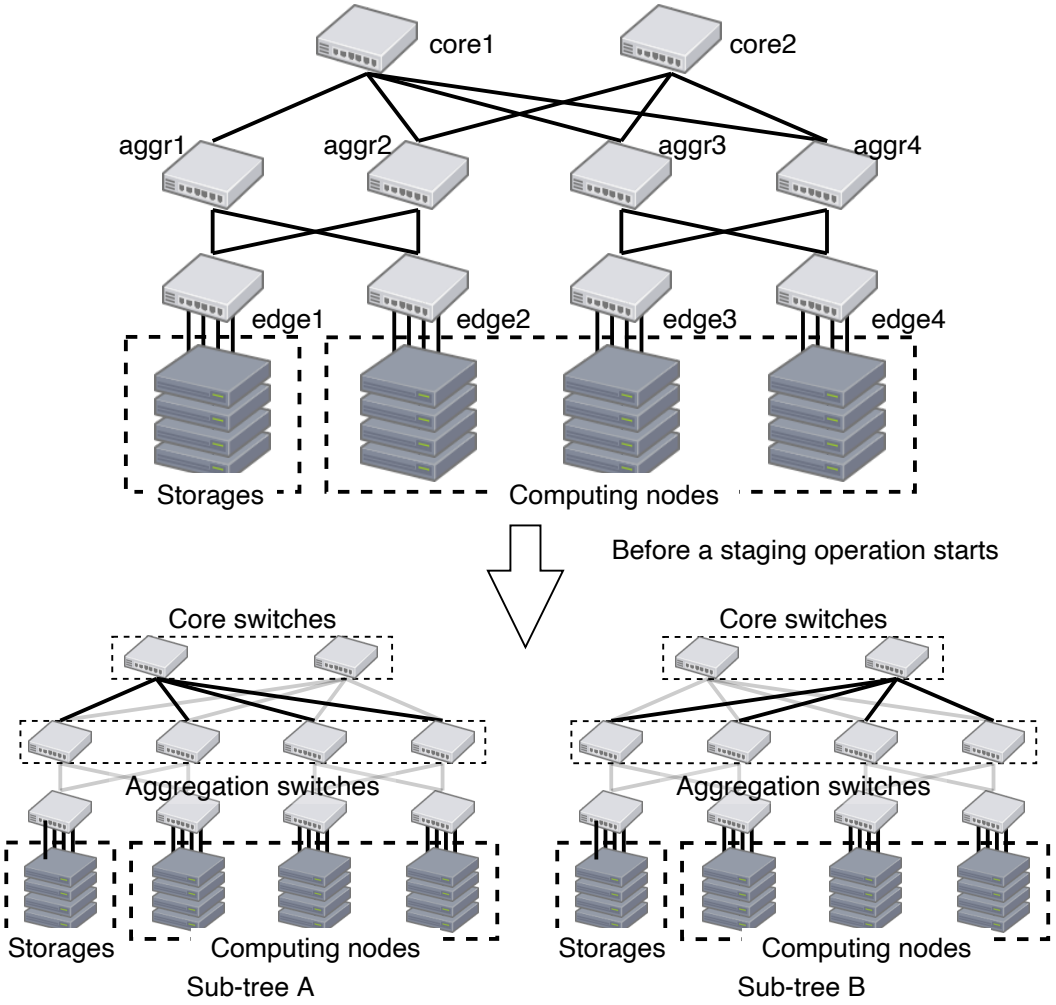


Figure 3.3: An example of sub-trees forming on a fat-tree interconnect.

3. Sub-tree allocation

4. Path search

In the first step, the link-separation function forms sub-trees between core switches and aggregation switches, so that links in a sub-tree are not shared by the other sub-trees (Fig. 3.3). As a result, the sub-tree is composed of only links connected to a core switch. It is noted that this formation is performed on each core switch.

In the second step, the link-separation function determines the number of sub-trees allocated to the sub-tree group for staging traffic ($staging\_trees$) and one allocated to the sub-tree for inter-node communication traffic ($internode\_trees$) based on the amount of inter-node communication traffic. This step is responsible for deciding how much bandwidth the link-separation function can assure each type of traffic. Table 3.1 summarizes the variables used in the following explanation. For the acceleration of

Table 3.1: The definition of the variables.

Variable	Definition
staging_group	The sub-tree group for staging traffic
staging_trees	The number of sub-trees allocated to *staging_group*
internode_group	The sub-tree group for inter-node communication traffic
internode_trees	The number of sub-trees allocated to *internode_group*
maximum_st_trees	The maximum value of *staging_trees*
cores	The number of core switches on the interconnect
bandwidth_usage	The average bandwidth consumed by inter-node communication traffic on all links connecting to core switches
link_capacity	The capacity of each link connecting core and aggregation switches
bandwidth_rate	The rate of *bandwidth_usage* to *link_capacity*
network	The network graph of the interconnect
staging_network	The partial network of *network* containing sub-trees in *staging_group*
internode_network	The partial network of *network* containing sub-trees in *internod_group*
path	The path for a pair of nodes

staging operation, staging traffic should be able to use a single sub-tree at least. In addition, in this study, there needs to be a restriction that the inter-node communication traffic should leverage more than half of the sub-trees, based on the idea that it occurs more frequently than staging communications. Based on this restriction, the maximum number of sub-trees allocated to the sub-tree group for staging traffic is determined by Equation (3.1).

$$maximum\_st\_trees = \lfloor \frac{cores}{2} \rfloor. \tag{3.1}$$

Under these conditions, this function determines *staging_trees* and *internode_trees* as indicated by Algorithm 1. In Algorithm 1, when *bandwidth_rate* is large, *staging_trees* is reduced to secure sub-trees for the inter-node communication traffic because a large *bandwidth_rate* means that inter-node communication traffic consumes a large amount of bandwidth on links connecting core and aggregation switches.

In the third step, the link-separation function divides the sub-trees formed in the first

Algorithm 1 The determination of the number of sub-trees allocated to each sub-tree group.

```
1: if maximum_st_trees == 1 then
2:     staging_trees = 1
3:     internode_trees = 1
4: else
5:     bandwidth_rate = bandwidth_usage/link_capacity
6:     for tree = 1 to maximum_st_trees do
7:         if tree/maximum_st_trees ≥ bandwidth_rate then
8:             staging_trees = tree
9:         end if
10:    end for
11:    internode_trees = cores − staging_trees
12: end if
```

step into two sub-tree groups for staging traffic and inter-node communication traffic. This function allocates $internode\_trees$ sub-trees to the sub-tree group for inter-node communication traffic and the remained sub-trees to the sub-tree group for staging traffic so that the number of rerouting inter-node communication traffic is reduced. For this purpose, this function learns how many inter-node communications pass through each core switch by counting the flow entries associated with inter-node communication on each core switch. In detail, a sub-tree whose core switch has the most flow entries associated with inter-node communication in unallocated sub-trees is allocated to the sub-tree group for inter-node communication ($internode\_group$), and this process is repeated $internode\_trees$ times. After that, the remained unallocated sub-trees are allocated to the sub-tree group for staging traffic ($staging\_group$).

As the fourth step, a set of paths to be used by the staging traffic and by the inter-node communication traffic are determined respectively. The determination algorithm is shown in Algorithm 2. In this algorithm, the link-separation function searches paths so that each type of traffic can pass through the sub-trees composing allocated sub-tree group. In detail, this function first logically divides $network$ into $staging\_network$ and $internode\_network$ as partial network graph. For this purpose, $staging\_network$ is formed by eliminating links composed of sub-trees contained in $internode\_group$ from $network$, and $internode\_network$ is formed by eliminating links composed of sub-trees contained in $staging\_group$ from $network$. Second, this function searches

Algorithm 2 The determination of paths to be used by each type of traffic.

```
▷ The partial network graph formation for each type of traffic
staging_network = network.copy()
internode_network = network.copy()
staging_network.remove(internode_group)
internode_network.remove(staging_group)
▷ The path search for each type of traffic on its partial network graph
for (src, dst) in internode_network do
    path = Dijkstra(internode_network, src, dst)
    push path to all switches
end for
for (src, dst) in staging_network do
    path = Dijkstra(staging_network, src, dst)
    push path to all switches
end for
```

path on *staging_network* for staging traffic and searches *path* on *internode_network* for inter-node communication traffic, using the Dijkstra algorithm. Finally, after searching each *path*, this function instructs the OpenFlow controller to configure *path* on the interconnect.

3.3.2 Implementation

Figure 3.4 illustrates the architecture of the implemented OpenFlow controller. The implementation was based on the design described in Section 3.3.1. This OpenFlow controller is composed of a *topology detection* module, a *traffic monitoring* module, a *staging detection* module, a *path search* module, and a *flow entry installer* module. The staging detection module offers the path switching function and the other modules realize the link-separation function. This OpenFlow controller was implemented with Ryu, an OpenFlow development framework.

The sequence diagram used in the implemented OpenFlow controller is shown in Fig. 3.5. When the OpenFlow controller is launched, the OpenFlow controller installs the flow entries for ECMP-based traffic control to all OpenFlow switches. Next, the OpenFlow controller performs the path switching every time the job scheduler notifies the OpenFlow controller of a staging event. Before a staging operation starts, the job scheduler notifies the staging detection module that the staging operation starts as a staging event. After the

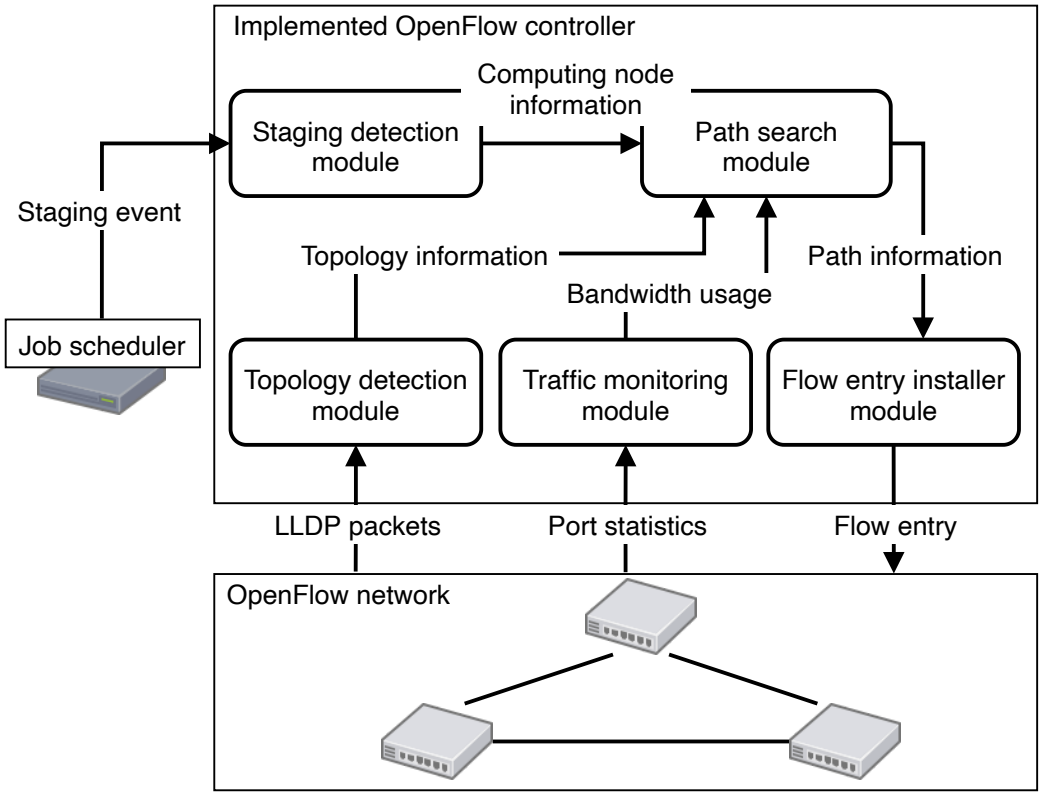


Figure 3.4: The architecture of the implemented OpenFlow controller.

path search module calculates paths on the interconnect for inter-node communication traffic and staging traffic, the flow entry installer module installs the calculated paths to all OpenFlow switches and the job scheduler starts the staging operation. When the staging operation finishes, the job scheduler notifies the flow entry installer module that the staging operation has finished and the flow entry installer module eliminates the paths from all OpenFlow switches. Next, the implementation detail of each module is explained.

Topology Detection Module

The topology detection module obtains a network graph of the interconnect to determine paths. To build the network graph, this module uses the Link Layer Discovery Protocol (LLDP) [59] provided by Ryu. This module instructs each OpenFlow switch to send LLDP packets from all its ports. The OpenFlow switches receiving an LLDP packet send *Packet-In* messages to the implemented OpenFlow controller. The implemented OpenFlow controller analyzes the received *Packet-In* messages to obtain the port information pertaining to the source and destination switches of the LLDP packets. The list of

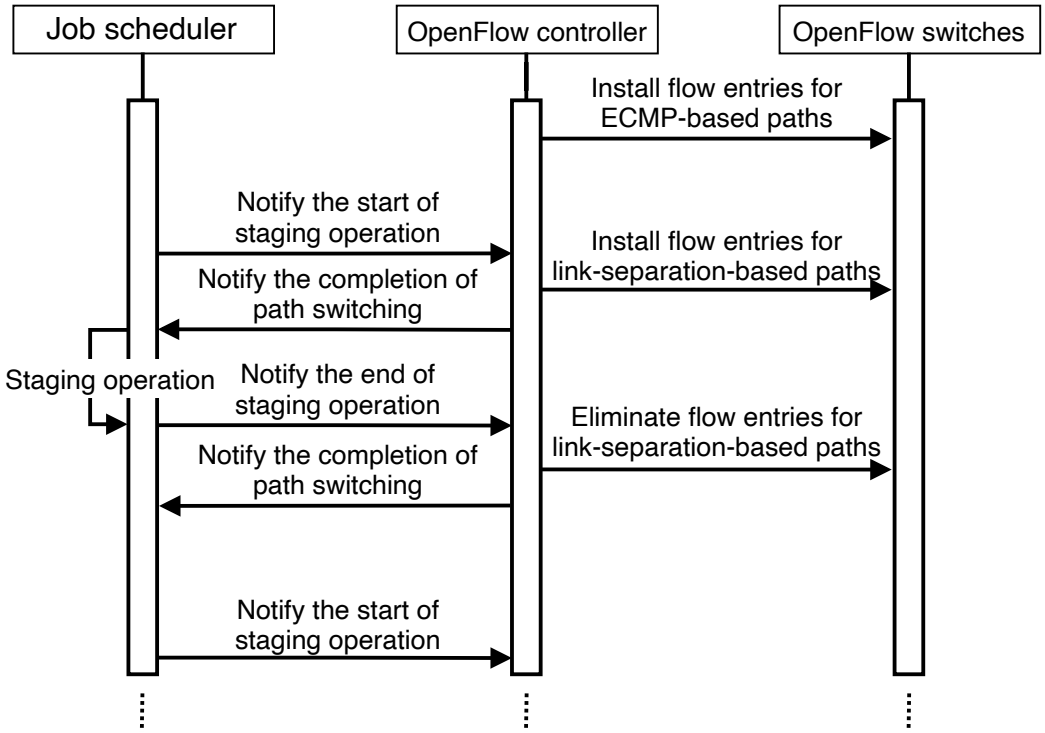


Figure 3.5: The sequence diagram of the implemented OpenFlow controller.

adjacency OpenFlow switches, or *topology information*, is generated based on the port information. This topology information is sent to the path search module, as illustrated in Fig. 3.4.

Traffic Monitoring Module

The traffic monitoring module obtains the bandwidth usage of each link periodically. To obtain the bandwidth usage, this module uses *port statistics*, which records receiving and sending bytes of each port, held on the OpenFlow switches. The implemented OpenFlow controller collects the port statistics from all OpenFlow switches periodically and then calculates each bandwidth usage of links by Equations 3.2 and 3.3. The calculated bandwidth usage is used by the path search module (Fig. 3.4).

