Resource Aware Deep Learning Model Partitioning and Allocation for Inference Task in Clusters With Heterogeneous Graphics Processing Units

Akishige Ikoma, Yuichi Ohsita, *Member, IEEE*, and Masayuki Murata, *Life Member, IEEE*

***Abstract*—Deep learning (DL) models have rapidly evolved, and their scales have become larger. Pipeline parallelism is used to execute a large-scale DL model. In pipeline parallelism, DL models are partitioned and allocated graphics processing units (GPUs) to execute each partition. However, to execute numerous DL services in clusters with heterogeneous GPUs, a DL model partitioning that considers a specific type and number of GPUs available and the GPUs allocated for the service is required. We propose resource aware model partitioning and allocation (RAMPA) to execute more DL services while satisfying the performance requirements. RAMPA minimizes the allocation of important resources for future DL service execution to avoid inhibiting future DL service execution. We define the resource allocation cost based on resource importance. Furthermore, we formulate the impact of model partitioning and allocated resources on service performance. We define an optimization problem to minimize resource allocation costs while satisfying service performance requirements. We evaluated the effectiveness of RAMPA by simulating the execution of DL services in clusters with heterogeneous GPUs. The results demonstrate that more services can be executed while satisfying performance requirements compared to the conventional method. RAMPA enabled efficient GPU utilization to deliver many DL services.**

***Index Terms*—Deep learning (DL) model partitioning, resource allocation, pipeline parallelism, GPU cluster.**

I. Introduction

In RECENT cloud computing, several deep learning (DL) services, such as computer vision, natural language processing, and streaming video processing, have been provided [1]. Accordingly, the DL models used by DL services are rapidly evolving. Large language models with up to 100 billion parameters [2] and vision models with over 10 billion parameters [3] have emerged.

Received 2 October 2024; revised 6 April 2025; accepted 26 October 2025. Date of publication 30 October 2025; date of current version 8 December 2025. This work was supported in part by JST SPRING under Grant JPMJSP2138 and in part by JSPS KAKENHI under Grant JP24K14914. Recommended for acceptance by D. Dechev. *(Corresponding author: Akishige Ikoma.)*

Akishige Ikoma was with the Graduate School of Information Science and Technology, Osaka University, Suita 565-0871, Japan (e-mail: a-ikoma@ist.osaka-u.ac.jp).

Yuichi Ohsita is with the D3 Center, The University of Osaka, Toyonaka 560-0043, Japan (e-mail: yuichi.ohsita.cmc@osaka-u.ac.jp).

Masayuki Murata is with the Graduate School of Information Science and Technology, The University of Osaka, Suita 565-0871, Japan (e-mail: murata@ist.osaka-u.ac.jp).

This article has supplementary downloadable material available at https://doi.org/10.1109/TCC.2025.3626959, provided by the authors.

Digital Object Identifier 10.1109/TCC.2025.3626959

To deliver several DL services, cloud service providers construct graphics processing unit (GPU) clusters. In GPU clusters, multiple GPUs are interconnected through a network. Through cooperation among multiple GPUs, GPU clusters enable the execution of DL services that cannot be executed on a single GPU. Furthermore, owing to the fast release cycle of GPU products, GPU clusters are typically equipped with GPUs with various performances [4]. Hereinafter, this GPU cluster is referred to as cluster with heterogeneous GPUs.

A cluster with heterogeneous GPUs is typically multi-tenant [5] and multiple services require simultaneous execution. However, the demand for cloud DL services is so high that the number of GPUs in a cluster is insufficient [6]. To make matters worse, service providers cannot always configure clusters with plentiful GPUs owing to the rising cost of using GPUs [7]. The efficient use of GPUs in a cluster with heterogeneous GPUs to simultaneously execute many DL services is an important issue to be resolved in the cloud computing field.

During the allocation of execution resources for DL services in a cluster with heterogeneous GPUs, a GPU with sufficient performance is allocated to satisfy the following three performance requirements:

- Throughput requirement: Sufficient throughput to complete all requested DL service tasks.
- Execution latency requirement: Completion of service tasks within an acceptable time for the service users.
- Memory requirement: Sufficient memory capacity of GPU to execute the DL model.

However, if the memory and computing capacity of the GPUs in the cluster are insufficient for the size and computational complexity of the DL model, DL services cannot satisfy the listed performance requirements. Pipeline parallelism is used to address this problem [8]. In pipeline parallelism, a DL model is partitioned, and a GPU is allocated for each partition. Hereinafter, this partitioning is referred to as the pipeline stage. Throughput increases because the input data for DL service can be processed in parallel at each pipeline stage. Furthermore, larger DL models can be executed via the cooperation of multiple GPUs. Therefore, pipeline parallelism is crucial to satisfying the throughput and memory requirements of services that provide inference for streaming data and inference using a large-scale DL model.

Two key points need to be considered when executing many services using pipeline parallelism in a cluster with

heterogeneous GPUs. First, the impact of pipeline parallelism on service performance needs to be considered. Although pipeline parallelism can improve throughput, execution latency increases owing to communication delays between pipeline stages [9]. Furthermore, when the execution resource for the next pipeline stage is processed, the input data from the previous stage wait until the resource becomes available. If the execution latency of one pipeline stage is excessively large, the overall execution latency will be larger.

Second, currently available resources in the cluster need to be considered. If a sufficient number of GPUs with large memory and/or high computing capacity are available, DL service tasks can be executed with fewer GPUs, even on larger-scale DL models. Therefore, if consideration for a specific type and number of GPUs currently available in GPU clusters is lacking, it can result in excess GPU allocations for each service. Consequently, GPUs for executing future services may be rapidly depleted, and the number of services that can be executed simultaneously is constrained.

We propose resource aware model partitioning and allocation(RAMPA). RAMPA aims to minimize the allocation of important resources for future DL service execution to avoid inhibiting future DL service execution. We define the resource allocation cost for GPU and network links in terms of the resource importance. Furthermore, we formulate the impact of model partitioning, allocated GPUs, and paths on execution latency and throughput. Subsequently, to comprehensively consider resource allocation and model partitioning, we define an optimization problem to minimize the resource allocation costs while satisfying the service performance requirements. By comparing with other model partitioning and allocation methods, we demonstrate that RAMPA can run additional DL services while satisfying performance requirements. Furthermore, we compare the execution performances of the services allocated by RAMPA with those allocated by the conventional method. Finally, we investigate whether RAMPA could allocate DL services within a practical computation time.

The main contributions of this study are as follows:

- We formulate the impact of model partitioning, allocated GPUs, and paths on execution latency and throughput.
- We define an optimization problem to execute more DL services simultaneously.
- We demonstrate that RAMPA can run more DL services while satisfying the performance requirements.

The remainder of this paper is organized as follows: Section II discusses the related work. Section III provides an overview of clusters with heterogeneous GPUs. Section IV provides an overview of RAMPA. Section V validates whether more DL services can be executed by RAMPA and discusses DL service execution performance and computational time. Finally, Section VI concludes the paper.

II. Related Work

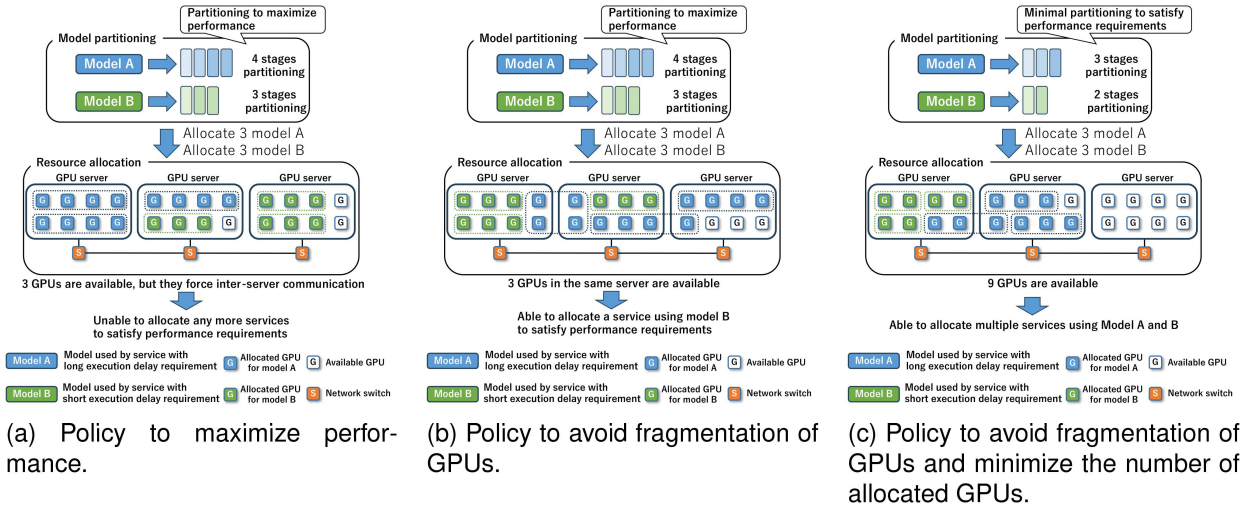
In GPU clusters, pipeline parallelism is used when the memory and computing capacity of the GPUs are insufficient for the size and computational complexity of the DL model. In pipeline parallelism, a DL model is partitioned into multiple pipeline stages, and each pipeline stage is allocated to a GPU. Larger-size DL models can be executed by the cooperation of multiple GPUs. Each pipeline stage is processed in parallel. The throughput increases because the amount of data that can be processed simultaneously increases. However, the execution latency and throughput change based on the process of the DL model that is executed by the GPU at each pipeline stage. An appropriate model partitioning and allocation method is required to exploit pipeline parallelism.

Several model partitioning and allocation methods have been proposed [8], [9], [10]. Huang et al. proposed a pipeline parallelism method, GPipe, to achieve the fast training of large models [8]. Gpipe maximizes the efficiency of the pipeline parallelism by minimizing the variance in the estimated computational cost of each pipeline stage. Narayanan et al. proposed a pipeline parallelism method, PipeDream, to minimize the large model training time [10]. PipeDream partitions the DL model to minimize the maximum execution latency for each pipeline stage by estimating the execution latency of each pipeline stage and the communication time between the stages based on the DL model. Zhuohan et al. proposed a model partitioning method, AlpaServe, to execute the maximum number of DL services to satisfy the execution latency requirements of a requested set of services [9]. AlpaServe partitions DL models to minimize the maximum execution latency for each pipeline stage and selects a combination that maximizes the number of services that satisfy the performance requirements of services. These methods achieve a high-performance DL model execution. However, they do not consider the impact of the performance of allocated resources on service execution performance because they are targeted for execution on homogeneous architectures. Therefore, most of the reported methods cannot achieve proper model partitioning and resource allocation to satisfy the performance requirements of clusters with heterogeneous GPUs.

A method that considers the impact of the performance of allocated resources on the service execution performance has been proposed [11]. Hu et al. proposed a pipeline parallelism method, PipeEdge, for fast inference of large DL models in a heterogeneous device-connected environment [11]. Their method maximizes the throughput by model partitioning and resource allocation to minimize the maximum execution latency for each pipeline stage, considering the resource performance and communication delays.

Conventional methods aim to maximize DL service performance requested at a given time and do not consider the resources allocated to future services. Consequently, the number of executable DL services is limited. We previously proposed a resource allocation method to execute more services simultaneously by considering future service execution [12]. However, the method does not target DL service execution using pipeline parallelism in GPU clusters. Thus, a comprehensive consideration of resource allocation and model partitioning to execute many DL services in a GPU cluster is required.

Fig. 1 shows three examples of DL model partitioning and allocation in a GPU cluster comprising three GPU servers with eight GPUs. In each example, three services with long



(a) Policy to maximize performance.

(b) Policy to avoid fragmentation of GPUs.

(c) Policy to avoid fragmentation of GPUs and minimize the number of allocated GPUs.

Fig. 1. Example of DL model partitioning and allocation.

execution latency requirements using model_A and three services with short execution latency requirements using model_B are allocated. In Fig. 1(a), a DL model is partitioned to maximize throughput and allocate GPUs within the same GPU server to minimize execution latency. Consequently, the GPUs within each server are fragmented, forcing GPUs for newly allocated services to communicate with longer delays. In Fig. 1(b), the allocation of GPUs in servers with numerous available resources is avoided. Under this policy, GPUs on different servers can be used for services that use model_A, which accepts longer delays. Consequently, GPUs are not fragmented, and some GPUs are available for running services with shorter execution latency requirements. In Fig. 1(c), in addition to the resource allocation policy in Fig. 1(b), the number of model partitions should be as low as possible to reduce allocated GPUs. In this policy, the performance is not maximized; however, the performance requirements can be satisfied. Consequently, all GPUs on one server are available. Therefore, additional services can be provided. Accordingly, model partitioning and resource allocation must be performed considering the resources to be used for services and the resource allocation situation to execute more DL services.

III. Cluster With Heterogeneous GPUs

A. Overview of Cluster With Heterogeneous GPUs

In this study, it is assumed that DL services are executed in a GPU cluster with multiple types of GPUs. This cluster comprises multiple racks equipped with multiple GPU servers. GPUs in each rack are aggregated using a network switch. Network switches are connected to each other to form an inter-rack network. For the flexible cooperation of multiple GPUs, we assume that any GPUs in the cluster, regardless of the rack or server, can collaborate to process by abstracting the GPUs. This abstraction method was proposed by Jin et al. [13].

Fig. 2 shows the DL services execution process. We assume that service deployment requests are sent to a cluster at any time. Once a DL service deployment request is sent to the

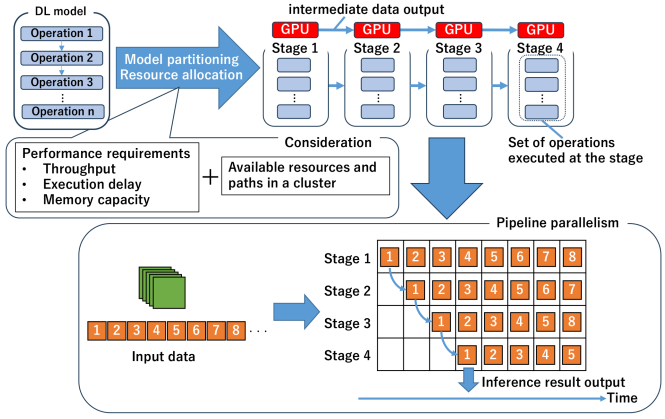


Fig. 2. Overview of service processing.

cluster, the corresponding DL model is partitioned into multiple pipeline stages. Subsequently, the execution GPU for each pipeline stage and the communication paths between GPUs are allocated. Following resource allocation, the allocated GPUs load the corresponding model partition into the memory of the GPU from the storage devices in the GPU cluster. Once these processes are completed, the DL service is executed. In this study, it is assumed DL services perform inference processing on the input data. Following deployment, the data for inference are sent to the deployed service. Note that each GPU does not communicate with any storage device during execution. Therefore, communication with a storage device does not affect the service execution performance.

B. Information Considered for Model Partitioning and Allocation

The notations for the information considered for DL model partitioning and allocation are listed in Table I.