Equation (3.2) shows the amount of transmitted data on a link, or, $Trans\_data$. In this equation, rx_bytes(t) and tx_bytes(t) are receiving and sending bytes recorded at time t, respectively. The time interval of information acquisition, Δt, was set at 5 seconds in this study. Equation (3.3) shows the bandwidth usage of a link, or $Bandwidth\_usage$. A

link bandwidth is presented to $link\_bw$ in this equation.

$$Trans\_data = \{\text{rx\_bytes}(t) + \text{tx\_bytes}(t)\} - \{\text{rx\_bytes}(t - \Delta t) + \text{tx\_bytes}(t - \Delta t)\}. \tag{3.2}$$

$$Bandwidth\_usage = Trans\_data \frac{8}{\Delta t} \frac{1}{link\_bw}. \tag{3.3}$$

Staging Detection Module

The staging detection module receives a staging event sent by the job scheduler and then instructs the path search module to perform the path switching. The staging event has the information on whether a staging operation starts or finishes and the information on computing nodes which runs the staging operation, or *computing node information* (Fig. 3.4). The path search module uses the computing node information as the identifier of the staging operation. In this study, MAC addresses are used as the identifier of the computing nodes. In detail, when this module receives a staging event which informs the commencement of a staging operation, this module instructs the path search module to calculate *link-separation-based paths*, which is determined by the link-separation function. On the other hand, when this module receives a staging event which informs the end of a staging operation, this module instructs the path search module to eliminate link-separation-based paths.

Path Search Module

The path search module is the core module of the link-separation function to determine paths. This module uses topology information, computing node information, and bandwidth usage. When this module is instructed to search link-separation-based paths, this module calculates paths using the topology information and the bandwidth usage. In this path calculation, this module uses NetworkX, which is a representative python library for exploration and analysis of network [60]. This module sends *path information*, which has the calculated paths and the computing node information, to the flow entry installer module. The flow entries for link-separation-based paths are associated with the computing node information in the flow entry installer module. When this module is instructed to eliminate link-separation-based paths, this module instructs the flow entry installer to eliminate the flow entries for the link-separation-based paths using the computing node information.

Flow Entry Installer Module

The flow entry installer module provides an HTTP server to communicate with this module and the path searcher module. The path searcher module instructs this module through the use of an HTTP request that contains the path information.

When the flow entry installer module receives path information, the flow entry installer module calculates the flow entries required for the paths specified by the path information sent from the path search module. Next, this module installs the flow entries to OpenFlow switches. This module records the flow entries in association with the computing node information contained in the path information.

On the other hand, when the flow entry installer module is instructed to eliminate flow entries for link-separation-based paths, this module eliminates flow entries associated with the computing node information given by the path searcher module from the OpenFlow switches.

3.4 Evaluation

This section evaluates the possibility that the dynamic traffic control method in conjunction with staging operation can accelerate the staging operation, by performing a simulation. For this purpose, how the proposed DLSCA behaves when inter-node communication traffic and staging traffic take place on the interconnect is investigated. For this investigation, an experimental simulation is conducted on a virtual cluster environment.

3.4.1 Evaluation Environment

The virtual cluster system shown in Fig. 3.6 was built for this evaluation. This virtual cluster system contains twelve computing nodes, a shared storage and ten OpenFlow switches. The computing nodes and storages composing a shared storage were constructed as Kernel-based Virtual Machine (KVM) machines [61] on a host machine. The specification of the host machine and KVM machines are shown in Tables 3.2 and 3.3. Each OpenFlow

Table 3.2: The spec of the host machine used for the evaluation.

OS	CentOS 6.5
CPU	Intel® Xeon® E52430 (2.50 GHz)
Memory	24 GB

Table 3.3: The spec of a KVM machine used for the evaluation.

OS	CentOS 6.8
Memory	1 GB

Table 3.4: Software used for the evaluation.

Software	Version
Ryu	4.7
OpenFlow	1.3
Torque	6.0.1
KVM	0.10.2
Open vSwitch	2.5.0
Gluster	3.8.3
GCC	4.4.7
OpenMPI	1.10.2

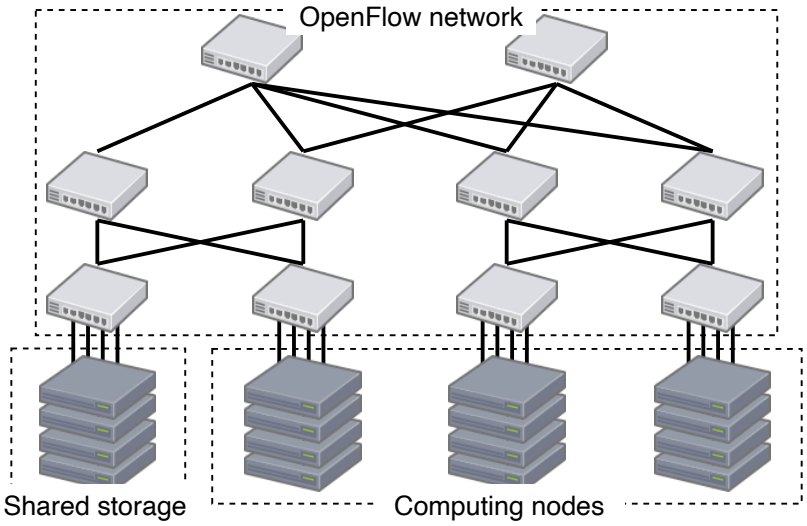


Figure 3.6: The virtual cluster system.

switch was constructed with Open vSwitch [62]. Ryu was used for the implementation of the OpenFlow controller. The shared storages were set up as Gluster, a parallel file system. Torque was used as the job scheduler in the system. Table 3.4 summarizes software used for this evaluation.

Algorithm 3 The benchmark program.

```
1: MPI.Init()
2: Start = getTime()
3: for i = 1 to REPEATS do
4:     MPI.Alltoall(send_buf, recv_buf)
5: end for
6: Time = getTime() − Start
7: MPI.Finalize()
```

3.4.2 Evaluation Method

This evaluation investigates whether inter-node communication generated by computation and staging communication before and after the computation can be controlled for the acceleration of the staging operation. For this purpose, a set of job requests was submitted to the job scheduler on the virtual cluster system. During this job submission experiment, the staging operation time from the start until the end of each staging execution, the computation time from the start until the end of each benchmark program, and the job execution time, that is, the sum of the staging operation time and the computation time in a job execution were measured per job. This measurement was performed under the deployment of the DLSCA and the ECMP-based traffic control method as an existing static traffic control method. The reason why ECMP was used in this evaluation is that ECMP is a static traffic control widely used as a method that can control multiple paths without the functions of SDN on an OpenFlow Switch. The benchmark program shown in Algorithm 3 was used as a job. In the benchmark program, the MPI program was used as an example of the representative parallel distributed processing program run on an HPC cluster system. This benchmark program performs the MPI_Alltoall collective communication, where each process sends its data given in $send\_buf$ to the other processes. This benchmark program runs this MPI_Alltoall at *REPEATS* times to produce inter-node communication traffic. In this study, *REPEATS* was set 10,000. This configuration of this benchmark program is aimed at producing amount of inter-node communication traffic so that the situation is reproduced where the traffic collision happens every time a staging operation occurs. Although the traffic collision does not always happen every time a staging operation occurs in actual HPC cluster systems, this configuration is reasonable for the purpose of this evaluation because the traffic collision reproduced in this experiment can happen in the actual HPC cluster systems.

Two kinds of job requests shown in Table 3.5 were used in the experimental simulation.

Table 3.5: Job request information.

Name	Staging	Data size	Application	Parallel
Job A	Yes	1 GB	*MPI_Alltoall*	3
Job B	No	0 GB	*MPI_Alltoall*	3

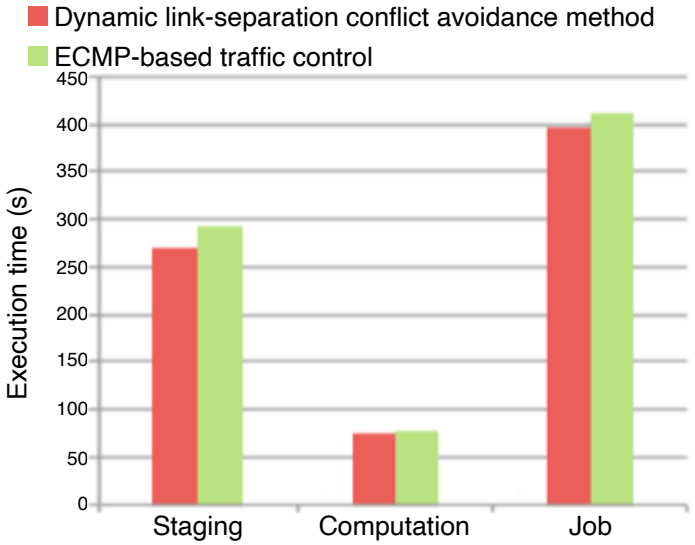
Job A is distributed to three computing nodes and the staging operation of Job A transmits data whose size is 1 GB to the nodes. Job B is distributed to three computing nodes and there is no data transmitted by the staging operation of Job B. In this experiment, both job requests were submitted to the job scheduler four times alternately to reproduce the situation where two types of traffic co-exist on the interconnect.

Furthermore, the experiment with the above configuration was conducted under the following two conditions. In these conditions, the ratio of the staging operation time in the job execution time was changed. The first condition was that the staging operation time was longer than the computation time (staging operation time > computation time). The purpose of this condition is to investigate whether the link-separation function of the DLSCA works properly for the acceleration of the staging operation. The effect of the link-separation function clearly appears when the staging operation time is longer. The second condition was that the computation time must be longer than the staging operation time (staging operation time < computation time). This condition is intended for investigating whether the path switching function of the DLSCA suppresses the performance degradation in inter-node communication.

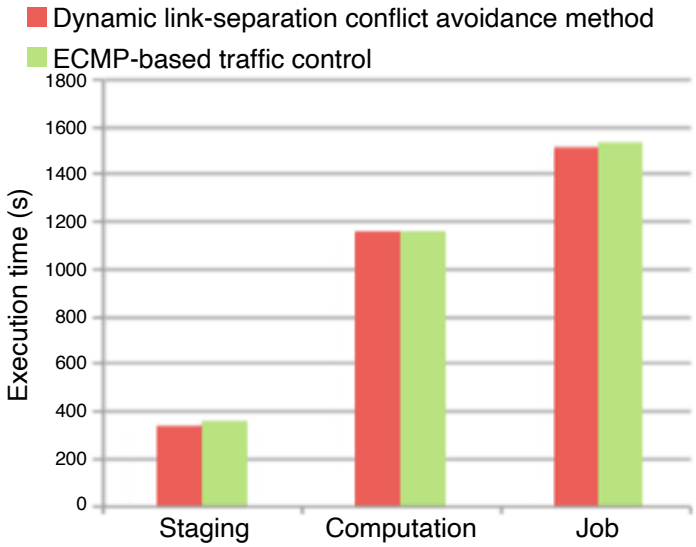
3.4.3 Results and Insights

Figure 3.7 shows the measurement results. Figures 3.7 (a) and 3.7 (b) indicate that the DLSCA succeeded in shortening the staging operation time and job execution time. Specifically, Fig. 3.7 (a) shows that the staging operation time was reduced by 7.50% and the job execution time was reduced by 3.40% in comparison with ECMP-based traffic control. Also, both figures indicate that the DLSCA kept the computation time almost the same as the ECMP-based traffic control.

Based on the above evaluation result, the author believes that the DLSCA worked properly in the job submission experiment and thus, the dynamic traffic control method in conjunction with the staging operation is feasible and promising. This reason is explained from the following two insights. First, Fig. 3.7 (a) clearly indicates that the link-separation



(a) Staging operation time > computation time.



(b) Staging operation time < computation time.

Figure 3.7: The execution time measured under two conditions.

function of the DLSCA has succeeded in accelerating the staging operation. The author speculates that the acceleration of the staging operation occurs because the staging traffic was able to use sufficient bandwidth on the paths determined by the link-separation function. Second, Fig. 3.7 (b) shows that the path switching function has succeeded in avoiding the performance degradation in inter-node communication. If the path switching function had not worked efficiently, the paths determined by the link-separation function had remained longer when no staging operation occurred. At this time, the performance in inter-node communication would have been degraded compared to the existing method,

especially when the computation time became longer (from Fig. 3.7 (a) to Fig. 3.7 (b)). However, the DLSCA kept the computation time almost the same when compared to the ECMP-based traffic control.

3.5 Related Work

Improved communication performance on the interconnect constructed with InfiniBand has been shown in several research studies [63, 64]. In vFtree [63], each traffic identified by a pair of source and destination nodes is allocated to a different virtual lane to alleviate the performance degradation caused by Head-of-Line blocking. However, this method does not solve the problem where each type of traffic blocks each other due to insufficient bandwidth on the oversubscribed fat-tree interconnect. A QoS-aware data-staging framework [64] targets the interference between inter-node communication traffic and I/O traffic, which is caused by processes to access the data on a global file system during computation. This framework allocates I/O traffic to different virtual lanes from the lanes used by inter-node communication to transfer the inter-node communication traffic with priority over I/O traffic so that I/O traffic does not affect the performance in inter-node communication. Although this framework controls multiple types of traffic to improve the performance in inter-node communication, this way to sacrifice the performance of I/O is contrary to the issue in this study of wanting to improve both the inter-node communication performance and the staging operation performance.

Traffic load-balancing methods using SDN have been presented in [65–67]. Hedera [65] estimates the bandwidth required by traffic and allocates paths to the traffic based on the estimated bandwidth so that the traffic is distributed on the interconnect without traffic collision. Dynamic load-balancing (DLB) [66] and SDN-based ECMP [67] monitor the bandwidth consumed by traffic, and allocate the path with the most available bandwidth to new traffic when the new traffic starts to flow on the interconnect. These traffic load-balancing methods fairly distribute traffic loads on the interconnect without looking into the characteristics of each type of traffic for optimal bandwidth allocation in preventing traffic collisions. However, based on the characteristic that inter-node communication traffic always flows on the interconnect while the staging traffic does not always flow on the interconnect, the DLSCA switches different types of traffic controls applied to the interconnect between the ECMP-based traffic control and the link-separation when the staging operation starts or finishes.

The idea of using a dynamic traffic control method on the interconnect of HPC cluster

systems is shared in [68–72]. In [68], an SDN-accelerated HPC cluster system has been proposed as a concept to integrate SDN into HPC cluster systems. Based on this concept, an SDN-enhanced job management system (JMS) [69, 70], which is a job scheduler to manage not only the computing nodes, but also the interconnect to improve the job throughput, and SDN-enhanced MPI [71, 72], which enhances MPI protocol using SDN to improve the performance of inter-node communication, have been proposed. In SDN-enhanced JMS [69, 70], the job scheduler allocates computing nodes to a job based on the topology of the interconnect so that as many of the computing nodes as possible are connected in one hop, and then allocates the path with the most available bandwidth to a pair of computing nodes allocated to the job for improving the inter-node communication performance. In the SDN-enhanced MPI_Bcast [71], MPI_Bcast, which is one of the MPI functions, has been enhanced using SDN so that the amount of the traffic caused by MPI_Bcast is reduced. In [72], the coordination mechanism, which applies the traffic control based on the SDN-enhanced MPI in synchronization with the execution of the MPI functions, has been proposed to apply the SDN-enhanced MPI to real-world MPI applications. Although these research studies improve the performance in inter-node communication, they do not assume that other types of traffic such as staging traffic flow on the interconnect.

3.6 Conclusion

This chapter first investigated whether the static traffic control method, which is usually adopted in traditional interconnects, can accelerate the staging operation or not. As a result, it was found that the static traffic control method made it difficult to accelerate the staging operation without the performance degradation in inter-node communication. Based on this consideration, this dissertation has adopted a dynamic traffic control method in conjunction with the staging operation, which switches traffic controls depending on whether any staging operation occurs, as an approach. This approach aimed to accelerate the staging operation with little performance degradation in inter-node communication.

This dissertation proposed and prototyped a DLSCA that separates the interconnect links to the interconnect links used by inter-node communication traffic and the interconnect links used by staging traffic only while any staging operation is running. In the DLSCA, the OpenFlow controller, which is in charge of controlling the packet flows on the interconnect, was designed and implemented to interact with the job scheduler to switch different types of traffic controls. In detail, the controller switches the paths determined

by ECMP-based traffic control to the paths determined by the link-separation function right after being notified of the commencement of a staging operation by the job scheduler. After that, the controller notifies the job scheduler of the completion to path switching. On receiving the notification, the job scheduler starts to perform the staging operation. On the other hand, the controller switches the paths determined by the link-separation function back to the paths determined by the ECMP-based traffic control right after being notified of the completion of a staging operation by the job scheduler.

Evaluation conducted on a virtual cluster system built on a single host machine showed that the DLSCA succeeded in shortening the staging operation time by 7.50% and the job execution time by 3.40% in comparison with ECMP-based traffic control, a representative static traffic control. The insight gained from this evaluation result indicates that the path switching function contributed to suppressing the degradation performance in inter-node communication and the link-separation functions succeeded in accelerating the staging operations. Also, based on the result of an experimental simulation performed on a virtual cluster system, the dynamic traffic control method in conjunction with the staging operation is considered feasible and promising.

For future work, the dynamic traffic control method in conjunction with the staging operation has to be evaluated on an actual HPC cluster system to investigate the practicality of the dynamic traffic control method for the acceleration of the staging operation. Again, the evaluation performed in this chapter used a virtual cluster system built on a single host machine. Therefore, several behaviors in the DLSCA, such as packet behavior on the interconnect, and interaction between the flow entry installer module and the OpenFlow switches, may be slightly different from the real environment. For example, the DLSCA should be reinforced to consider communication delay because it is neither ideal nor stable as expected in the simulation environment. In particular, the communication delay in the flow entry installer module's installing flow entries to the OpenFlow switches is considered to severely affect the path switching function between the two types of traffic controls. This problem is tackled in Chapter 4 next.

4 Progressive Switching-based DLSCA

4.1 Introduction

In Chapter 3, the dynamic traffic control method in conjunction with the staging operation was proposed to avoid traffic collision between inter-node communication traffic and staging traffic. Next, the possibility that this dynamic traffic control method can be realized was verified on a virtual cluster system built on a single host machine. In this chapter, as described in Section 3.1, the practicality of this dynamic traffic control is verified through the evaluation experiment using an actual HPC cluster system. In other words, whether the prototyped DLSCA can accelerate the staging operation and improve the job throughput on an actual HPC cluster system is investigated.

In advance of this investigation, the author attempted to reinforce the prototyped DLSCA so that it works robustly and efficiently to realistic system and network behaviors on an actual HPC cluster system. As described in Section 3.6, the system and network behaviors differ from those of virtual cluster systems. Examples of such differences include packet flows on the interconnect, and interaction between the flow entry installer module and the OpenFlow switches. In particular, communication latency and delay in actual environments might severely affect the behavior of the DLSCA because the communication latency and delay in real environments is not ideal, as described later in Section 4.2. For example, communication latency and delay which change over time could make the installation and elimination time of flow entries fluctuate and even become unstable. As a result, it causes the unexpected behavior of the DLSCA, such as longer switching time of traffic controls, than this study expected in the previous chapter. In this chapter, the problems caused by the instability and fluctuation of communication latency and delay are tackled.

The structure of this chapter is as follows: Section 4.2 explains a severe problem caused by the communication latency and delay and summarizes the technical issues to be achieved for resolving this problem. Section 4.3 describes the design and implementation of the DLSCA reinforced with the communication latency and delay. Section 4.4

verifies that the reinforced prototype method accelerates the staging operation in a real environment. Section 4.5 concludes this chapter with future research directions.

4.2 Problem and Technical Issues

When the DLSCA is deployed on a real environment instead of a virtual environment, the communication latency and delay give rise to a serious problem because the DLSCA is neither ideal nor stable as expected in simulation environments. In the simulation environment, the components composing an HPC cluster system, such as computing nodes and switches, are deployed on the inside of the host machine, meaning that such components share the host machine's resources including the processor and memory. On the other hand, in the real environment, the components composing an HPC cluster system are deployed with physical machines and network switches. Although the communication latency and delay between components on a virtual cluster system can be configured by the administrator to some degree, unstable and unpredictable characteristics and fluctuation of network parameters, such as jitter and packet loss, cannot be faithfully replayed in the virtual environment.

For this reason, in the real environment, communication latency and delay may affect the DLSCA more severely than we expected. In particular, in the case where the OpenFlow controller, which takes a central role of the implementation of the DLSCA, incurs long latency and jitter, the DLSCA might not be able to smoothly switch different types of traffic controls, which will result in performance degradation in the staging operation and job execution. In more detail, as explained in Section 3.3, the link-separation function of the DLSCA separates the interconnect links into the links used by staging traffic and the links used by inter-node communication traffic for traffic collision avoidance. To configure these paths, the link-separation function has to install and eliminate flow entries to and from the OpenFlow switches. At this time, the communication latency and delay between the OpenFlow controller and the OpenFlow switches might make the controller's installation and elimination of flow entries fluctuate and even unstable.

This situation is explained by the following two factors:

1. The staging operation synchronized with path switching: In the DLSCA, the staging operation is performed in synchronization with the completion of the path switching. In other words, the job scheduler waits for the completion of the path switching to perform a staging operation. Therefore, the time to install and eliminate flow entries for the path switching is added to the operation time of the stage-in and

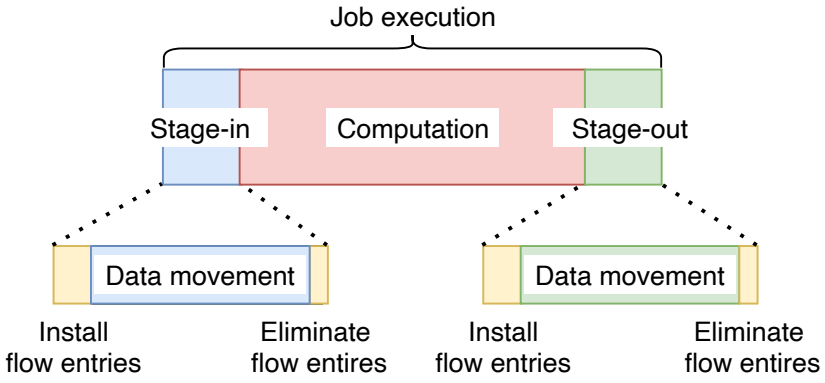


Figure 4.1: The breakdown of job execution in the DLSCA.

stage-out, as depicted in Fig. 4.1. Thus, when the path switching takes a long time due to the communication latency and delay, the staging operation time is increased. Moreover, the longer staging operation causes the degradation in job throughput as described in Section 1.1.4.

2. Large number of path configurations: The link-separation collision avoidance method configures a lot of paths on the interconnect for inter-node communication traffic and staging traffic. This is because the DLSCA configures the paths for all pairs of computing nodes and the paths for all pairs of computing nodes and storages on the interconnect when a staging operation starts. Therefore, the number of flow entries to be installed to OpenFlow switches becomes large. It takes a longer time to install a large number of flow entries to OpenFlow switches especially in a real environment due to the communication latency and delay. For example, when the paths for all pairs of sixteen computing nodes need to be configured with five flow entries individually and the install of a single flow entry takes 10 milliseconds, the total time to install all flow entries is 12 seconds ($_{16}P_2$ paths $\times$ 5 flow entries $\times$ 0.01 seconds).

To avoid the undesirable problem caused by the communication latency and delay, the following two technical issues have to be achieved. This study proposes to reinforce the DLSCA to achieve these technical issues.

1. Progressive switching: To reduce the negative influence of the longer path switching for the staging operation, the job scheduler is required to asynchronously perform staging operations without waiting for the completion of the path switching. Simultaneously, staging operations must continue functioning during the path switching because the staging operations start before the path switching starts.

2. Reduction of paths to be configured by the link-separation function on the interconnect: The link-separation function of the DLSCA separates the interconnect links into the links used by staging traffic and the links used by inter-node communication traffic for traffic collision avoidance. For the traffic control, this link-separation function configures paths for all pairs of computing nodes and all pairs of computing nodes and storages on the interconnect. This path configuration is inefficient because the set of pairs contains pairs of computing nodes where no communication happens. Therefore, for the reduction of flow entries, the link-separation function has to configure paths only for pairs of nodes that may communicate with each other on the interconnect.

4.3 Progressive Switching-based DLSCA

4.3.1 Design

A *progressive switching-based DLSCA* is proposed as a reinforced method of the DLSCA to achieve the two issues listed in Section 4.2. An overview of the reinforced method is illustrated in Fig. 4.2. The reinforced method consists of two functions: *progressive path switching* and *link-separation*. With the progressive path switching function, the reinforced method switches the paths determined by the ECMP-based traffic control (*ECMP-based paths*) and the paths determined by the link-separation function (*link-separation-based paths*) in conjunction with the job scheduler. To achieve the first issue, the reinforced method performs this path switching in a progressive way without blocking any traffic. Note that ECMP-based traffic control as a simple traffic load-balancing method was adopted for the situation when no staging operation occurs. On the other hand, with the link-separation function, the reinforced method performs a *link-separation traffic control*, which separates the interconnect links connected to core switches into the links used by staging traffic and the links used by inter-node communication traffic for traffic collision avoidance. Under the link-separation traffic control, the two types of traffic pass through the corresponding links to exclude interference with each other. Also, to achieve the second issue, the link-separation function is designed so that path configurations are reduced in comparison with the original DLSCA proposed in Chapter 3.

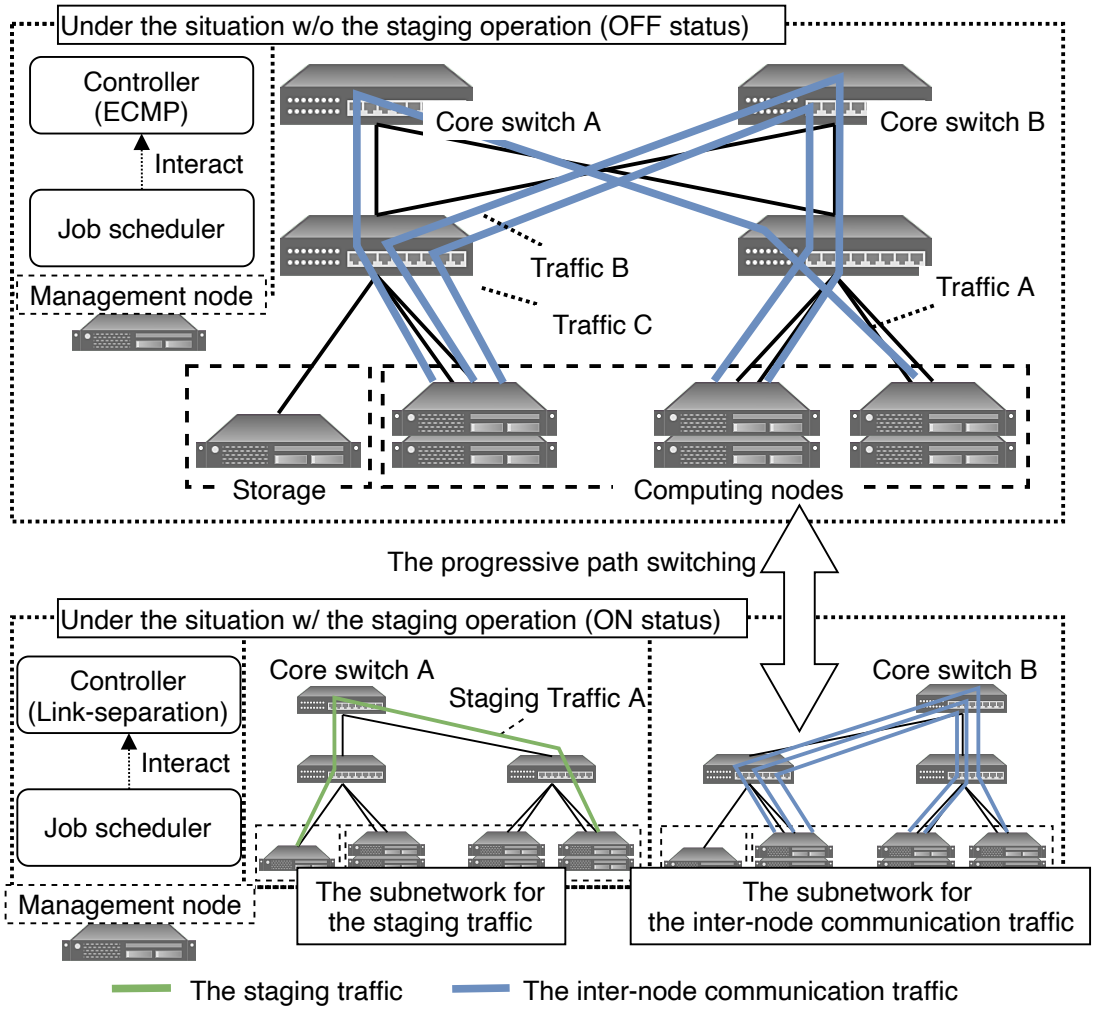


Figure 4.2: An overview of the reinforced method.

Progressive Path Switching Function

The progressive path switching function is designed to interact with the job scheduler. It is assumed that ECMP-based traffic control works when no staging operation occurs (OFF status), while, on the other hand, the link-separation traffic control works when any staging operation occurs (ON status). The progressive path switching function switches the paths for each type of traffic between ECMP-based paths and link-separation-based paths depending on whether the current status is ON or OFF. This path switching is performed after this function is notified by the job scheduler that a staging operation starts and finishes.

Figure 4.3 shows the interaction sequence diagram of the progressive path switching function with the job scheduler. As an initialization step, the reinforced method configures the ECMP-based paths on the interconnect. The progressive path switching function

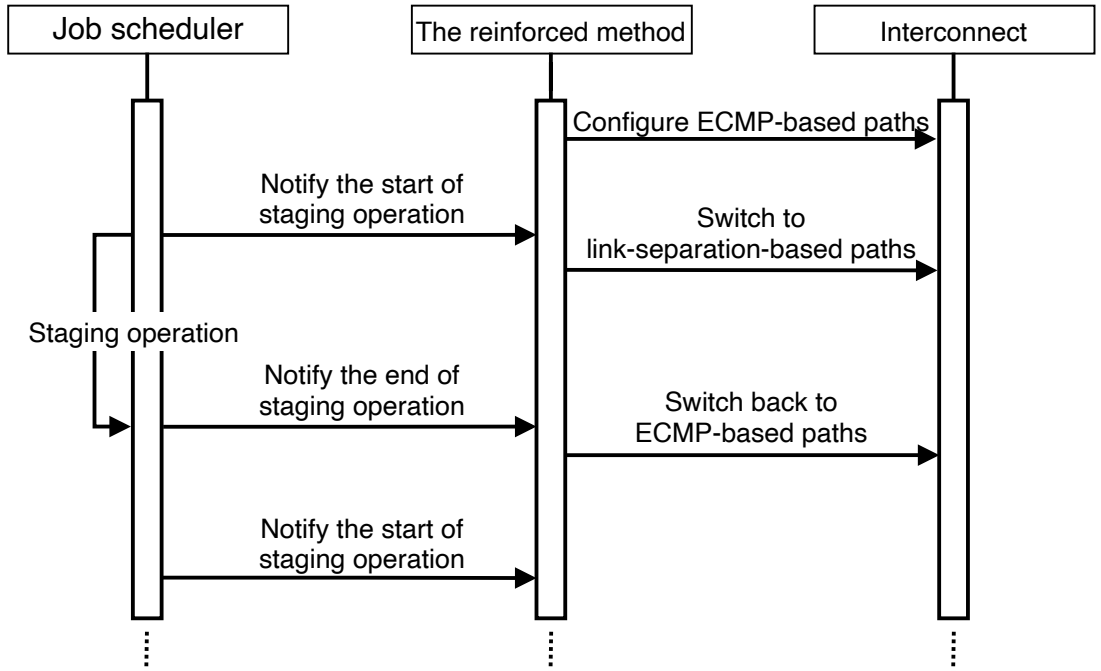


Figure 4.3: The interaction sequence diagram of progressive path switching with the job scheduler.

switches the ECMP-based paths to the link-separation-based paths, immediately after being notified of the commencement of the staging operation, or after this function recognizes the situation is under ON status. Likewise, it switches the link-separation-based paths back to the ECMP-based paths right after being notified of the completion of the staging operation, or after this function recognizes the situation is under OFF status. In this way, the reinforced method does not make the job scheduler wait for the completion of this path switching to start the staging operation, unlike in the original DLSCA.

To realize the above path switching performed asynchronously with staging operations, staging operations have to be able to continue working properly even before the completion of the path switching. In short, even while there is no link-separation-based paths configured for staging traffic during the progressive path switching function's switching motion of traffic controls, the staging operation should not be blocked. Therefore, the implementation of the reinforced method has the challenge of enabling the staging operation to continue their data movement until the path switching finishes.

Link-Separation Function

The link-separation function searches a set of link-separation-based paths for each type of traffic through the following three steps: core switch allocation, subnetwork formation

and path search. Specifically, the path search step is an important step for achieving the second issue listed in Section 4.2. The path search step attempts to reduce the number of link-separation-based paths for each type of traffic using the information on which computing nodes are allocated and to which jobs are allocated to such computing nodes, in comparison with the original DLSCA. To realize the link-separation function, it is assumed that the job scheduler sends the information on which computing nodes are allocated to which jobs to the OpenFlow controller implemented and described later in Section 4.3.2.