1) GPU Cluster Information: We represent the sets of available GPUs, network links, and switches as G, L, and S, respectively. For each GPU $g \in G$, we define the floating-point

TABLE I
NOTATION OF THE GPU CLUSTER AND SERVICE EXECUTION REQUEST

Symbols	Definition
Notation of GPU cluster	
G	Set of available GPUs
L	Set of network links
S	Set of switches
f_g	Performance metric (FLOPS) of GPU $g \in G$
m_g	Memory capacity of GPU $g \in G$
R	Set of available paths between GPUs
b_l	Bandwidth of network link $l \in L$
t_e^p	Bandwidth of propagation delay $e \in L$
t_s^s	Switching delay of switch $s \in S$
Notation of DL service execution information	
K	Set of execution service
γ_k	Throughput requirement of service $k \in K$
δ_k	Acceptable time of service $k \in K$
a_k	DL model used for service $k \in K$
V_a	Set of operations of DL model a
E_a	Set of edges indicating the correspondence of operations in DL model a
$R_{a,v,v'}^v$	Set of paths between operations $v, v' \in V_a$ in DL model a
w_v	Memory consumption for operation $v \in V_a$
$t_{v,g}^g$	Computation time for operation $v \in V_a$ on GPU $g \in G$
d_e	Output data size between operations corresponding to edge $e \in E_a$
v_e^s	Source node in operation graph edge e
v_e^t	Target node in operation graph edge e
Notation of mapping of services to GPU clusters	
$\chi(v,g)$	Mapping of operation v to GPU g
$\upsilon(e,r)$	Mapping of edge indicating the correspondence of operations e to path r

operations per second (FLOPS) f_g and GPU memory capacity m_g. We define the set of paths that can be established between any GPU pair as R. Each path is a subset of the set L of network links. For each network link $l \in L$, we define the bandwidth b_l and the propagation delay t_l^p. For each switch $s \in S$, we define the switching processing time t_s^s.

2) DL Service Execution Information: In this study, we represent a set of execution services as K. For each service, we denote a_k using the DL model used for each service $k \in K$. As the execution information of a service $k \in K$, operation graph $G(V_{a_k}, E_{a_k})$ representing the relationship between the operations required to execute DL model a_k, throughput requirement γ_k, and execution latency requirement δ_k is provided. An operation corresponds to the layer of the DL model and can be a single pipeline stage in pipeline parallelism. In this study, we only target inferences using the DL model in the DL service task. Therefore, the operation graph is constructed without considering training processes such as backpropagation. Model partitioning and allocation for training is future work.

In the operation graph of DL model a, each node $v \in V_a$ corresponds to the operation, and each edge $e \in E_a$ represents the relationships between operations. For each operation $v \in V_a$, we define the amount of memory consumed to execute the operation as w_v. In addition, we define $t_{v,g}^g$ as the execution latency for executing operation $v \in V_a$ on GPU $g \in G$. For each operation graph edge $e \in E_a$, the intermediate data size transferred between the corresponding operations is defined as d_e. These are set by prior profiles. Furthermore, for the operation graph edge e, we define the source node v_e^s and target node v_e^t. We define the set of paths between operations $v, v' \in V_a$ as $R_{a,v,v'}^v$. This is a subset of the set of operation graph edges.

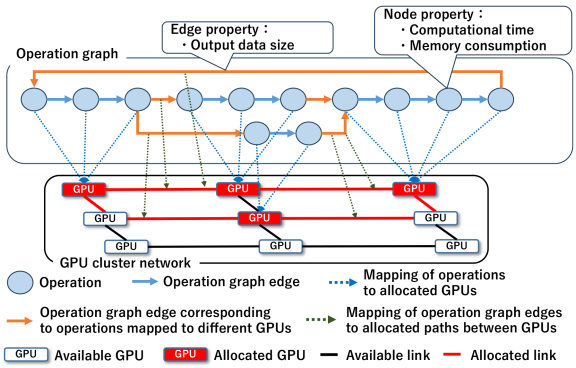


Fig. 3. Example of mapping of the operation graph to GPU cluster.

C. Mapping the Service Execution Request

To represent DL model partitioning and allocation, we map the nodes and edges of the operation graph to the GPUs and paths in the cluster, respectively. Fig. 3 shows the mapping of the operation graph and the GPU cluster when the DL model is partitioned by four pipeline stages. Each GPU executes all the operations mapped to the GPU. The number of GPUs to which the operation graph nodes are mapped corresponds to the number of pipeline stages. In addition, communication occurs between GPUs to send data to the subsequent pipeline stage. To determine the path in this communication, the mapping between an operation graph edge and a path between GPUs to which the operations are mapped.

$\chi(v,g)$ denotes the mapping between an operation and a GPU. When an operation $v \in V_a$ of model a is executed on GPU $g \in G$, $\chi(v,g) = 1$ and $\chi(v,g) = 0$ otherwise.

$\upsilon(e,r)$ denotes the mapping between an operation graph edge and the path between GPUs. When an operation graph edge $e \in E_a$ of model a is mapped to a path $r \in R_{g^1,g^2}$ between GPU pairs $g^1, g^2 \in G$, $\upsilon(e,r) = 1$ and $\upsilon(e,r) = 0$ otherwise.

IV. RESOURCE AWARE MODEL PARTITIONING AND ALLOCATION

In this study, we propose RAMPA to run several DL services while satisfying performance requirements. RAMPA determines (1) the number of pipeline stages, (2) GPUs executing each pipeline stage and the path between GPUs, and (3) operations corresponding to each pipeline stage. First, we formulate the impact of model partitioning, allocated GPUs, and paths on execution latency and throughput. Subsequently, we define the allocation costs for GPUs and network links in the cluster to avoid the allocation of resources required for future requested services. Finally, we define an optimization problem to determine the model partitioning and allocation that can minimize the allocation cost while satisfying the performance requirements to execute numerous DL services simultaneously.

A. Impact of Model Partitioning, Allocated GPUs, and Paths on Execution Latency and Throughput

We formulate the impact of model partitioning and resource allocation on throughput and execution latency. In this study, it

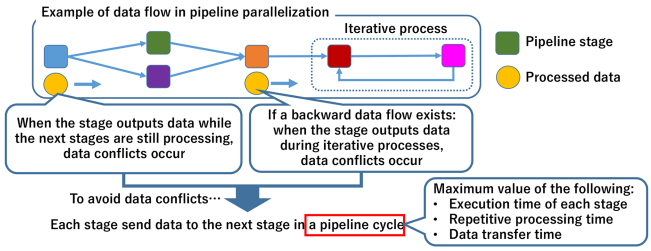


Fig. 4. Overview of pipeline cycle.

is assumed that data in a pipeline stage are output to the next pipeline stage in such a manner that no data conflicts occur. Therefore, data are sent to the next stage in a cycle that does not cause data conflicts. This cycle is referred to as the pipeline cycle, and we estimate performance based on the pipeline cycle. An overview of the pipeline cycle is shown in Fig. 4.

1) Throughput of Service: As the pipeline stages can be processed in parallel, the throughput is the inverse of the pipeline cycle. Throughput P_k of service $k \in K$ is obtained as follows:

$$P_k = \frac{1}{T_k^u} \tag{1}$$

where T_k^u denotes the pipeline cycle of service k.

2) Execution Latency of Service: The execution latency of the service is the time from the data input to the first pipeline stage until the completion of execution in the last pipeline stage. As data are sent to the next stage of every pipeline cycle, the execution latency of the service is the sum of the pipeline cycle and communication delay between the pipeline stages in the flow of input data. In pipeline parallelism, execution latency depends on the processing of the pipeline stage with the highest latency. Therefore, the execution latency of a service is the maximum value of the execution latency when the input data passes through the corresponding pipeline stage in each path of the operation graph. Execution latency T_k^r of service $k \in K$ is obtained as follows:

$$T_k^r = \max_{y \in P_{v_k^f, v_k^e}} \left\{ T_k^u + \sum_{e \in y} \sum_{r \in R} \upsilon(e, r) \left(T_k^u + T_{a_k, e}^c \right) \right\} \tag{2}$$

where T_k^u denotes the pipeline cycle of service k and $T_{a_k,e}^c$ denotes the communication delay in the path mapped to the operation graph edge e. v_k^f and v_k^e represent operations that receive data first and output data last, respectively.