First, the core switch allocation step divides the core switches composing the fat-tree interconnect into the staging traffic group ($staging\_group$) and the inter-node communication traffic group ($internode\_group$) based on the amount of the inter-node communication traffic passing through each core switch. Although there is room for a better division algorithm of core switches, a simple way to equally divide core switches into the two groups has been adopted in this stage of the research. The division of the core switches is designed so that the path switching does not affect any existent inter-node communication traffic. In detail, this division is conducted based on the number of flow entries to force more paths of the inter-node communication traffic to pass through the same core switches before and after the link-separation-based paths are configured on the interconnect. The link-separation function allocates half of the core switches with more flow entries than those of the other core switches to $internode\_group$ because the more paths that pass through a core switch, the more flow entries are installed to the core switch. On the other hand, it allocates the other core switches to $staging\_group$.

Second, in the subnetwork formation step, the link-separation function forms a subnetwork for each type of traffic so that the corresponding subnetwork does not contain any core switches allocated to the other type of traffic. For this formation, this study designs the subnetwork formation algorithm shown in Algorithm 4. This algorithm logically divides the network graph of the interconnect ($network$) into the subnetwork for staging traffic ($staging\_network$) and the subnetwork for inter-node communication traffic ($internode\_network$). In detail, $staging\_network$ is formed by removing core switches composing $internode\_group$, and $internode\_network$ is formed by removing core switches composing $staging\_group$.

Third, in the path search step, the link-separation function searches the shortest paths for each type of traffic in the corresponding subnetwork so that the pairs of the nodes are reduced using the information on which computing nodes are allocated to which job. The path search step is designed based on the path search algorithm shown in

Algorithm 4 The subnetwork formation for each type of traffic.

1: $staging\_network$ = $network$.copy()
2: $internode\_network$ = $network$.copy()
3: $staging\_network$.remove($internode\_group$)
4: $internode\_network$.remove($staging\_group$)

Table 4.1: The definition of the variables.

Variable	Definition
job_ids	A set of job IDs of running jobs
$computing\_nodes$	The computing nodes allocated to the job specified by a job ID

Algorithm 5. Table 4.1 shows the definition of variables to explain this algorithm. Also, this algorithm is composed of the staging traffic path search (line 2–10), the inter-node communication traffic path search (line 11–20) and the path configuration step (line 21–24). In the staging traffic path search step, this function searches a path on $staging\_subnetwork$ for any pair of the storages and the computing nodes allocated to a job. In the inter-node communication traffic path search step, this function searches a path on $internode\_subnetwork$ for any pair of the computing nodes allocated to a job. In the path configuration step, this function configures the paths searched in the two above steps on the interconnect. In this way, the number of link-separation-based paths for each type of traffic is reduced so that the OpenFlow controller implemented later in Section 4.3.2 can take less time for the configuration of the link-separation-based paths. Note that this algorithm uses the Dijkstra algorithm as an algorithm for searching the shortest paths in the staging traffic and inter-node communication traffic path search steps.

The link-separation function works as follows for the example in Fig. 4.2. *Core switch A* and *Core switch B* are allocated to two groups individually. Core switch A is allocated to the staging traffic group and Core switch B is allocated to the inter-node communication traffic group because more traffic flows on Core switch B than Core switch A. Next, the subnetwork for the staging traffic is formed so that it contains the Core switch A, which belongs to the staging traffic group. The subnetwork for the inter-node communication traffic is formed so that it contains the Core switch B, which belongs to the inter-node communication group. On the subnetwork for the staging traffic, the link-separation function searches the shortest path for *staging traffic A*, which newly appeared between the computing node and the storage. On the other hand, on the subnetwork for the inter-node

Algorithm 5 The path search step of the link-separation function.

```
1: for job_id in job_ids do
2:     ▷ The staging traffic path search step
3:     for src in storages do
4:         for dst in computing_nodes[job_id] do
5:             path = Dijkstra(src, dst, staging_subnetwork)
6:             path_list.add(path)
7:             path = Dijkstra(dst, src, staging_subnetwork)
8:             path_list.add(path)
9:         end for
10:    end for
11:    ▷ The inter-node communication traffic path search step
12:    for src in computing_nodes[job_id] do
13:        for dst in computing_nodes[job_id] do
14:            if src == dst then
15:                continue
16:            end if
17:            path = Dijkstra(src, dst, internode_subnetwork)
18:            path_list.add(path)
19:        end for
20:    end for
21:    ▷ The path configuration step
22:    for path in path_list do
23:        configure_path(path)
24:    end for
25: end for
```

communication traffic, the function searches the shortest path for each *traffic A*, *traffic B* and *traffic C*. The reinforced method forces each type of traffic to flow on its shortest paths and consequently each type of traffic passes through a different core switch.

4.3.2 Implementation

Figure 4.4 shows the implementation detail of the reinforced method. The reinforced method was implemented using the OpenFlow framework described in Section 3.2.2. The implementation is composed of a *notificator* module and *dynamic path switching*

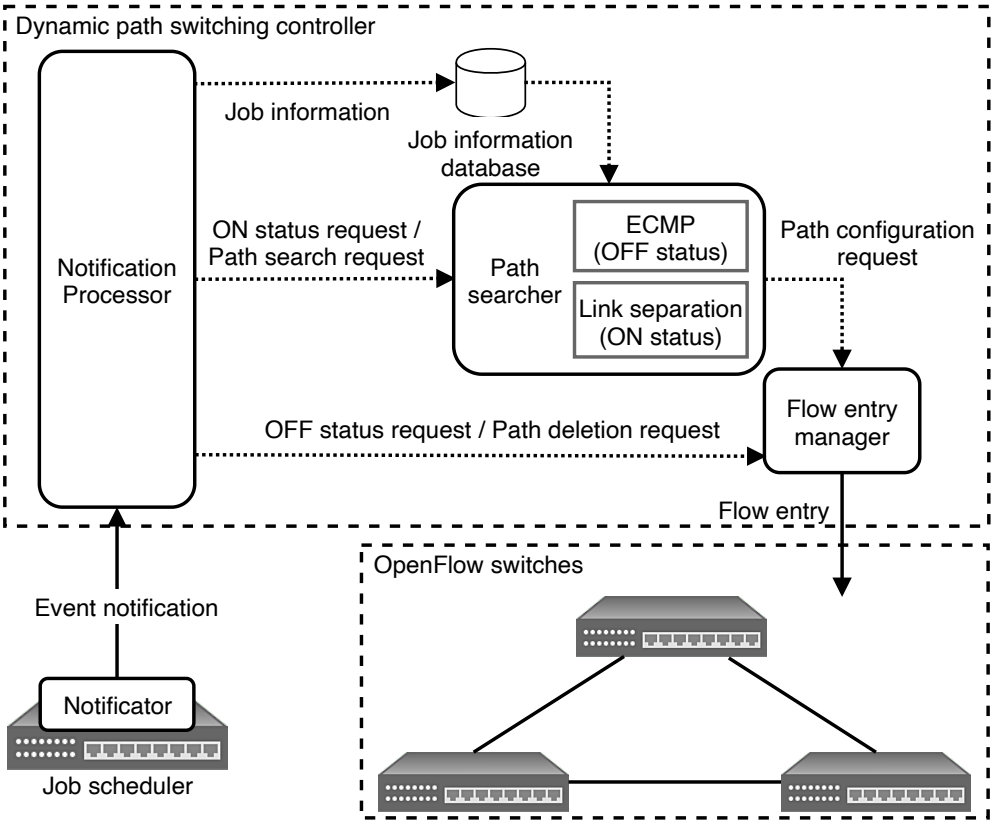


Figure 4.4: Architecture of the reinforced method.

controller comprised of a *notification processor*, *job information database*, *path searcher* and *flow entry manager* modules. The dynamic path switching controller receives event notifications regarding job and staging operation sent from the notificator module and then configures a set of link-separation-based paths for each type of traffic on the interconnect based on the event notifications. The job information database module and path searcher module provide the link-separation function. Specifically, the job information database module provides the information on which computing nodes are allocated to which job required to achieve the second issue. The notificator module, notification processor module and flow entry manager module provide the progressive path switching function. In detail, the progressive path switching function achieves the first issue by using the flow entry manager module. The job information database module is used by the notification processor module and the path searcher module.

Notificator Module

The notificator module notifies events regarding jobs and staging operations to the notification processor. For example, when a job starts, the notificator module delivers a

job event notification that contains the job ID and the flag information identifying the job being started as well as the information on the computing nodes on which the jobs are allocated. Also, when a job ends, it delivers a job event notification that contains the job ID and the flag information identifying the job being finished. On the other hand, when a staging operation starts or ends, the notificator module sends a staging event notification that contains the job ID and the flag information identifying the staging operation being started or finished.

Job Information Database Module

The job information database module holds *job information*. It is composed of the ID of a running job, the ID of the computing nodes on which the job is allocated and the flag information on whether the staging operation of the job occurs. This job information is used by the notification processor module when the notification processor module performs the switching of the traffic controls. Furthermore, the path searcher module searches a set of link-separation-based paths for each type of traffic based on the job information.

Notification Processor Module

The notification processor module controls the path switching by instructing the path searcher module and the flow entry manager module to configure and delete the link-separation-based paths on the interconnect. When the notification processor module receives an event notification, it parses the event notification and then updates the job information held by the job information database module using the parsed information. After the job information is updated, the notification processor module instructs the path searcher module to configure a set of link-separation-based paths or instructs the flow entry manager module to delete a set of link-separation-based paths. For the instruction, the notification processor module sends a request regarding the switching of the routing algorithm or the control of the path search under the link-separation traffic control according to the flowchart shown in Fig. 4.5. The control of the path search under the link-separation is mentioned in Section 4.3.2 in detail.

Four kinds of requests sent by the notification processor module to the path searcher module or the flow entry manager module exist: *ON status request*, *OFF status request*, *path search request* and *path deletion request*. ON status request and OFF status request are used for the switching of the routing algorithm, and path search request and path

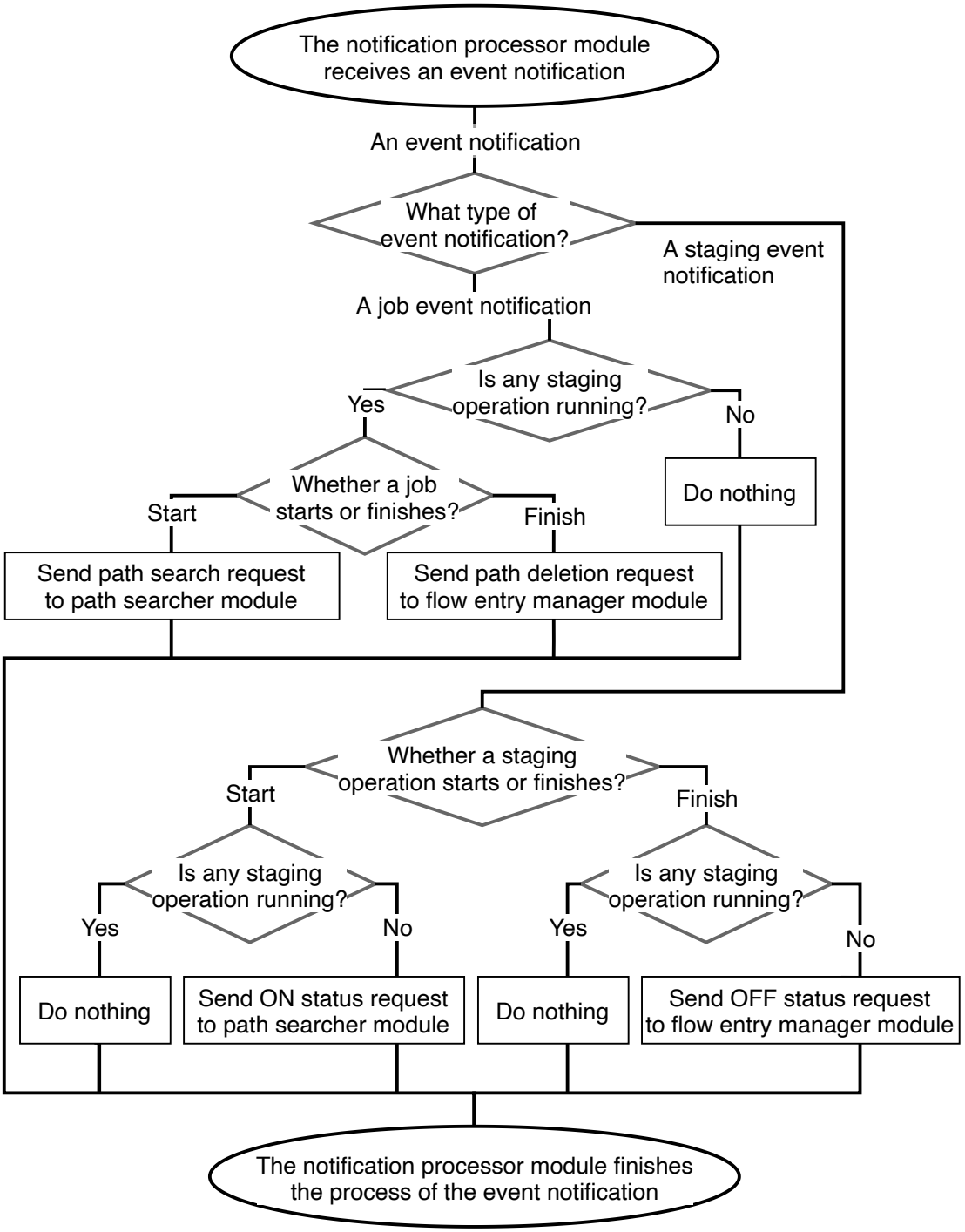


Figure 4.5: Flowchart of path switching on the notification processor module.

deletion request are used for control of the path search under the link-separation traffic control. The notification processor module sends each type of the request as follows:

- ON status request: When a staging operation occurs under OFF status, the notification processor module sends an ON status request to the path searcher

module.

- OFF status request: When all the staging operations are finished, the notification processor module sends an OFF status request to the flow entry manager module.
- Path search request: When a new job starts under ON status, the notification processor module sends the path search request to the path searcher module for configuration of paths for the traffic caused by the started job.
- Path deletion request: When a job is finished under ON status, the notification processor module sends the path deletion request to the flow entry manager module.

Path Searcher Module

The path searcher module has the following two functionalities: *default path search* and *link-separation path search*. The default path search functionality is for searching the ECMP-based paths under OFF status. On the other hand, the link-separation path search functionality is for searching the set of link-separation-based paths under ON status. The *path configuration request*, which instructs the flow entry manager module to configure the switches composing the interconnect so that searched paths are configured on the interconnect, is sent to the flow entry manager module. For this instruction, the path configuration request holds the *searched path information*, which is expressed as the array of node and switch IDs such as [$comp\_node1$, $edge\_switch1$, $core\_switch1$, $edge\_switch2$, $comp\_node2$],

In the case of ECMP (OFF status), the default path search functionality is performed only when the dynamic path switching controller is initialized and launched. The information on the searched path is sent to the flow entry manager module. On the other hand, the link-separation path search functionality is performed whenever the path searcher module receives any ON status request or path search request from the notification processor module. After that, the information on the searched path is sent to the flow entry manager module.

The link-separation path search functionality, which is an implementation of the link-separation function, is as follows. This functionality is composed of three steps as mentioned in Section 4.3.1: the core switch allocation step, the subnetwork formation step and the path search step. The path searcher module performs these three steps in the order as designed when receiving an ON status request. On the other hand, it performs only the path search step when receiving a path search request. The ON status request case

is for searching a set of link-separation-based paths for the traffic caused by the running jobs and the path search request case is for searching a set of link-separation-based paths for the traffic caused by a started job. Moreover, the information on which computing nodes are allocated to which job is obtained from the job information recorded by the notification processor module, and then this information is used in the path search step.

Again, for the example in Fig. 4.6, the path search based on this path searcher module works as follows. First, the path between the computing nodes *c1* and *c3* allocated to the job *J1* is configured according to ECMP-based traffic control under OFF status. When the job *J2* starts a stage-in operation after J2 starts on the computing nodes *c5* and *c6*, the notification processor module sends an ON status request to the path searcher module. On receiving the ON status request, the path searcher module searches a set of link-separation-based paths for the traffic caused by J1 and J2, and then it sends the path configuration requests to configure the link-separation-based path for the inter-node communication traffic caused by J1 and the requests for the staging traffic caused by J2 on the interconnect. The path configuration request to configure the former link-separation-based path contain the searched path information [$c1$, $sw3$, $sw1$, $sw4$, $c3$] and the requests to configure the latter link-separation-based path contains the searched path information [$storage$, $sw3$, $sw2$, $sw4$, $c5$]. When job *J3* starts its computation under this situation after J3 starts on the computing nodes *c2* and *c4*, the notification processor module sends a path search request to the path searcher module. On receiving the path search request, the path searcher module searches a set of link-separation-based paths for the traffic caused by the started job J3, and then it sends the path configuration request to configure the link-separation-based paths for the inter-node communication traffic caused by J3 using the same core switch sw1 used by the request for the inter-node communication traffic caused by J1. The path configuration request to configure this link-separation-based path contains the searched path information [$c2$, $sw3$, $sw1$, $sw4$, $c4$].

Flow Entry Manager Module

The flow entry manager module configures paths when receiving the path configuration request sent from the path searcher module and removes paths when receiving an OFF status request or a path deletion request sent from the notification processor module. When the flow entry manager module receives a path configuration request, it makes flow entries to configure the requested path and then installs them to the switches composing the interconnect. When the flow entry manager module receives an OFF status request, it removes all of flow entries for link-separation-based paths from the switches. When the

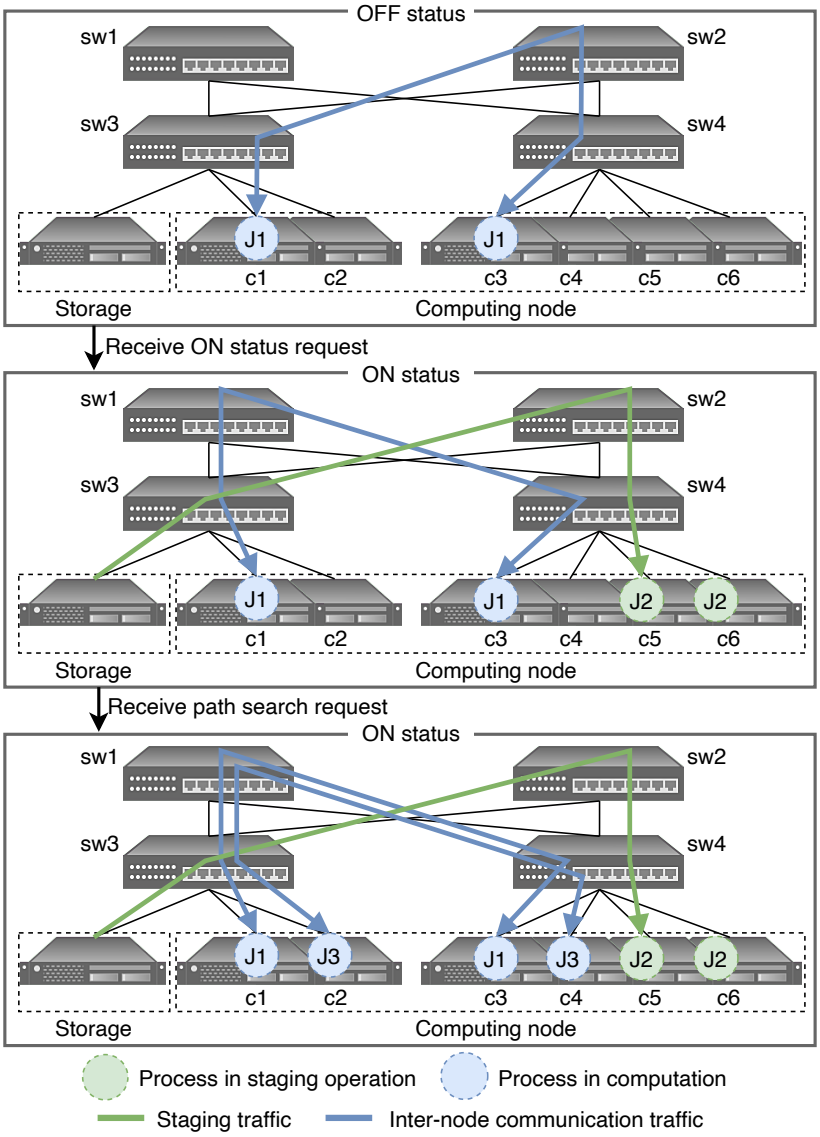


Figure 4.6: A path search example based on the path searcher module.

flow entry manager module receives a path deletion request, it removes the flow entries for the link-separation-based paths for the traffic caused by a finished job.

To perform the switching of ON and OFF status seamlessly without blocking both the two types of traffic, the flow entry manager module manages the flow entries configuring the ECMP-based paths and link-separation-based paths so that the ECMP-based paths can be available to any traffic until the flow entries configuring the link-separation-based paths are installed on each switch. Figure 4.7 shows the switching mechanism based on this idea. In this switching mechanism, the flow entries for path configuration under OFF status are always on switches, while the flow entries for path configuration under ON

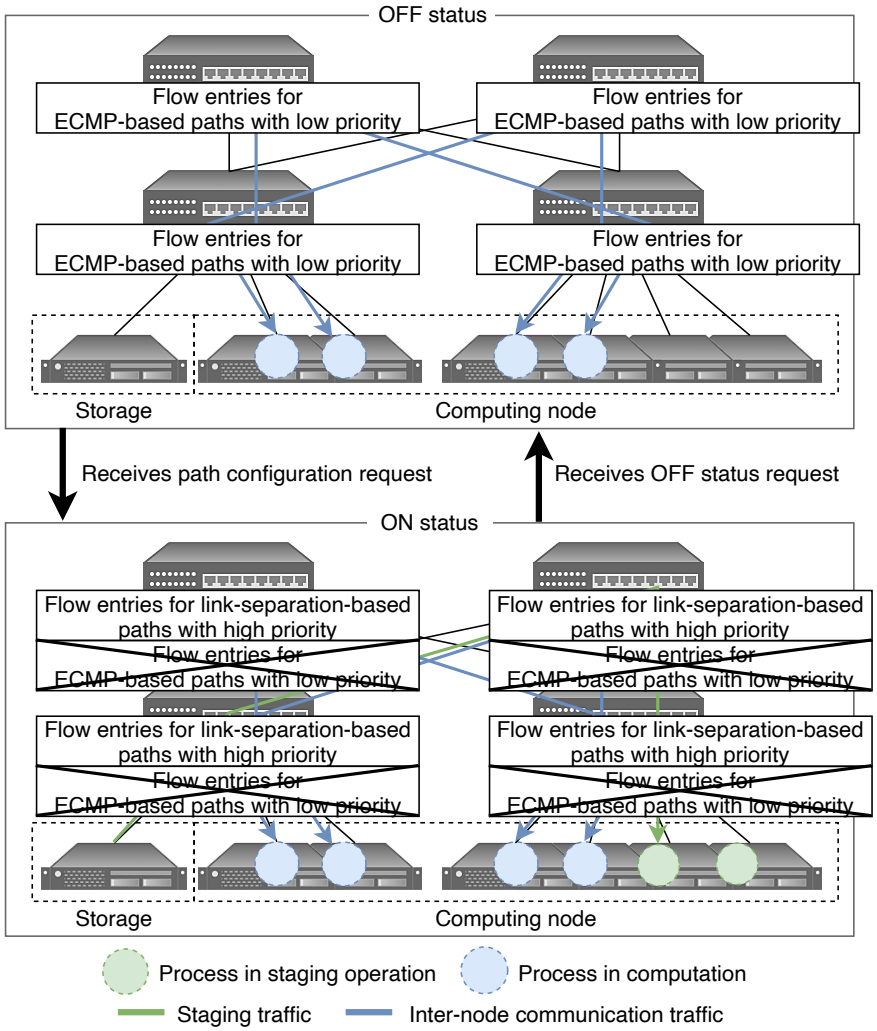


Figure 4.7: The switching mechanism of ON and OFF status.

status are always regenerated and installed with higher priority than the ones under OFF status.

4.4 Evaluation

4.4.1 Evaluation Environment

Figure 4.8 shows the HPC cluster system built for this evaluation. The HPC cluster system is composed of six computing nodes, one storage node, four switches, one management node and four monitoring nodes (*mnt1* to *mnt4*). The specification of the nodes used for this evaluation is shown in Table 4.2. The storage node stores data to be moved in staging operations as the global file system of this HPC cluster system. The management

Table 4.2: The spec of each node.

OS	Fedora release 27
CPU	Intel® Xeon® E5520 (2.27 GHz)

Table 4.3: The spec of each OpenFlow switch.

Switch name	Product name
ofs1, ofs2	Nexus9372PX
ofs3	UNIVERGE PF5240
ofs4	Arista 7050T-36

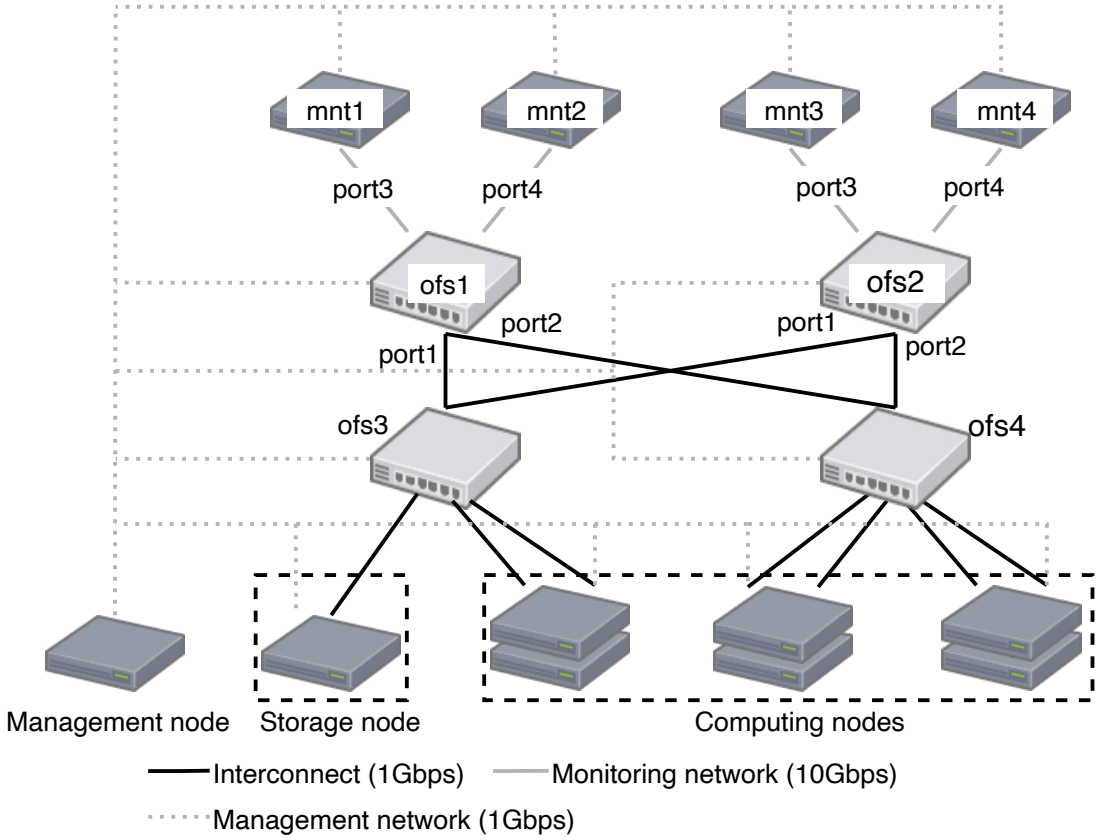


Figure 4.8: The experimental environment.

node manages the computing nodes with Slurm, which is a job scheduler widely used in modern HPC cluster systems.

An OpenFlow network composed of OpenFlow switches connects the computing nodes and the storage node as the interconnect. The information on OpenFlow switches used

Table 4.4: The port mirroring configuration.

The target port	The destination port
port1 on ofs1	port3 on ofs1
port2 on ofs1	port4 on ofs1
port1 on ofs2	port3 on ofs2
port2 on ofs2	port4 on ofs2

Table 4.5: Software used for this evaluation.

Software	Version
Ryu	4.15
Slurm	17.02.10
GNU C Compiler	7.3.1
OpenMPI	3.0.0rc5

for this evaluation is summarized in Table 4.3. On the OpenFlow network, two core switches (*ofs1*, *ofs2*) and two edge switches (*ofs3*, *ofs4*) form a two-level oversubscribed fat-tree (2:3 blocking on ofs3 and ofs4). The implemented dynamic path switching controller was deployed on the management node to control OpenFlow switches via the management network and then the reinforced method was adopted for the traffic control on the interconnect. In addition, all of switches, the computing nodes, the storage node and the monitoring nodes were connected on a management network.

The management node and the four monitoring nodes were configured so that the packet-monitoring tool developed in Section 2.4 can be used. On these monitoring nodes, the monitoring processes of the packet-monitoring tool captures all outgoing packets passing through on the core switches ofs1 and ofs2. Also, on the management node, the master process of the packet-monitoring tool visualizes the bandwidth consumed by each type of traffic. For this configuration, a port mirroring function on the core switches was used. Table 4.4 shows the port mirroring configuration. Table 4.5 summarizes software used for this evaluation.

Importantly, the evaluation environment described above is the same environment used in Chapter 2 except that the dynamic path switching controller that switches the ECMP-based paths and the link-separation-based paths was deployed. This configuration environment was rebuilt for comparison purposes with the evaluation result of Chapter 2.

4.4.2 Evaluation Method

In this evaluation, a set of six job requests, each of which requests MPI_Alltoall collective communication to be executed repeatedly, was submitted fifty times as job submission experiments. Each job performs the stage-in and the stage-out operation before and after computation. In the staging operation, 2 GB data is moved between tmpfs on the storage node and tmpfs on a computing node that executes a job. Each job request is set to request three computing nodes so that two jobs are executed simultaneously on six computing nodes to reproduce the situation where the traffic collision between each type of traffic can happen when the reinforced method is not applied to the interconnect. Furthermore, the author adjusted the job scheduler configuration so that the processes of a job were not allocated to a set of computing nodes linked to the same edge switch for the same reason as above. This job submission experiment is designed to be the same as in Section 2.5.

To investigate whether the reinforced method can improve performance in the staging operation and job throughput or not, the job execution time composed of the staging operation time and computation time was measured per job and the job throughput was measured per job submission experiment. Moreover, how the bandwidth consumed by each type of traffic is changed over time was observed using the packet-monitoring tool. Furthermore, how long it takes to perform the progressive path switching between ON and OFF status was measured in the job submission experiment.

4.4.3 Evaluation Result

This evaluation shows three types of evaluation results: performance improvement verification, verification of traffic collision avoidance, and operational verification of progressive path switching. In the performance improvement verification, whether the reinforced method can accelerate the staging operation with little performance degradation in inter-node communication and can improve the job throughput in the job submission experiment or not is investigated. In the verification of traffic collision avoidance, whether the reinforced method can avoid the traffic collision between each type of traffic or not is investigated. In the operational verification of progressive path switching, how the progressive path switching function of the reinforced method contributes to improvement in the job throughput or not is investigated.

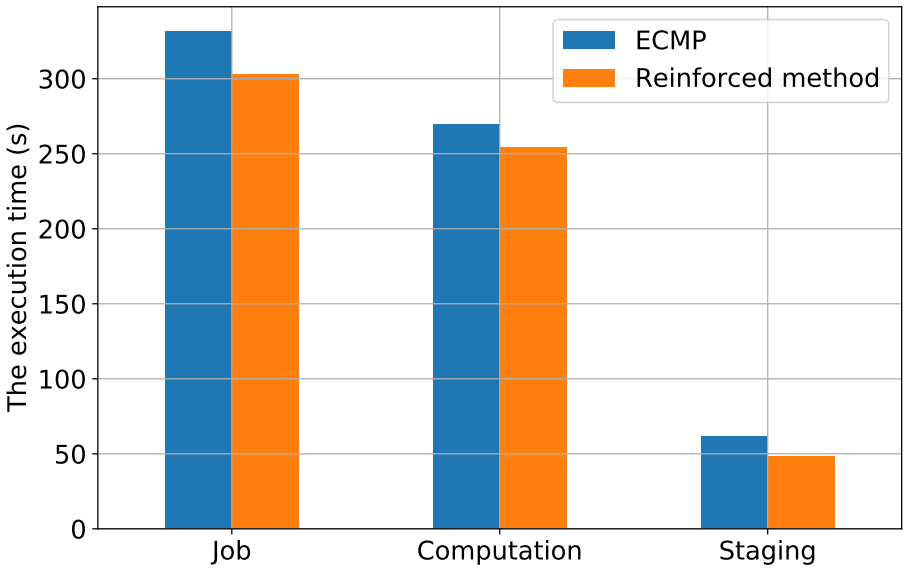


Figure 4.9: The average execution time of the jobs in the job sets.