3) Pipeline Cycle: In pipeline parallelism, the input data from the previous stage are processed until a GPU is available. In addition, communication between pipeline stages and pipeline stage processing can overlap [11]. Therefore, to complete processing without data conflicts in pipeline parallelism, where data transition is only forward, the pipeline cycle is the maximum execution time of each pipeline stage and the communication delay between stages. However, if the operation graph has backward edges and the operations are repeated, the data sent from a later stage to the previous stage must also be considered. In this case, the pipeline cycle is the maximum execution latency of the set of pipeline stages that are to be repeated, in addition to the time mentioned above. Therefore, pipeline cycle T_k^u for service $k \in K$ is obtained as follows:

$$T_k^u = \max_{e \in E} \sum_{r \in R} \upsilon(e, r) \cdot \begin{cases} \max \left(T_{a_k, v_e^s}^e, T_{a_k, e}^c \right) & e\, is\, forward \\ \max \left(T_{a_k, v_e^t, v_e^s}^b, T_{a_k, e}^c \right) & e\, is\, backward \end{cases} \tag{3}$$

where $T_{a_k, v_e^s}^e$ denotes the execution latency in a pipeline stage corresponding to operation v_e^s. It is the sum of the execution latencies of all operations mapped by the GPU. $T_{a_k,e}^c$ denotes the communication delay in the path mapped to the operation graph edge e. It is the sum of the time required to obtain the head of the intermediate output data and the transmission delay. $T_{a_k, v_e^t, v_e^s}^b$ denotes the execution latency from the pipeline stage corresponding to operation v_e^t to the pipeline stage corresponding to operation v_e^s. It is the sum of the execution latency in the pipeline stages and the communication delay between the pipeline stages. Mathematical details are given in supplementary material.

B. Defining the Optimization Problem

We aim to execute several services simultaneously in a cluster with heterogeneous GPUs. To achieve this objective, we avoid allocating important GPUs and network links that may be required for future service requests. This policy is similar to that proposed in our previous study [12]. However, the method does not target DL service execution using pipeline parallelism in GPU clusters. In this study, we define an optimization problem for model partitioning and resource allocation to execute numerous services simultaneously.

1) Allocation Cost: We define the allocation costs for GPUs and network links in the GPU cluster. The concept of allocation cost aims to minimize the allocation of GPUs and network links that are important for executing services while satisfying performance requirements. Fig. 5 shows examples of inefficient GPU and path allocation. Inefficient GPU allocation in Fig. 5 leads to the rapid depletion of high-performance GPUs, thereby limiting the number of services that can be executed. Inefficient path allocation in Fig. 5 increases communication latency between GPUs and leads to increased routing constraints. To prevent such problems in this study, higher costs are imposed on critical resources necessary for executing services that satisfy performance requirements.

a) GPU allocation cost: GPUs with higher computational and memory capacities are more capable of satisfying performance requirements. Furthermore, GPUs in the racks with more available GPUs have more GPUs in close proximity. This implies that low-latency communication between pipelines is probable. Therefore, GPUs with high computing and memory capacities and several available GPUs in the corresponding rack are important. We define GPU allocation cost as the product of these factors. GPU allocation cost C_g^g for GPU $g \in G$ is obtained as follows:

$$C_g^g = f_g \cdot m_g \cdot q_g, \tag{4}$$

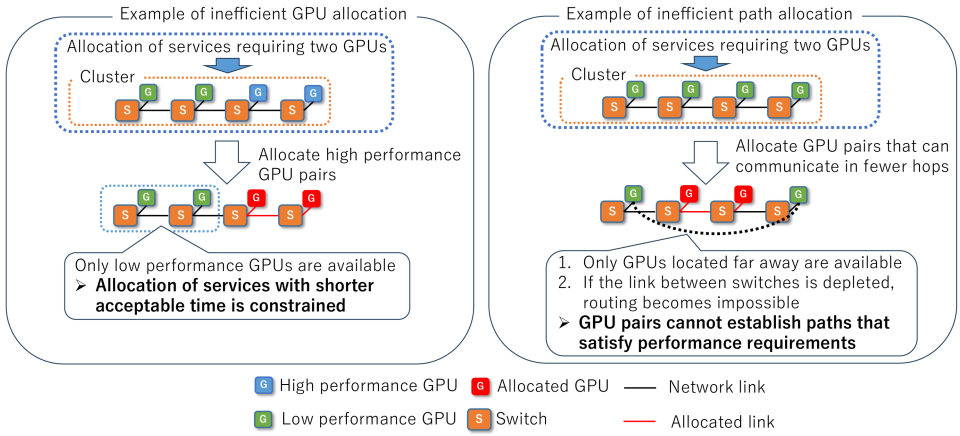


Fig. 5. Example of inefficient GPU and path allocation.

where q_g denotes the number of available GPUs in the rack with GPU g.

b) Network link allocation cost: Network links used as paths between important GPUs are essential. Furthermore, the path length should be short for low-latency communication. Therefore, the network links, which may be the shortest paths between important GPU pairs, are essential. This policy is similar to that in our previously proposed resource allocation method [12]. We set the network link allocation cost using the same policy as that used in this study.

The potential of network link $l \in L$ to be on the shortest path for the GPU pair $g^1, g^2 \in G$ is the proportion of the number of shortest paths through link l to the number of shortest paths for GPU pair g^1, g^2. The importance of a GPU pair is the sum of the costs of the two GPUs divided by the shortest number of hops between them. The network link allocation cost C_l^l for network link $l \in L$ is obtained as follows:

$$C_l^l = \sum_{g^1,g^2 \in G} \left\{ \left(\frac{N^r_{g^1,g^2}(l)}{N^r_{g^1,g^2}} \right) \cdot \left(\frac{C_{g^1} + C_{g^2}}{H_{g^1,g^2}} \right) \right\} \tag{5}$$

where $N^r_{g^1,g^2}(l)$ denotes the number of shortest paths through link l and $N^r_{g^1,g^2}$ denotes the number of shortest paths for GPU pair g^1, g^2. H_{g^1,g^2} denotes the shortest hop between GPU pair $g^1, g^2 \in G$.

2) Optimization Problem: We define an optimization problem that outputs mappings between operations and executing GPUs and between operation graph edges and paths based on information on the GPU cluster and DL service execution. Solving this optimization problem minimizes the allocation of important GPUs and paths for service execution while satisfying the performance requirements.

3) Objective: The objective is to minimize the sum of the allocation costs of GPUs and network links allocated to the DL service that is,

$$\begin{aligned} minimize \sum_{g \in G} 1_{\sum_{v \in V_{a_k}} \chi(v,g) > 0} C_g^g + \\ \sum_{r \in R} \sum_{e \in E_{a_k}} \upsilon(e,r) \sum_{l \in r} C_l^l \end{aligned} \tag{6}$$

where $1_{\sum_{v \in V_{a_k}} \chi(v,g) > 0}$ is one when $\sum_{v \in V_{a_k}} \chi(v,g) > 0$ and zero otherwise.

4) Constraints: We define four constants. Details of these formulations are shown in the supplemental material.

a) Mapping constraint: These operations must be mapped to an available GPU.

b) Throughput requirement: The throughput of the allocated DL service must be larger than the throughput requirement of the service.

c) Execution latency requirement: The execution latency of the allocated DL service must be smaller than the execution latency requirement of the service.

d) Memory requirement: The total memory consumption of the operations mapped to the GPU and the data size input to the pipeline stage corresponding to that GPU must be less than or equal to the GPU memory capacity.

C. Deriving Solutions to Optimization Problem

To derive the optimization problem defined in Section IV-B, we searched for mappings between the operation graph nodes and GPU, operation graph edges, and paths. However, such mappings are a binomial combinational optimization problem, and resource allocation based on the binomial combinational optimization problem is NP-hard [14]. Metaheuristic methods have been used to address such problems. In this study, we solve this problem using ant colony optimization (ACO).