Performance Improvement Verification

Figure 4.9 shows the result of the job submission experiment. This result is the average execution time of the jobs in the fifty job sets. In this result, three component times are plotted: the job execution time, the computation time and the staging operation time. To investigate how the progressive switching-based DLSCA changes these times as compared to ECMP, the times between the reinforced method and ECMP were compared. From this result, it is shown that the job execution time of the reinforced method is shorter than ECMP and its reduction rate is 8.7%. The computation time and the staging operation time of the reinforced method were also shorter than ECMP. The reduction rate of the computation time was 5.6%, which means the overhead of the computation incurred by the reinforced method was negligible, and the rate of the staging operation time was 22.0%, which means the acceleration of the staging operation was achieved. In short, the reinforced method achieved the acceleration of the staging operation without performance degradation in inter-node communication. Specifically, there was room to reduce the staging operation time under the ECMP method by 31.1% as described in Section 2.5.3, while the reinforced method reduced the staging operation time by 22.0%.

Table 4.6 shows the average of the job throughput calculated from the time taken for each of fifty job submission experiments and the number of submitted jobs. The

Table 4.6: The average of the job throughput in the job submission experiments.

	ECMP	Reinforced method
Job throughput (jobs per hour)	19.06	20.63

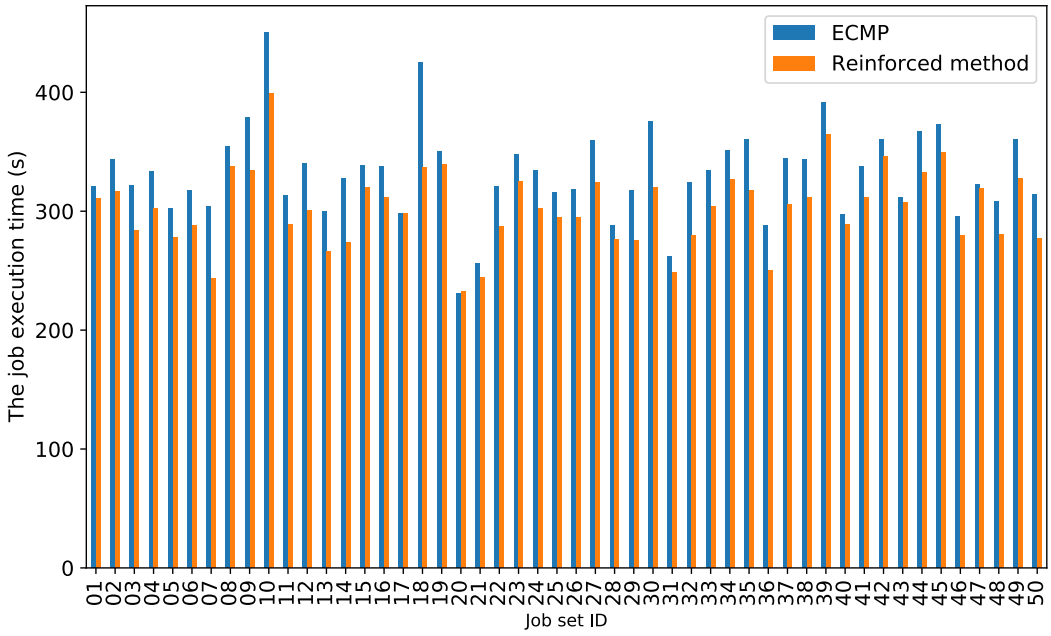


Figure 4.10: The average job execution time for each job set.

calculation way is the same as Equation (2.1) described in Section 2.5.3. This result shows that the job throughput under the reinforced method is higher than that under the ECMP method. In detail, the reinforced method improved the job throughput by 7.6% in comparison with the ECMP method. This fact means that the acceleration of the staging operation without performance degradation in inter-node communication resulted in the improvement in job throughput.

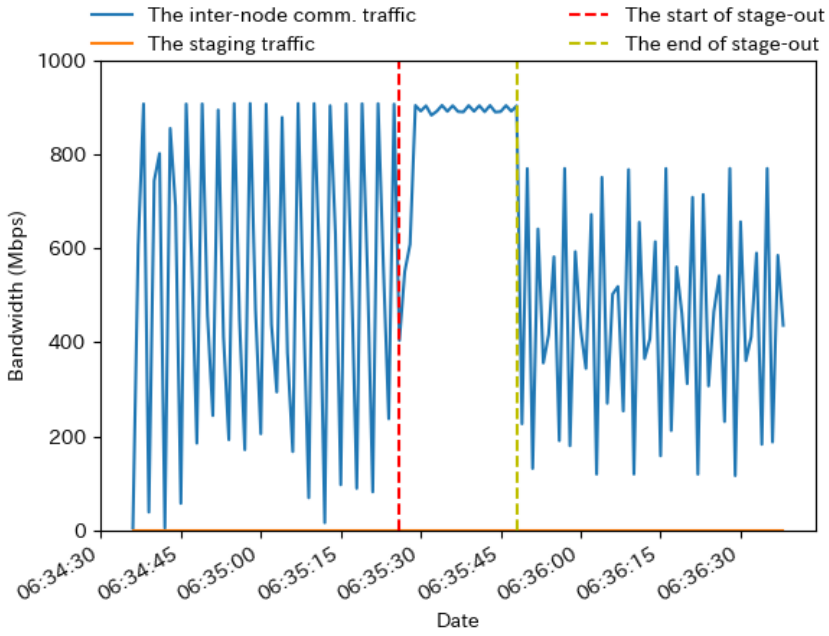
Figure 4.10 shows the result for each job set submitted in the job submission experiment. Each bar in this result is the average job execution time of the six jobs in a job set. The graph in this result indicates that the job execution time under the reinforced method is shorter than that under the ECMP method in 45 job sets. This fact indicates that the reinforced method stably improves the job throughput.

Verification of Traffic Collision Avoidance

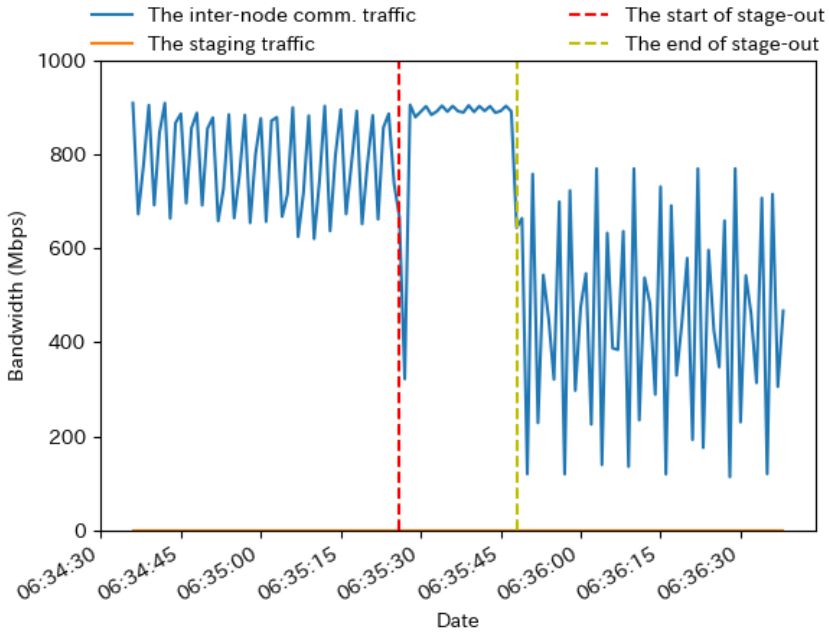
Figure 4.11 shows the bandwidth consumed on each port of the switches during the stage-out operation of a job in a job set when the reinforced method was adopted for the traffic control method on the interconnect. In Fig. 4.11, the Y-axis indicates the bandwidth consumed by each type of traffic, and the X-axis indicates time. These graphs indicate that the dynamic path switching controller configures a set of link-separation-based paths for each type of traffic properly while the stage-out operation occurs as follows. Until 06:35:26, only the inter-node communication traffic flowed on all the ports. Under this situation, the stage-out operation started at 06:35:26 and finished at 06:35:48. The change in the bandwidth from before the start of this stage-out operation to during this stage-out operation is illustrated as follows. First, Fig. 4.11 (a) shows that the bandwidth used by the inter-node communication traffic was raised from 519 Mbps to 845 Mbps on port1 on ofs1. Second, Fig. 4.11 (b) shows that the bandwidth used by the inter-node communication traffic was raised from 784 Mbps to 849 Mbps on port2 on ofs1 during this stage-out operation. Third, Fig. 4.11 (c) shows that the bandwidth used by the inter-node communication traffic on port1 on ofs2 was lowered to 42 Mbps from 522 Mbps, and the bandwidth used by the staging traffic on port1 on ofs2 was raised from 0 Mbps to 721 Mbps. Finally, Fig. 4.11 (d) shows that the bandwidth used by the inter-node communication traffic on port2 on ofs2 was lowered to 30 Mbps from 260 Mbps. These observation results indicate that ofs2 was used for the link-separation-based paths of the staging traffic and ofs1 was used for the link-separation-based paths of the inter-node communication traffic during this stage-out operation. The reason why the bandwidth used by the inter-node communication traffic on port1 and port2 on ofs2 was not 0 Mbps even though ofs2 was used for the link-separation-based paths of the staging traffic is that the inter-node communication traffic flowed there for approximately 2 seconds before the dynamic path switching controller allocated the link-separation-based paths for each type of traffic. After the stage-out operation finished, only the inter-node communication traffic flowed on all the ports again. In this way, the reinforced method avoids the traffic collision between each type of traffic dynamically. This path switching was observed in the staging operations of the other jobs in the job set.

Operational Verification of Progressive Path Switching

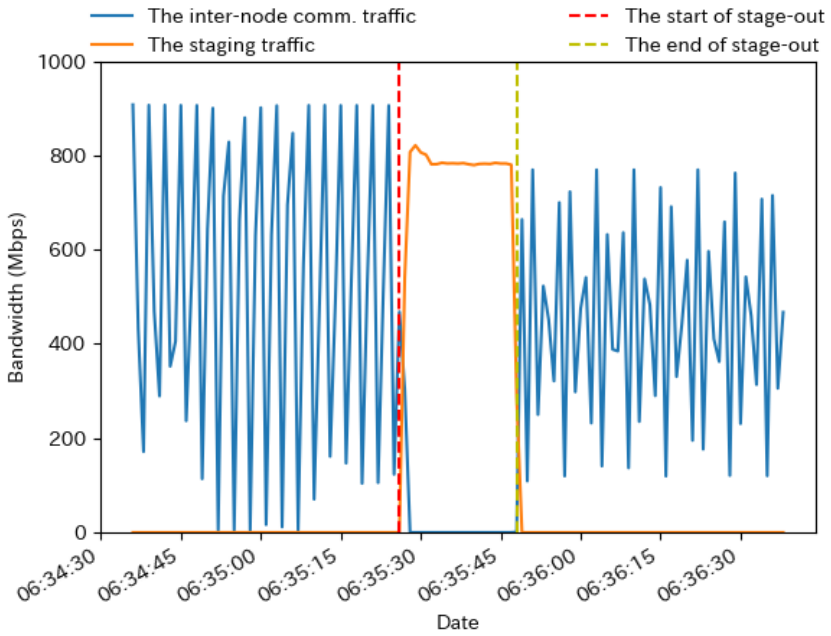
Figure 4.12 shows the average times of the progressive path switching between ON and OFF status. The path switching time from OFF status to ON status is from when the



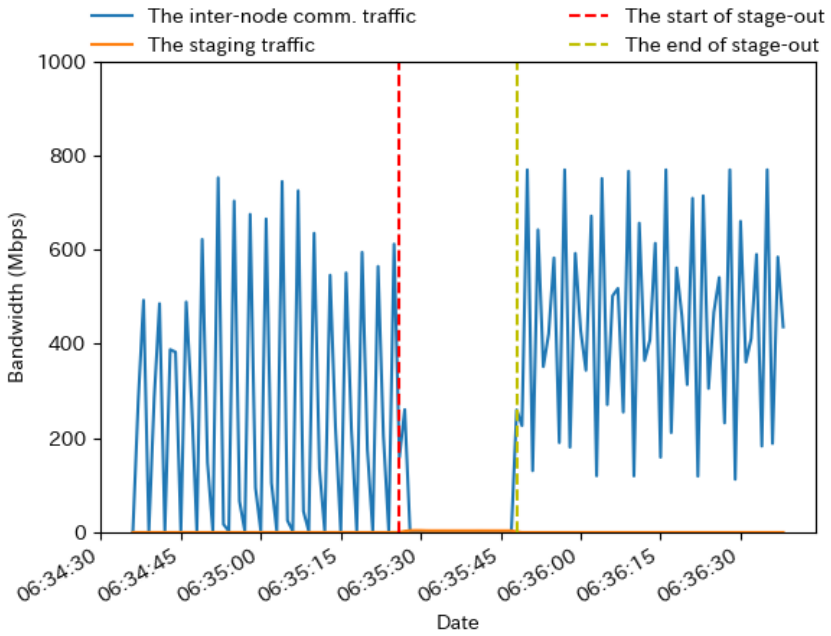
(a) The bandwidth consumed by each traffic on port1 on ofs1.



(b) The bandwidth consumed by each traffic on port2 on ofs1.



(c) The bandwidth consumed by each traffic on port1 on ofs2.



(d) The bandwidth consumed by each traffic on port2 on ofs2.

Figure 4.11: The bandwidth consumed on each port of the switches during a stage-out operation under ON status.

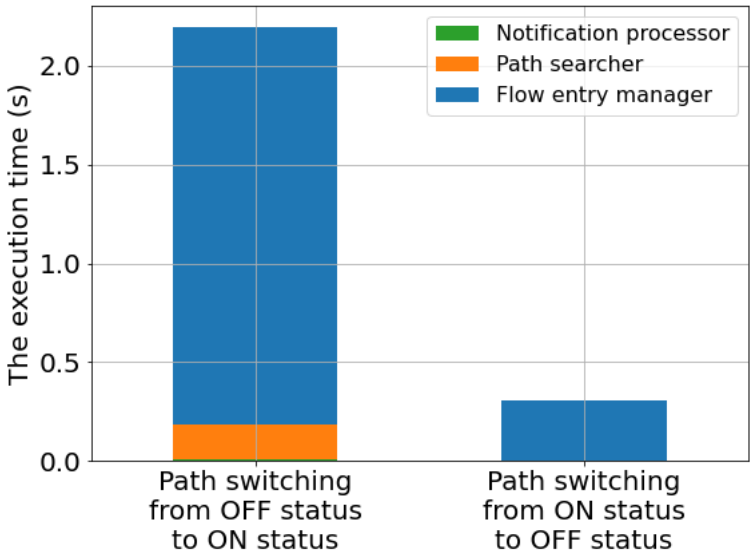
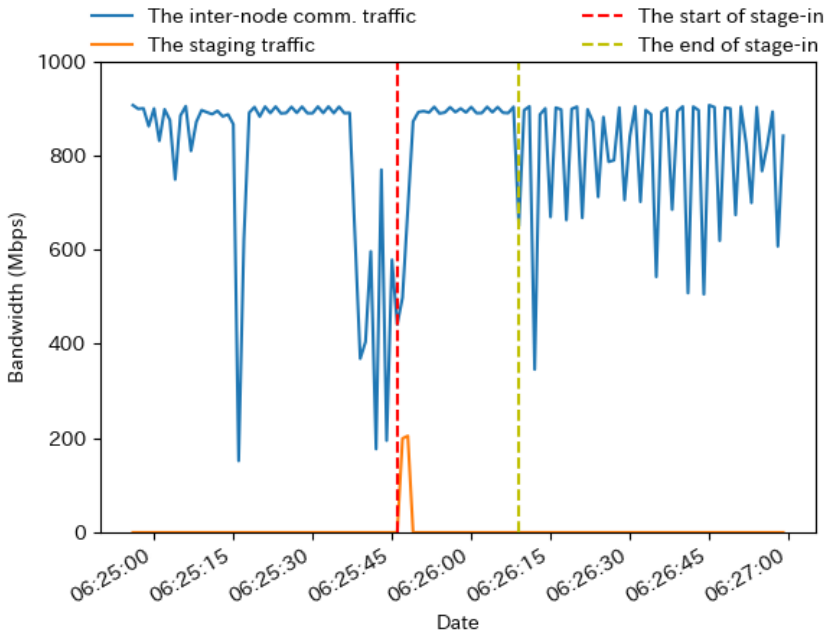