ACO is a population-based metaheuristic method in which multiple agents probabilistically search for solutions. ACO is flexible and can adapt to changes in the environment [15] to flexibly search for solutions even if the resource utilization status in a cluster changes. In ACO, the pheromone values are first assigned to GPUs and network links. The higher the pheromone values of the GPU and network link, the more likely they are to be selected by the agent. Once multiple agents probabilistically search for a solution based on pheromones, an optimal solution is selected from the searched solutions. Finally, the pheromone value in the optimal solution is increased. This process is repeated several times.

TABLE II
NOTATION OF GPU AND PATH ALLOCATION BASED ON ACO

Symbols	Definition
τ_r	Pheromone of GPU or link r
α	Pheromone weight
β	Resource allocation cost weight
ρ	Pheromone decrease rate
ϕ	Pheromone increase rate
G^b	Set of GPUs in current best solution
L^b	Set of links in current best solution

VNE-AC was proposed for resource mapping using ACO [14]. However, the method allocates only the shortest routing paths. In this study, because network link allocation costs are not directly related to communication delays, the performance requirements may not be satisfied because of communication delays between GPUs if a path is allocated based on the shortest path problem. We arranged and used VNE-AC to use ACO to select network links. However, any method can be used as long as the solution can be derived.

In deriving the solution using ACO, we changed the number of pipeline stages from one to the number of operations and determined the lowest cost solution for each number of pipeline stages. Thereafter, we derived an optimal solution by selecting the lowest cost among these solutions. To search for a solution, the following steps were performed: (1) GPU search, (2) network link search, (3) performance requirement check, and (4) pheromone update. If the allocation cost exceeds the current minimum allocation cost, then the process is rejected to avoid unnecessary processes. The notations used for ACO are listed in Table II.

1) GPU Search: During the GPU search, the agent probabilistically selects the GPU corresponding to each pipeline stage from the available GPUs. The objective is to minimize the allocation cost; therefore, the allocation probability of GPUs with a low cost is set high. We define GPU $g \in G$ allocation probability p_g^g as follows:

$$p_g^g = \frac{(\tau_g)^\alpha \left(\frac{1}{(C_g^g)^\beta}\right)}{\sum_{x \in G}\left[(\tau_x)^\alpha \frac{1}{(C_x^g)^\beta}\right]},$$

2) Network Link Search: In a network link search, the agent generates sub-agents to explore the paths between the GPUs selected in the GPU search. Each sub-agent probabilistically selects a network link from the source GPU. Subsequently, the sub-agent probabilistically selects the next network link from the destination node of the first link. This process is repeated until the destination GPU is reached. We define network link $l \in L$ and allocation probability $p_{l,n}^l$ as follows:

$$p_{l,n}^l = \frac{(\tau_l)^\alpha \frac{1}{(C_l^l)^\beta}}{\sum_{x \in L}\left[(\tau_x)^\alpha \frac{1}{(C_x^l)^\beta}\right]}$$

3) Performance Requirement Check: In this phase, we check whether the performance requirements are satisfied when the DL service is executed by the selected GPUs and paths. First, we calculate the throughput, execution latency, and memory consumption for each combination of operations executed at each pipeline stage. Thereafter, we check whether a combination exists that satisfies the performance requirements. If no combination satisfies the performance requirements, the process is rejected.

4) Pheromone Update: Following the performance requirement check, pheromones of all GPUs and network links decay based on the pheromone reduction rate rho. However, only the pheromones of the GPU and network link in the optimal solution for each iteration are augmented based on the pheromone increase rate phi and the allocation cost value. The pheromone enhancement value h is obtained as follows:

$$h = \frac{\phi}{\sum_{g \in G^b} C_g^g + \sum_{l \in L^b} C_l^l}$$

The pheromones τ_g, τ_l for GPU $g \in G$ and network link $l \in L$ are updated as follows:

$$\tau_g = \rho\tau_g + h, \ \tau_l = \rho\tau_l + h,$$

V. EVALUATION

We evaluate RAMPA through simulations of a cluster with heterogeneous GPUs and DL service allocation. Other researchers can access and extend our evaluation using our simulator published below: https://github.com/a-ikoma/RAMPA_simulation.

A. Settings

We describe the cluster with heterogeneous GPUs, execution services, and comparative methods used to evaluate RAMPA.

1) Cluster With Heterogeneous GPUs: We assume a GPU cluster comprising multiple racks. Each rack contains five GPU servers with eight GPUs. We evaluate RAMPA in the following four GPU clusters with different GPU and network performances:

- Base cluster: Neutral GPU cluster for comparison.
- High-bandwidth cluster: GPU cluster with high network bandwidth.
- High-performance cluster: GPU cluster with many high-performance GPUs.
- Large cluster: GPU cluster with many GPUs.

A high-bandwidth cluster differs from the base cluster only in terms of the bandwidth of each network link. In a high-performance cluster, relatively low-performance GPUs are removed from the base cluster. In all other aspects, the base and high-performance clusters were identical. In a large cluster, relatively more GPUs are connected than in the base, high-bandwidth, and high-performance clusters. In all other aspects, the base and large clusters were identical. Fig. 6 shows the base and high-bandwidth clusters, high-performance cluster, and large cluster. For stable and fast communication between GPUs, the same type of GPUs in each rack are aggregated using an optical circuit switch. The optical circuit switches are connected to each other by an optical fiber to form an inter-rack network. We assume that the network topology is a two-dimensional torus topology of 4×4 with 16 optical circuit switches or 6×6 with 36 optical circuit switches. For flexible routing, an optical circuit

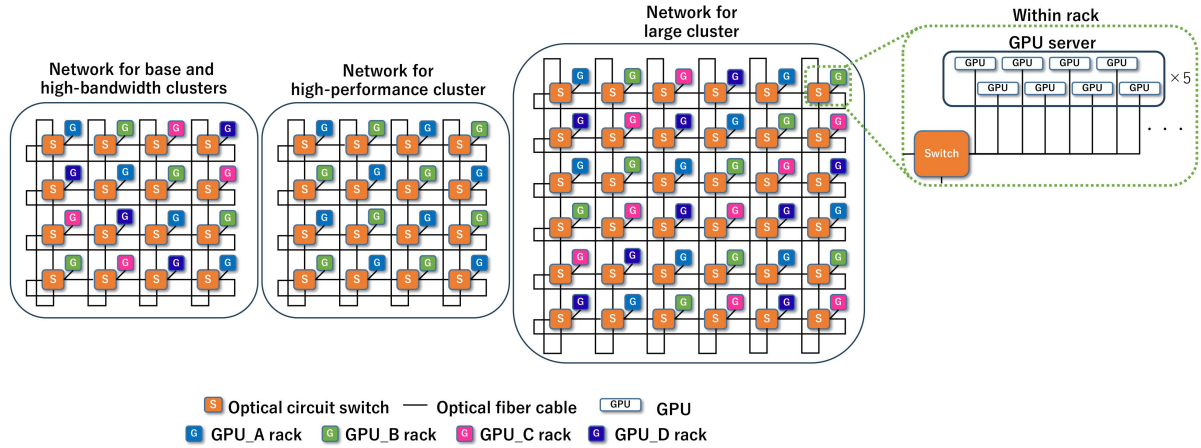


Fig. 6. Evaluation networks.