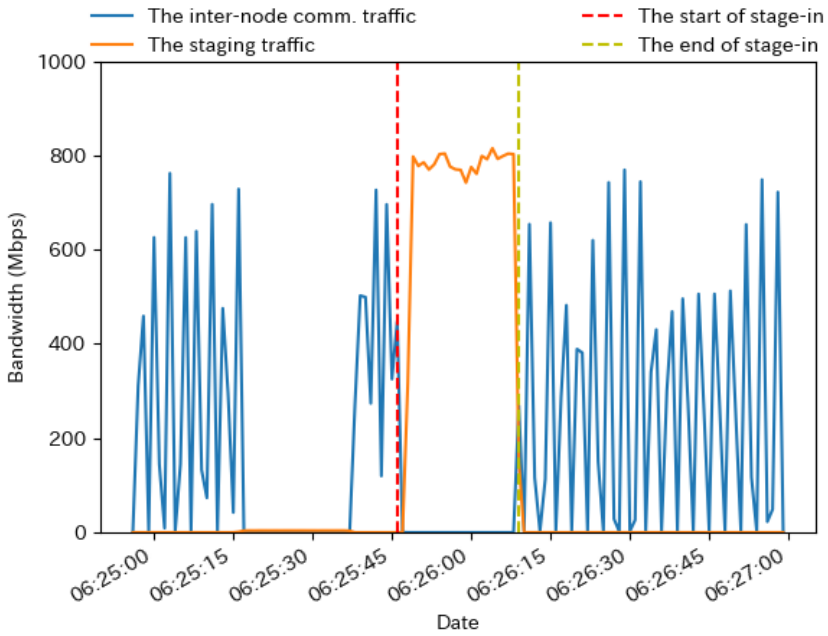
Figure 4.12: The average execution time of the progressive path switching between ON and OFF status.

notificator module informs the dynamic path switching controller that a staging operation starts until when the controller completes the flow entry installs, and the path switching time from ON status to OFF status is from when the notificator module informs the controller that a staging operation finishes until when the controller completes the flow entry deletions. In this graph, the path switching time from OFF status to ON status is plotted in the stacked bar graph composed of the processing times in the notification processor module, the path searcher module and the flow entry manager module. From this graph, it is shown that there was a 2.20 seconds delay to complete the path switching from the ECMP-based paths to the link-separation-based paths, while there was a 0.31 seconds delay to complete the path switching from the link-separation-based paths to the ECMP-based paths. Moreover, the overhead time to complete the path switching from OFF status to ON status was 0.0085 seconds for the notification processor module, 0.17 seconds for the path searcher module and 2.01 seconds for the flow entry manager module. The breakdown of the overhead time shows that the flow entry installs on the flow entry manager module incurs the longest delay in the path switching from OFF status to ON status. On the other hand, the path switching time from ON status to OFF status is predominantly occupied by the flow entry deletions on the flow entry manager module. The reason for this is because the path searcher module is not performed in this path switching and the notification processor module takes a few milliseconds as shown above.

However, each type of traffic can continue to flow on the interconnect until the path switching completes because the progressive path switching works as shown in Fig. 4.13. The stage-in operation observed in these graphs started at 06:25:46 and finished at 06:26:09. Figure 4.13 (a) shows the staging traffic flows on port2 of ofs1 at 202 Mbps from 06:25:47 to 06:25:48. After that, the staging traffic flows on port2 of ofs2 at 735 Mbps from 06:25:49 to 06:26:09 as shown in Fig. 4.13 (b), and the inter-node communication traffic flows on port2 of ofs1 as shown in Fig. 4.13 (a). In this way, the progressive path switching function enables each type of traffic to pass through the ECMP-based paths in 2.20 seconds on average until the path switching completes and suppresses the effect of the delay in the path switching from OFF status to ON status to the job throughput.



(a) The bandwidth consumed by each traffic on port2 on ofs1.



(b) The bandwidth consumed by each traffic on port2 on ofs2.

Figure 4.13: The bandwidth consumed on port2 of each switch during a stage-in operation under ON status.

4.5 Conclusion

Taking the instability and fluctuation of communication latency and delay in an actual environment into consideration, this chapter reinforced the DLSCA proposed in Chapter 3. The reinforced method dynamically switches different types of traffic control depending on whether the staging operation occurs in a progressive way. Even while the path switching is being performed, the staging operation and inter-node communication seamlessly continue without being blocked. Also, the reinforced method configures link-separation-based paths with fewer flow entries than the original DLSCA while a staging operation occurs (ON status). On the other hand, it configures ECMP-based paths for the inter-node communication traffic while no staging operation happens (OFF status).

Next, the reinforced method was evaluated using an actual HPC cluster system in this chapter. The evaluation result indicates the reinforced method succeeded in avoiding traffic collision between each type of traffic and could accelerate the staging operation by 22.0% on the actual HPC cluster system. Also, the evaluation indicates the overhead of the computation incurred by the reinforced method is negligible. Furthermore, it is shown that 7.6% of the job throughput was improved by the reinforced method. Based on the result of this experiment performed on an actual HPC cluster system, the dynamic traffic control method in conjunction with the staging operation is practical. In other words, the dynamic traffic control method in conjunction with the staging operation can achieve a staging operation with little performance degradation in inter-node communication and can improve the job throughput.

In future research, the progressive switching-based DLSCA should provide a new direction for improving the performance in inter-node communication. In detail, adopting SDN-based traffic control method, such as DLB and SDN-enhanced MPI described in Section 3.5, for the traffic control of the reinforced method in OFF status instead of ECMP may lead to the performance improvement in inter-node communication.

Furthermore, the reinforced method can be enhanced considering the case where a computing node is shared by multiple jobs in a time-sharing manner, which this study did not assume. Note that this study assumed a space-sharing manner as described in Section 1.2. In this case, the two types of traffic can simultaneously flow on the same interconnect links between computing nodes and edge switches. This is because some processes proceeding computation and other processes proceeding the staging operation can be deployed on one and the same computing nodes at the same time. Therefore, the situation where each type of traffic may block each other on the network interface of a

computing node may need to be considered for the acceleration of the staging operation.

Moreover, the core switch allocation algorithm of the reinforced method must be enhanced towards the large-scale HPC cluster systems which have more than three core switches. The reinforced method has adopted a simple way to equally divide core switches into the two groups. However, there is room for improving this division method as mentioned in Section 4.3.1. Note that the original DLSCA has an example of an improved algorithm for this division method as described in Section 3.3.1, although how efficiently the improved algorithm works has not been investigated yet.

5 Conclusion

5.1 Concluding Remarks

On the modern HPC cluster systems with a two-layered file system, the staging operation is essential to satisfy the researchers' requirements for storage capacity and I/O performance. This staging operation moves input and output data between a local file system and a global file system before and after the computation of a scientific application. Therefore, the rapid increase in the size of data treated in today's scientific research strongly requires the acceleration of the staging operation for job throughput improvement. However, few studies approach improving the performance of staging communication. Therefore, this dissertation tackled the acceleration of the staging operation by focusing on staging communication, targeting HPC cluster systems with an oversubscribed fat-tree interconnect shared by computing nodes and storages.

To achieve acceleration of the staging operation, this dissertation has tackled the following three issues: (1) revealing how the staging operation is slowed down on interconnect, (2) exploring an approach to accelerate the staging operation, and (3) verifying the practicality of the explored approach.

To address the first issue, Chapter 2 investigated how the staging operation is slowed down on interconnect and how slowed staging operation degrades the job throughput. For this investigation, this dissertation speculated that the traffic collision between inter-node communication traffic and staging traffic prolongs the staging operation time on the target class of HPC cluster systems. Verifying the correctness of this speculation requires a packet-monitoring that observes how the bandwidth consumed by each type of traffic changes in association with a staging operation. For the purpose of this verification, in this packet-monitoring, the researchers had to capture a series of packets during a job submission experiment, calculate consumed bandwidth from a large amount of captured packets, and visualize it in association with a staging operation. However, each of the job submission experiments and consumed bandwidth calculations took a much longer time, and there is no existing packet-monitoring tool available for automating this

packet-monitoring. As a result, the researchers had difficulty in manually performing this packet-monitoring. Therefore, this dissertation has developed a packet-monitoring tool that automatically performs this packet-monitoring. The tool is composed of packet-capture, consumed bandwidth calculation, and visualization to this investigation efficient. With this packet-monitoring tool, this dissertation revealed that the traffic collision between each type of traffic slows the staging operation through a job submission experiment. Specifically, this investigation showed that the traffic collision increased the staging operation time by 45.2% and degraded the job throughput by 6.4%.

To address the second issue, Chapter 3 presented a dynamic traffic control method in conjunction with the staging operation as an approach to avoid the traffic collision between inter-node communication traffic and staging traffic. This chapter first investigated whether the static traffic control method, which is usually adopted in traditional interconnects, can accelerate the staging operation or not. As a result, it was speculated that the static traffic control method might be difficult to use in accelerating the staging operation without performance degradation in inter-node communication. Based on this speculation, as an approach, this dissertation has adopted a dynamic traffic control method in conjunction with the staging operation, which switches different types of traffic controls depending on whether any staging operation occurs. This approach aimed to accelerate staging operation with little performance degradation in inter-node communication. To verify the possibility that this approach can be realized, this dissertation proposed and prototyped a dynamic traffic control method that separates the interconnect links to the links used by the inter-node communication traffic and the links used by the staging traffic when any staging operation is running. The evaluation conducted in this dissertation showed that the proposed method is feasible through an experiment performed in a virtual simulation environment. Also, the evaluation showed that the DLSCA succeeded in shortening the staging operation time by 7.50% and the job execution time by 3.40% as compared to ECMP.

To address the third issue, Chapter 4 verified that the dynamic traffic control method in conjunction with the staging operation contributes to the acceleration of the staging operation and the job throughput improvement, by performing a job submission experiment on an actual HPC cluster system. Also, this chapter attempted to reinforce the DLSCA by considering that communication latency and delay in an actual environment are not ideal as expected in Chapter 3. Specifically, this dissertation had to tackle the undesirable problem that the communication latency and delay might make the OpenFlow controller's installation and elimination of flow entries fluctuate and become unstable. To address

this problem, this chapter proposed the progressive switching-based DLSCA (reinforced method), which was reinforced from the DLSCA in terms of the following two points. First, the reinforced method performs the path switching in a progressive way without blocking any traffic to reduce the negative influence of the longer path switching. Second, the reinforced method configures link-separation-based paths on the interconnect with fewer flow entries than the original so that the OpenFlow controller takes less time to configure link-separation-based paths. The evaluation result showed that the progressive switching-based DLSCA is capable of avoiding a traffic collision between each type of traffic and can accelerate the staging operation by 22.0%. Also, the evaluation indicates that the overhead of the application incurred by the reinforced method is negligible. Furthermore, 7.6% of the job throughput was improved by the reinforced method. Based on this result, the dynamic traffic control method in conjunction with the staging operation is considered practical for accelerating the staging operation with little performance degradation in inter-node communication and improving the job throughput.

5.2 Future Direction

There are two directions for the future of this study.

The progressive switching-based DLSCA was evaluated using small-scale HPC cluster systems because large-scale HPC cluster systems were not available for the evaluation. Although the progressive switching-based DLSCA was designed and implemented so that it can work on large-scale HPC cluster systems, it is not clear how the progressive switching-based DLSCA actually works on the large-scale HPC cluster systems. Therefore, the progressive switching-based DLSCA needs to be investigated on the larger-scale HPC cluster systems for further improvement in practicality and usefulness of the research achievement in this dissertation. Conducting this investigation requires the technical challenge of reducing a large number of flow entries because the volume of flow entries installed to a switch must be less than the memory capacity of the switch. When the volume of flow entries exceeds the memory capacity of the switch, the progressive switching-based DLSCA cannot install new flow entries to the switch and therefore, cannot control traffic properly. Although reducing the number of flow entries was tackled in Chapter 4, further efforts and elaboration on the management of flow entries are required for progressive switching-based DLSCA on large-scale HPC cluster systems. Enhancing the progressive switching-based DLSCA to merge redundant flow entries and then using fewer flow entries to switch traffic controls might be a possible solution to the

management of flow entries.

In this dissertation, the progressive switching-based DLSCA was evaluated when traffic collision between each type of traffic frequently happens. The evaluation aimed to verify that the dynamic traffic control method in conjunction with the staging operation can accelerate the staging operation by avoiding traffic collision. Although it was revealed that this dynamic traffic control contributes to the acceleration of the staging operation, how this dynamic traffic control method works where practical HPC applications run still needs to be investigated. Towards further verification in this investigation, the progressive switching-based DLSCA has to be enhanced to switch traffic controls by considering how much each type of traffic flows on interconnect because the progressive switching-based DLSCA switches from ECMP-based paths to link-separation-based paths even when a small amount of staging traffic flows on interconnect, assuming the traffic collision occurs whenever the staging operation happens.

Acknowledgements

First, I would like to thank my supervisor Professor Shinji Shimojo at the Cybermedia Center, Osaka University. This study benefited from the countless suggestions based on his broad and deep knowledge.

I would like to express my gratitude to Professor Toru Fujiwara and Professor Makoto Onizuka at the Department of Multimedia Engineering, the Graduate School of Information Science and Technology, Osaka University, for offering insightful and valuable comments to revise this doctoral dissertation.

I would like to thank Professor Takahiro Hara and Professor Yasuyuki Matsushita at the Department of Multimedia Engineering, the Graduate School of Information Science and Technology, Osaka University for their thoughtful guidance.

I owe my deepest gratitude to Associate Professor Susumu Date at the Cybermedia Center, Osaka University, for his invaluable advice on my studies. This excellent guidance based on his deep understanding of science helped me grows greatly. Without his enthusiastic support, I could not complete this dissertation.

I am grateful to Dr. Chunghan Lee at the Toyota Motor Cooperation, Dr. Hiroki Ohtsuji, Mr. Eiji Yoshida, Mr. Katsuhito Asano, Ms. Erika Hayashi at the Fujitsu Laboratories Ltd., Associate Professor Yoshiyuki Kido at the Cybermedia Center, Osaka University, and Associate Professor Yasuhiro Watashiba at the Department of Information Science, University of Fukui, for their invaluable knowledge sharing and insightful questions. Through discussions with them, the quality of this thesis was improved much further.

Sincere thanks also go to Associate Professor Hirotake Abe at the University of Tsukuba, 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 at the National Institute of Informatics, Guest Professor Takashi Yoshikawa, and Specially Appointed Associate Professor Chonho Lee at the Cybermedia Center, Osaka University, for their constructive comments and advice.

I give special gratitude to all students at Shimojo Laboratory for the enjoyable time we casually debated various topics together. The study activities with them were an

enormous pleasure for me.

Finally, I would like to express my deepest gratitude and appreciation to my parents, Motoko Endo and Ryusuke Endo, for their courteous support and continuous encouragement. They made me what I am today.

References

[1] R. J. Small, J. Bacmeister, D. Bailey, A. Baker, S. Bishop, F. Bryan, J. Caron, J. Dennis, P. Gent, H. Hsu, M. Jochum, D. Lawrence, E. Muñoz, P. DiNezio, T. Scheitlin, R. Tomas, J. Tribbia, Y. Tseng, and M. Vertenstein, "A New Synoptic Scale Resolving Global Climate Simulation Using the Community Earth System Model", *Journal of Advances in Modeling Earth Systems*, vol. 6, no. 4, pp. 1065–1094, Dec. 2014. DOI: 10.1002/2014MS000363.

[2] O. Fuhrer, T. Chadha, T. Hoefler, G. Kwasniewski, X. Lapillonne, D. Leutwyler, D. Lüthi, C. Osuna, C. Schär, T. C. Schulthess, and H. Vogt, "Near-Global Climate Simulation at 1 km Resolution: Establishing a Performance Baseline on 4888 GPUs with COSMO 5.0", *Geoscientific Model Development*, vol. 11, no. 4, pp. 1665–1681, May 2018. DOI: 10.5194/gmd-11-1665-2018.