TABLE III
PARAMETER SETTINGS FOR THE GPU CLUSTER

Parameters	Value
GPU_A FLOPS	30.3 TFLOPS
GPU_A memory size	24 GB
GPU_A memory bandwidth	300 GB/s
GPU_B FLOPS	14 TFLOPS
GPU_B memory size	32 GB
GPU_B memory bandwidth	900 GB/s
GPU_C FLOPS	10.3 TFLOPS
GPU_C memory size	24 GB
GPU_C memory bandwidth	933 GB/s
GPU_D FLOPS	8.1 TFLOPS
GPU_D memory size	16 GB
GPU_D memory bandwidth	320 GB/s
Propagation delay (Switch - GPU)	0.05 μs
Propagation delay (Switch - Switch)	0.1 μs
Switch latency of the optical circuit switch	30 ns
The bandwidth (Base / high bandwidth)	10 Gbps/100 Gbps

TABLE IV
GENERATION PROBABILITY OF DEPLOYMENT REQUESTS FOR EACH SERVICE CATEGORY IN EACH ENVIRONMENT

Service	High throughput	Low delay	Huge model
Same demand for all services	Same probability		
High demand for high-throughput service	0.8	0.1	0.1
High demand for low-delay service	0.1	0.8	0.1
High demand for huge model service	0.1	0.1	0.8

TABLE V
DL MODEL AND PERFORMANCE REQUIREMENTS FOR EACH SERVICE

Service	Execution latency (s)	Throughput (tps)
Service 1	0.1	60
Service 2	0.2	30
Service 3	0.05	30
Service 4	0.8	5
Service 5	10	0.1
Service 6	5	0.25

switch pair is connected to four optical fibers. Connected GPUs were of the following four types: NVIDIA L4 [16], NVIDIA V100 [17], NVIDIA A30 [18], and NVIDIA Tesla T4 [19]. Hereinafter, we refer to these as GPU_A, GPU_B, GPU_C, and GPU_D, respectively. In the high-performance cluster, eight racks with only GPU_A and GPU_B are connected. GPU_A and GPU_B performed better than GPU_C and GPU_D in terms of memory capacity and FLOPS.

The parameters of the GPU clusters used to estimate the performance and set the allocation costs are listed in Table III. The bandwidth in the base and high-performance clusters is 10 Gbps, and that in the high-bandwidth cluster is 100 Gbps. The network link length within a rack is 10 m, and that between racks is 20 m. The propagation delay of each network link is 0.05 μs and 0.1 μs. By referencing CALIENT's optical circuit switch [20], we set the switching delay to 0.03 μs. GPU FLOPS, memory bandwidth, and memory capacity are based on the data sheet of each GPU.

2) Execution Service: We assume the following DL services with different characteristics in throughputs, execution latencies, and memory requirements:

- Service 1 (High-throughput): Object recognition service for video streaming using YOLOS [21].
- Service 2 (High-throughput): Video classification service for video data using VideoMAE [22].
- Service 3 (Low-delay): Image classification service using vision transformer [21].
- Service 4 (Low-delay): Fast test generation service using Gemma2-2b [21].
- Service 5 (Huge model): AI chat service using Gemma2-27b [21].
- Service 6 (Huge model): AI chat service using Qwen2.5-32B [23].

These services fall into three categories, High-throughput, Low-delay, and Huge model, based on the severity of the performance requirements for the computation complexity. DL models used in this study were downloaded from Hugging Face. The throughput, execution latency requirements, and usage DL model for each service are listed in Table V. Throughput refers to the number of transactions per second (tps), where one inference of an input datum is a transaction. The execution latency is defined as the time required to complete a transaction. In service 1 and 2, one input data sample is set to one frame and one video data, respectively. In service 3 and 4, one input data sample is set to one image and 1000 characters of text, respectively. In service

TABLE VI
PARAMETER SETTINGS FOR EACH DL MODEL

Operation	FLOPs	memory consumption	output data size
yolos-base [21]			
Embedings	3.9 G	192.55 MB	109.60 MB
YolosLayer (× 12)	48.17 G	32.46 MB	9.96 MB
LayerNorm	13.06 M	0	9.96 MB
YolosPooler	1.18 M	2.7 MB	3 KB
VideoMAE [22]			
VideoMAEEmbeddings	4.93 G	7.02 MB	6.12 MB
VideoMAELayer (× 24)	39.48 G	56.3 MB	6.12 MB
LayerNorm	8.03 M	0	6.12 MB
vit-huge [25]			
ViTEmbeddings	385.68 M	4.94 MB	1.25 MB
ViTLayer (× 32)	10.11 G	90.08 MB	1.25 MB
LayerNorm	1.64 M	11.72 KB	1.25 MB
ViTPooler	3.28 M	7.51 MB	5.00 KB
gemma-2-2b [26]			
Embedding	0	2.64 GB	2.99 MB
GemmaDecoderLayer (× 26)	155.72 G	348 MB	2.99 MB
GemmaRMSNorm	0	10.5 KB	2.99 MB
Linear	1.18 T	2.63 GB	332.03 MB
gemma-2-27b [26]			
Embedding	0	5.28 GB	5.98 MB
GemmaDecoderLayer (× 46)	1.15 T	2.52 GB	5.98 MB
GemmaRMSNorm	0	20.4 KB	5.98 MB
Linear	2.36 T	5.28 GB	332.03 MB
Qwen2.5-32B [23]			
Embedding	0	1.74 GB	6.89 MB
Qwen2DecoderLayer (× 64)	256.66 G	1.08 GB	6.89 MB
Qwen2RMSNorm	0	10.8 KB	6.89 MB
lm_head	3.11 T	1.74 GB	204.77 MB

TABLE VII
PARAMETER SETTINGS FOR ACO

Parameters	Value
Number of agents	20
Number of agent generations	20
Pheromone decrease rate	0.1
Pheromone increase rate	100
Pheromone weight	2
Allocation cost weight	1
Initial pheromone value	1000

TABLE VIII
OBJECTIVE AND CONSIDERATION FOR NUMBER OF MODEL PARTITIONS OF EACH METHOD

Method	Objective	Consideration for number of model partitions
PE	Maximize throughput of requested service	No
NCAR	Minimize use of resources required for future services	No
RAMPA	Minimize use of resources required for future services	Yes

5 and 6, one input data sample is set to 1000 characters of text. We set the proportion of the number of services to be executed in the following four environments to evaluate RAMPA:

- Same demand for all services: The proportion of the number of executed services is balanced.
- High demand for high-throughput services: The proportion of the number of high-throughput services is high.
- High demand for low-delay services: The proportion of the number of low-delay services is high.
- High demand for huge model services: The proportion of the number of huge model services is high.

To simulate each environment, we generate a deployment request for each service category with a certain probability. Deployment of services within the same category is random. The probabilities of each service category are listed in Table IV.

Table VI lists the layer names corresponding to the operations of each DL model, corresponding to floating-point operations (FLOPs), amount of memory consumed, and size of the output data for each operation. The FLOPs and output data sizes are set by profiling using calflops [24]. Memory consumption is set to the parameter size of each operation multiplied by 1.2. The reason for multiplying by 1.2 is to consider the overhead of the consumed memory. In this evaluation, we set the execution latency of the operation for each GPU type based on the values shown in Section III. Execution latency is the sum of FLOPs divided by GPU FLOPS and memory consumption divided by GPU memory bandwidth. Therefore, execution latency $t^g_{v,g}$ of operation v in GPU g can be obtained as follows:

$$t^g_{v,g} = \frac{FLOPs(v)}{f_g} + \frac{d_v}{Memory\_Band(g)}$$

where $FLOPs(v)$ denotes the FLOPS of operation v and $Memory\_Band(g)$ refers to the memory bandwidth of GPU g. However, more appropriate execution latency estimation methods may exist, which will be a topic for future studies.

3) Parameter Settings for ACO: We use ACO for GPU and path allocation. Parameters for ACO are listed in Table VII.

4) Comparative Methods: RAMPA can optimize model partitioning and allocation by considering both the importance of allocated resources and number of allocated resources. To demonstrate their effectiveness, we compared them using the following two methods. Main differences between two comparative methods and RAMPA exist in terms of objectives and consideration for number of model partitions. The differences are presented in Table VIII. We describe these methods based on Table VIII.

a) PipeEdge (PE): PipeEdge [11] is a model partitioning and allocation method to maximize throughput by considering the performance of allocated resources. This method minimizes the maximum execution latency of each pipeline stage by preferentially selecting high-performance GPUs and low-latency paths. The number of pipeline stages is fixed for each model in advance. PipeEdge only considers the performance of the requested service. Resources required to execute future services may be depleted.

b) No considering allocated resource number(NCAR): NCAR is a model partitioning and allocation method modified from PE in terms of the objective to efficiently use resources. This method minimizes the use of GPUs and links required for future services to preserve the resources required for future services. However, in this method, the number of model partitions is fixed for each model in advance. It may unnecessarily allocate a large number of resources, which may constrain the number of services executed.

NCAR differs from EP only in terms of their objectives, and RAMPA and NCAR have the same objective. We demonstrate the effectiveness of the objective setting of RAMPA by comparing NCAR and EP. In addition, RAMPA and NCAR differ in terms of consideration of the number of model partitions. We demonstrate the effectiveness of model partitioning and

TABLE IX
SETTING OF THE NUMBER OF PIPELINE STAGES

Service	Ranges
Service 1	2 to 3
Service 2	2 to 4
Service 3	1 to 2
Service 4	2 to 4
Service 5	6 to 9
Service 6	4 to 5

allocation considering the number of model partitions by comparing RAMPA and NCAR.

5) Number of Pipeline Stages in Comparative Methods: In the comparison methods, the number of pipeline stages is fixed for each DL model. We evaluate the comparative methods for all possible pipeline stage number patterns to demonstrate the effectiveness of optimizing model partitioning and allocation, including the number of allocated resources. Therefore, we set the number of pipeline stages to range from the number of pipeline stages that can be executed on any GPU in this evaluation environment to the number of pipeline stages that can be executed only on a GPU with high performance (GPU_A and GPU_B). The setting of the number of pipeline stages is presented in Table IX. We evaluate all the combinations of the number of pipeline stages for the six types of services for each comparison method. Therefore, we evaluate each comparison method in 288 different pipeline stage divisions. Hereinafter, in the comparison method PE, when the number of pipeline stages for services 1 to 6 are 2, 3, 2, 4, 8, and 5, respectively, it is denoted as PE(2,3,2,4,8,5).

B. Metric

We measure the number of services that successfully allocated resources satisfying the performance requirements to evaluate the ability of RAMPA to execute more DL services. In the evaluation, we continue to generate service deployment requests until the allocation of services that satisfy performance requirements fails. The evaluation terminates when the allocation of a service that satisfies the performance requirements fails.

C. Number of Services Successfully Allocated Resources That Satisfy Performance Requirements

In Fig. 7, we show the number of services successfully allocated that satisfy the performance requirements for RAMPA and the comparative methods for all combinations of the four different GPU clusters and four different proportions of executed services. We measured 288 different combinations of pipeline stages using comparative methods. The results of the comparative methods are shown in Fig. 7; the best case, worst case, and average of all the combinations are shown. The numbers above the bars represent the corresponding number of pipeline stages.

First, we compare NCAR and PE, which differ only in terms of their objectives. NCAR tends to be capable of allocating more services. This is because it preserves the required GPUs to execute the future requested services by avoiding the allocation of resources used by other services. On the other hand, PE only considers the performance of the requested service. Resources required to execute future services may be depleted. The effectiveness of considering the resource utilization of other services to allocate more services was demonstrated. Subsequently, we compare RAMPA and NCAR. In all environments, RAMPA allocates the same or better services to satisfy the performance requirements compared to the best case of NCAR. Furthermore, the number of model partitions in the NCAR best case varies in each case. This means that the best partitioning strategy changes based on how the service arrives and its environment. To execute more services, it is necessary to consider the number of resources allocated.

To verify the model partitioning that was performed by RAMPA, we show the distribution of the number of pipeline stages of allocated services in the case of the same demand for all services in Fig. 8. As shown in Fig. 8, RAMPA partitions the DL model based on the number of pipeline stages in multiple patterns. This implies that the suitable model partitioning strategy changes based on the resource allocation situation. For the execution of numerous services, the optimization of the resources used and their number is effective based on the current resource allocation situation.

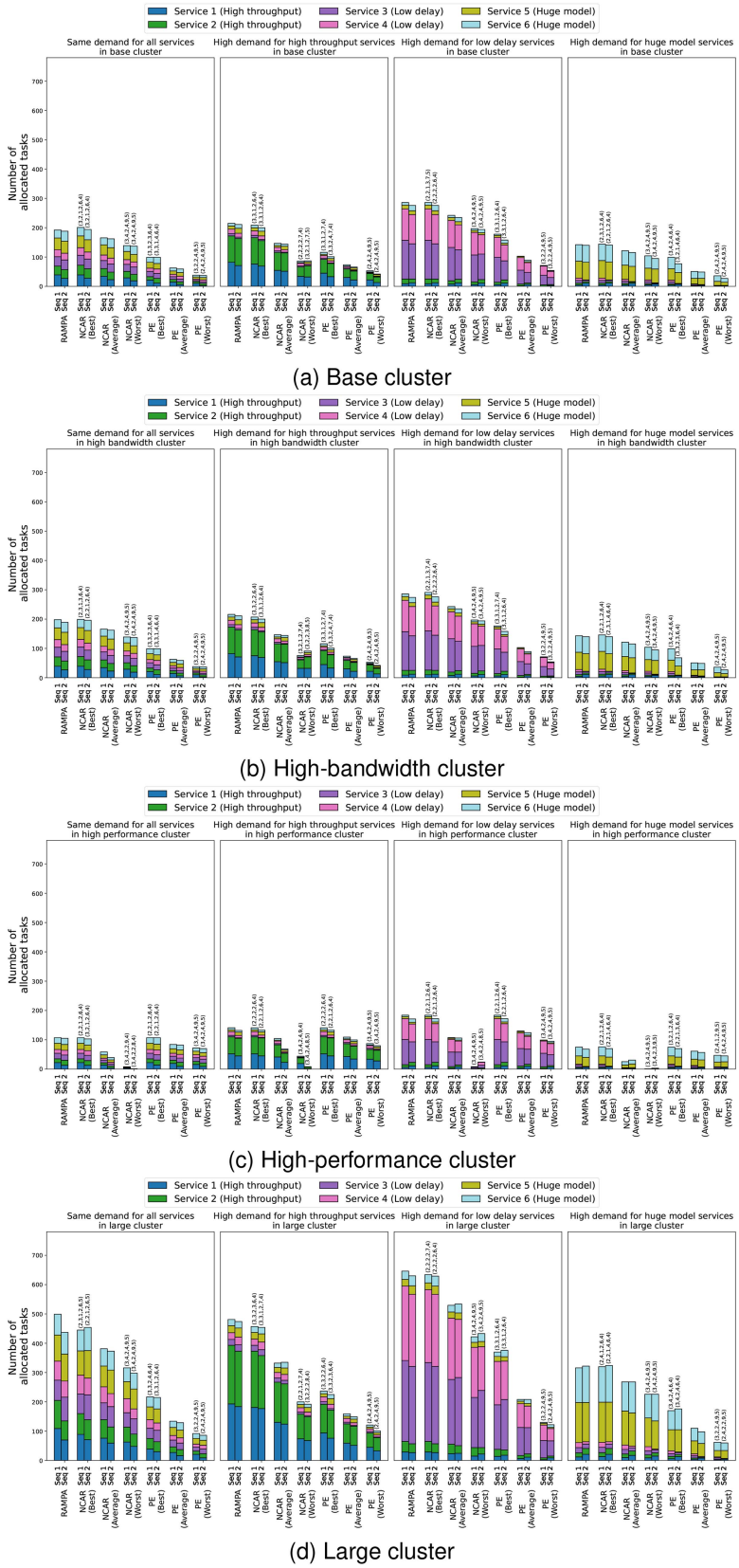
D. Comparison of Service Execution Performance

Unlike conventional methods, RAMPA does not aim to maximize performance. Therefore, it may be inferior to conventional methods in terms of service execution performance. We compare the service throughput and execution latency of RAMPA and PE and discuss the limitations of RAMPA on execution performance. Figs. 9 and 10 show the average values of throughput and execution latency for each service in the case of the same demand for all services in the base cluster. Error bars represent the maximum and minimum values. The red lines indicate the performance requirements. The orange bars in the figure represent RAMPA, and the other bars represent PE.