[3] N. Okimoto, N. Futatsugi, H. Fuji, A. Suenaga, G. Morimoto, R. Yanai, Y. Ohno, T. Narumi, and M. Taiji, "High-Performance Drug Discovery: Computational Screening by Combining Docking and Molecular Dynamics Simulations", *PLoS Computational Biology*, vol. 5, no. 10, e1000528, Oct. 2009. DOI: 10.1371/journal.pcbi.1000528.

[4] H. Ge, Y. Wang, C. Li, N. Chen, Y. Xie, M. Xu, Y. He, X. Gu, R. Wu, Q. Gu, L. Zeng, and J. Xu, "Molecular Dynamics-Based Virtual Screening: Accelerating the Drug Discovery Process by High-Performance Computing", *Journal of Chemical Information and Modeling*, vol. 53, no. 10, pp. 2757–2764, Oct. 2013. DOI: 10.1021/ci400391s.

[5] S. LeGrand, A. Scheinberg, A. F. Tillack, M. Thavappiragasam, J. V. Vermaas, R. Agarwal, J. Larkin, D. Poole, D. Santos-Martins, L. Solis-Vasquez, A. Koch, S. Forli, O. Hernandez, J. C. Smith, and A. Sedova, "GPU-Accelerated Drug Discovery with Docking on the Summit Supercomputer", in *Proceedings of the 11th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics*, Sep. 2020, pp. 1–10. DOI: 10.1145/3388440.3412472.

[6] H.-J. Yoon, J. Gounley, M. T. Young, and G. Tourassi, "Information Extraction from Cancer Pathology Reports with Graph Convolution Networks for Natural Language Texts", in *Proceedings of the 7th IEEE International Conference on Big Data*, Dec. 2019, pp. 4561–4564. DOI: 10.1109/BigData47090.2019.9006270.

[7] A. Ferrario and M. Naegelin, "The Art of Natural Language Processing: Classical, Modern and Contemporary Approaches to Text Document Classification", *SSRN Electronic Journal*, pp. 1–51, Mar. 2020. DOI: 10.2139/ssrn.3547887.

[8] S. Chibani and F. Coudert, "Machine Learning Approaches for the Prediction of Materials Properties", *APL Materials*, vol. 8, no. 8, p. 080 701, Aug. 2020. DOI: 10.1063/5.0018384.

[9] A. Dunn, Q. Wang, A. Ganose, D. Dopp, and A. Jain, "Benchmarking Materials Property Prediction Methods: the Matbench Test Set and Automatminer Reference Algorithm", *npj Computational Materials*, vol. 6, no. 1, p. 138, Dec. 2020. DOI: 10.1038/s41524-020-00406-3.

[10] D. Castelvecchi, "How to Hunt for a Black Hole with a Telescope the Size of Earth", *Nature News*, vol. 543, no. 7646, pp. 478–480, Mar. 2017. DOI: 10.1038/543478a.

[11] K. Akiyama, A. Alberdi, W. Alef, *et al.*, "First M87 Event Horizon Telescope Results. II. Array and Instrumentation", *The Astrophysical Journal*, vol. 875, no. 1, p. L2, Apr. 2019. DOI: 10.3847/2041-8213/ab0c96.

[12] H. W. Meuer, E. Strohmaier, J. Dongarra, H. Simon, and M. Meuer, *TOP500: TOP500 Supercomputer Sites*. [Online]. Available: https://www.top500.org (visited on 11/27/2020).

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

[14] 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. 2002, pp. 616–632. DOI: 10.1109/9780470544839.ch42.

[15] M. P. I. Forum, *MPI: A Message-Passing Interface standard*. [Online]. Available: https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (visited on 02/19/2021).

[16] 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.

[17] 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. 2427, Springer Berlin Heidelberg, Sep. 2002, p. 7. DOI: 10.1007/3-540-45825-5_5.

[18] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management", in *Proceedings of the 9th Job Scheduling Strategies for Parallel Processing*, Jun. 2003, pp. 44–60. DOI: 10.1007/10968987_3.

[19] R. L. Henderson, "Job scheduling under the Portable Batch System", in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 949, Springer Berlin Heidelberg, Apr. 1995, pp. 279–294. DOI: 10.1007/3-540-60153-8_34.

[20] Adaptive Computing Enterprise, Inc., *Torque 6.0.4 Administrator Guide*. [Online]. Available: http://docs.adaptivecomputing.com/torque/6-0-4/adminGuide/torquehelp.htm (visited on 01/21/2021).

[21] S. Hofmeyr, C. Iancu, J. Colmenares, E. Roman, and B. Austin, "Time-Sharing Redux for Large-Scale HPC Systems", in *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications; the 14th IEEE International Conference on Smart City; the 2nd IEEE International Conference on Data Science and Systems*, Jan. 2016, pp. 301–308. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0051.

[22] O. H. Mondragon, P. G. Bridges, and T. Jones, "Quantifying Scheduling Challenges for Exascale System Software", in *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, Jun. 2015, pp. 1–8. DOI: 10.1145/2768405.2768413.

[23] P. J. Braam and P. Schwan, "Lustre: The Intergalactic File System", in *Proceedings of the 4th Ottawa Linux Symposium*, Jun. 2002, pp. 50–54. [Online]. Available: https://www.kernel.org/doc/ols/2002/ols2002-pages-50-54.pdf (visited on 12/10/2020).

[24] *Gluster*. [Online]. Available: http://gluster.org (visited on 01/19/2021).

[25] F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters", no. January, pp. 231–244, Jan. 2002. [Online]. Available: https://cug.org/proceedings/cug2018_proceedings/includes/files/pap119s2-file1.pdf (visited on 03/25/2021).

[26] F. Herold and S. Breuner, "An Introduction to BeeGFS", ThinkParQ, Tech. Rep., Jun. 2018. [Online]. Available: https://www.beegfs.io/docs/whitepapers/Introduction_to_BeeGFS_by_ThinkParQ.pdf (visited on 02/19/2021).

[27] J. Shalf, "The Future of Computing beyond Moore's Law", *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2166, p. 20 190 061, Mar. 2020. DOI: 10.1098/rsta.2019.0061.

[28] J. Dongarra, "Report on the Fujitsu Fugaku system", University of Tennessee-Knoxville Innovative Computing Laboratory, Tech. Rep. ICL-UT-20-06, Jun. 2020. [Online]. Available: http://performance.netlib.org/utk/people/JackDongarra/PAPERS/icl-utk-1379-2020.pdf (visited on 12/10/2020).

[29] *TSUBAME 3.0*. [Online]. Available: https://www.t3.gsic.titech.ac.jp (visited on 03/25/2021).

[30] J. Hines, "Stepping Up to Summit", *Computing in Science & Engineering*, vol. 20, no. 2, pp. 78–82, Mar. 2018. DOI: 10.1109/MCSE.2018.021651341.

[31] B. Landsteiner and D. Pau, "DataWarp Transparent Cache: Implementation, Challenges, and Early Experience", in *Proceedings of the 61st Cray User Group Conference*, 2018, pp. 1–10. [Online]. Available: http://users.cis.fiu.edu/~pand/publications/13icc-yu.pdf (visited on 01/19/2021).

[32] B. Nitzberg, J. M. Schopf, and J. P. Jones, "PBS Pro: Grid Computing and Scheduling Attributes", in *Grid Resource Management*, ser. International Series in Operations Research & Management Science, vol. 64, Springer Boston MA, 2004, pp. 183–190. DOI: 10.1007/978-1-4615-0509-9_13.

[33] A. Ovsyannikov, M. Romanus, B. V. Straalen, G. H. Weber, and D. Trebotich, "Scientific Workflows at DataWarp-Speed: Accelerated Data-Intensive Science Using NERSC's Burst Buffer", in *Proceedings of the 1st Joint International Workshop on Parallel Data Storage and data Intensive Scalable Computing Systems*, Nov. 2016, pp. 1–6. DOI: 10.1109/PDSW-DISCS.2016.005.

[34] M. Vef, N. Moti, T. SuB, T. Tocci, R. Nou, A. Miranda, T. Cortes, and A. Brinkmann, "GekkoFS - A Temporary Distributed File System for HPC Applications", in *Proceedings of the 20th IEEE International Conference on Cluster Computing*, vol. 2018-Septe, Sep. 2018, pp. 319–324. DOI: 10.1109/CLUSTER.2018.00049.

[35] A. Hori, K. Yamamoto, and Y. Ishikawa, "Catwalk-ROMIO: A Cost-Effective MPI-IO", in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, Dec. 2011, pp. 120–126. DOI: 10.1109/ICPADS.2011.40.

[36] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture", in *Proceedings of the 8th ACM SIGCOMM Conference on Data Communication*, Oct. 2008, pp. 63–74. DOI: 10.1145/1402946.1402967.

[37] National Institute of Advanced Industrial Science and Technology (AIST), *ABCI: AI Bridging Cloud Infrastructure*. [Online]. Available: https://abci.ai (visited on 12/04/2020).

[38] A. Orebaugh, G. Ramirez, and J. Beale, *Wireshark & Ethereal Network Protocol Analyzer Toolkit*. Syngress Publishing, Dec. 2006. DOI: 10.1016/B978-1-59749-073-3.X5000-3.

[39] G. Combs, *Tshark - Dump and Analyze Network Traffic*. [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html (visited on 01/19/2021).

[40] L. M. Garcia, *TCPDUMP/LIBPCAP Public Repository*. [Online]. Available: https://www.tcpdump.org (visited on 01/19/2021).

[41] B. Claise, "Cisco Systems Netflow Services Export Version 9", *RFC 3954*, Oct. 2004. DOI: 10.17487/RFC3954.

[42] P. Phaal, S. Panchen, and N. McKee, "InMon Corporation's sFlow: A Method for Monitoring Traffic in Switched and Routed Networks", *RFC 3716*, Sep. 2001. DOI: 10.17487/RFC3176.

[43] SchedMD LLC, *Slurm Workload Manager - squeue*. [Online]. Available: https://slurm.schedmd.com/squeue.html (visited on 01/30/2021).

[44] M. Rocklin, "Dask: Parallel Computation with Blocked Algorithms and Task Scheduling", in *Proceedings of the 14th Python in Science Conference*, vol. 126, Jul. 2015, pp. 136–132. [Online]. Available: https://conference.scipy.org/proceedings/scipy2015/pdfs/matthew_rocklin.pdf (visited on 01/25/2021).

[45] W. McKinney, "pandas: a Foundational Python Library for Data Analysis and Statistics", in *Proceedings of the 1st Workshop on Python for High Performance and Scientific Computing*, vol. 14, Nov. 2011, pp. 1–9. [Online]. Available: https://www.dlr.de/sc/portaldata/15/resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf (visited on 01/19/2021).

[46] J. D. Hunter, "Matplotlib: A 2D Graphics Environment", *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, May 2007. DOI: 10.1109/MCSE.2007.55.

[47] D. Thaler and C. E. Hopps, "Multipath Issues in Unicast and Multicast Next-Hop Selection", *RFC 2991*, Nov. 2000. DOI: 10.17487/RFC2991.

[48] J. Moy, "OSPF Version 2", *RFC 2328*, Apr. 1998. DOI: 10.17487/RFC2328.

[49] Ryu SDN Framework Community, *Ryu SDN Framework*. [Online]. Available: https://github.com/osrg/ryu (visited on 01/19/2021).

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

[51] J. Bruck, C. Ho, S. Kipnis, and D. Weathersby, "Efficient Algorithms for All-to-All Communications in Multi-Port Message-Passing Systems", in *Proceedings of the 6th annual ACM symposium on Parallel Algorithms and Architectures*, Nov. 1994, pp. 298–309. DOI: 10.1145/181014.181756.

[52] P. Snyder, "tmpfs: A Virtual Memory File System", in *Proceedings of the 9th autumn European Unix Systems User Group Conference*, Oct. 1990, pp. 241–248. [Online]. Available: http://www.sunhelp.org/history/pdf/tmpfs.pdf (visited on 03/25/2021).

[53] L. A. Steffenel, M. Martinasso, and D. Trystram, "Assessing Contention Effects on MPI_Alltoall Communications", in *Advances in Grid and Pervasive Computing*, ser. Lecture Notes in Computer Science, vol. 4459, Springer Berlin Heidelberg, May 2007, pp. 424–435. DOI: 10.1007/978-3-540-72360-8_36.

[54] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey", *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan. 2015. DOI: 10.1109/JPROC.2014.2371999.

[55] *Floodlight OpenFlow Controller*. [Online]. Available: https://github.com/floodlight/floodlight (visited on 02/19/2021).

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

[57] *Trema: Full-Stack OpenFlow Framework in Ruby/C*. [Online]. Available: https://trema.github.io/trema/ (visited on 01/19/2021).

[58] *OpenFlow Switch Specification: Version 1.5.1*. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (visited on 03/25/2021).

[59] V. Z. Attar and P. Chandwadkar, "Network Discovery Protocol LLDP and LLDP-MED", *International Journal of Computer Applications*, vol. 1, no. 9, pp. 93–97, Feb. 2010. DOI: 10.5120/207-348.

[60] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function Using NetworkX", in *Proceedings of the 7th Python in Science Conferences*, vol. 2008, Jan. 2008, pp. 11–15. [Online]. Available: http://conference.scipy.org/proceedings/scipy2008/paper_2/full_text.pdf (visited on 03/25/2021).

[61] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: The Linux Virtual Machine Monitor", in *Proceedings of the 9th Ottawa Linux Symposium*, Jun. 2007, pp. 225–230. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf (visited on 01/19/2021).

[62] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, *et al.*, "The Design and Implementation of Open vSwitch", in *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation*, May 2015, pp. 117–130. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff (visited on 01/19/2021).

[63] W. L. Guay, B. Bogdanski, S. A. Reinemo, O. Lysne, and T. Skeie, "vFtree - A Fat-Tree Routing Algorithm using Virtual Lanes to Alleviate Congestion", in *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium*, May 2011, pp. 197–208. DOI: 10.1109/IPDPS.2011.28.

[64] R. Rajachandrasekar, J. Jaswani, H. Subramoni, and D. K. Panda, "Minimizing Network Contention in InfiniBand Clusters with a QoS-Aware Data-Staging Framework", in *Proceedings of the 14th IEEE International Conference on Cluster Computing*, Sep. 2012, pp. 329–336. DOI: 10.1109/CLUSTER.2012.90.

[65] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic Flow Scheduling for Data Center Networks", in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, vol. 10, Apr. 2010, p. 19. DOI: 10.5555/1855711.1855730.

[66] Y. Li and D. Pan, "OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support", in *Proceedings of the 12th IEEE International Conference on Communications*, Jun. 2013, pp. 1–5. [Online]. Available: http://users.cis.fiu.edu/~pand/publications/13icc-yu.pdf (visited on 01/19/2021).

[67] R. Dewanto, R. Munadi, and R. M. Negara, "Improved Load Balancing on Software Defined Network-based Equal Cost Multipath Routing in Data Center Network", *Jurnal Infotel*, vol. 10, no. 3, pp. 157–162, Aug. 2018. DOI: 10.20895/infotel.v10i3.379.

[68] 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. 01, Jun. 2016. [Online]. Available: http://www.intjit.org/cms/journal/volume/22/1/221_5.pdf (visited on 01/19/2021).

[69] Y. Watashiba, S. Date, H. Abe, Y. Kido, K. Ichikawa, H. Yamanaka, E. Kawai, S. Shimojo, and H. Takemura, "Efficacy Analysis of a SDN-enhanced Resource Management System through NAS Parallel Benchmarks", *The Review of Socionetwork Strategies*, vol. 8, no. 2, pp. 69–84, Dec. 2014. DOI: 10.1007/s12626-014-0045-9.

[70] M. Shimizu, Y. Watashiba, S. Date, and S. Shimojo, "Adaptive Network Resource Reallocation for Hot-Spot Avoidance on SDN-based Cluster System", in *Proceedings of the 8th IEEE International Conference on Cloud Computing Technology and Science*, Dec. 2016, pp. 608–613. DOI: 10.1109/CloudCom.2016.0105.

[71] H. Morimoto, K. Dashdavaa, K. Takahashi, Y. Kido, S. Date, and S. Shimojo, "Design and Implementation of SDN-enhanced MPI Broadcast Targeting a Fat-Tree Interconnect", in *Proceedings of the 15th International Conference on High*

Performance Computing & Simulation, Jul. 2017, pp. 252–258. DOI: 10.1109/HPCS.2017.46.

[72] K. 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. April, pp. 23 372–23 382, Apr. 2018. DOI: 10.1109/ACCESS.2018.2829532.