RAMPA had a smaller average and minimum throughput than PE for all combinations of pipeline stage numbers. As PE aims to maximize throughput, this result is similar to that of Hu et al. [11]. In execution latency, for services apart from high-throughput services, PE was superior. By contrast, in the high-throughput service, PE had a lower execution latency than RAMPA only when the number of pipeline stages was two. This is because the execution latency is affected by the communication delay between pipeline stages. If excess pipeline stages exist, then the communication delay overhead will increase. When the model was properly partitioned, RAMPA exhibited a lower service execution performance than PE. However, the performance requirements were satisfied. RAMPA is effective when performance requirements are properly set and performance maximization is not required.

E. Computational Time for Model Partitioning and Allocation by RAMPA

Unlike conventional methods, RAMPA considers resource allocation and optimizes the number of resources allocated. Owing to these processes, more computational time is required than in



(a) Base cluster

(b) High-bandwidth cluster

(c) High-performance cluster

(d) Large cluster

Fig. 7. Number of services successfully allocated resources that satisfy performance requirements.

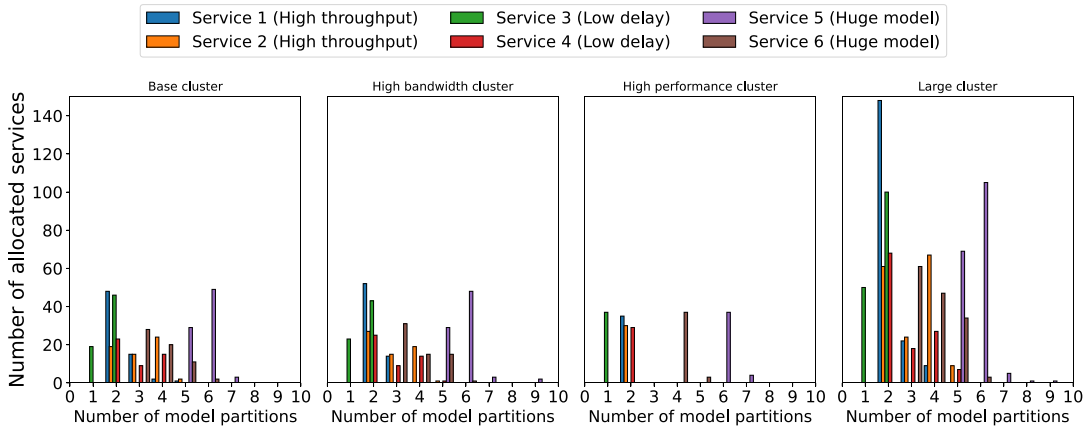
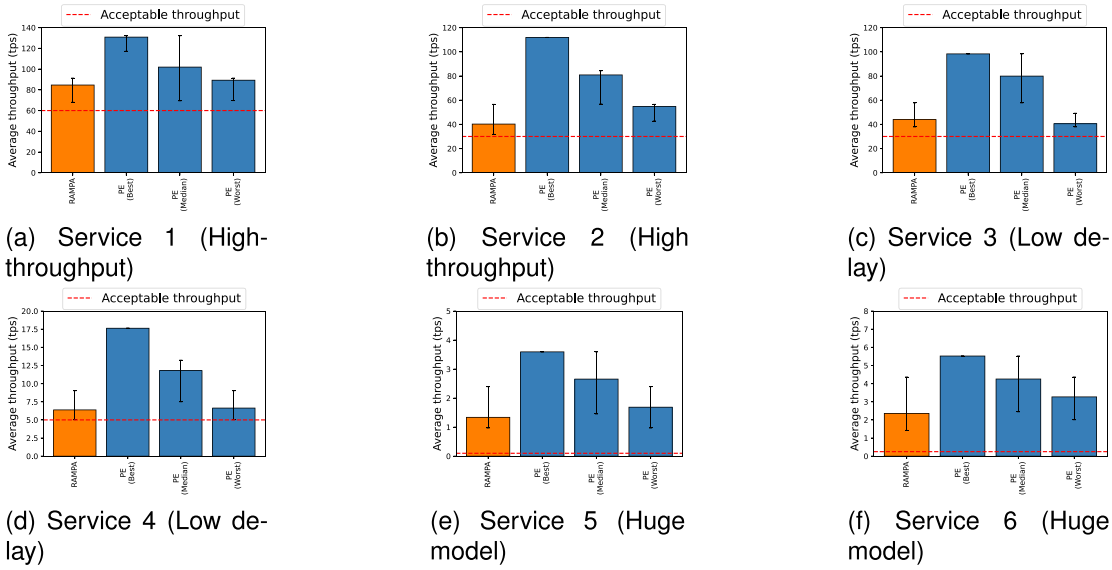


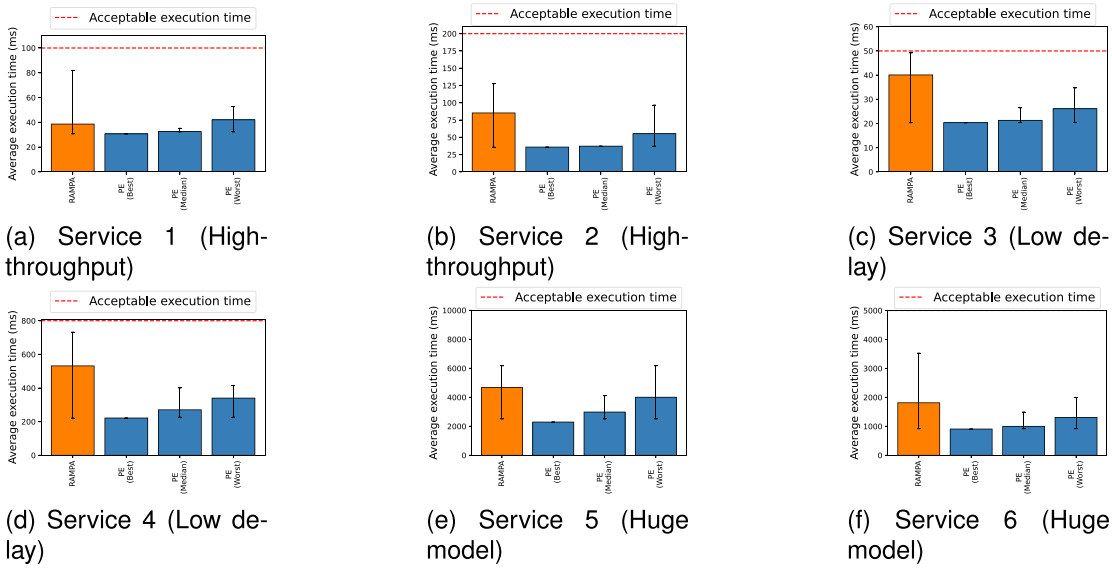
Fig. 8. Distribution of the number of pipeline stages of allocated services in RAMPA.



(a) Service 1 (High-throughput) (b) Service 2 (High throughput) (c) Service 3 (Low delay)

(d) Service 4 (Low delay) (e) Service 5 (Huge model) (f) Service 6 (Huge model)

Fig. 9. Throughput for each method.



(a) Service 1 (High-throughput) (b) Service 2 (High-throughput) (c) Service 3 (Low delay)

(d) Service 4 (Low delay) (e) Service 5 (Huge model) (f) Service 6 (Huge model)

Fig. 10. Execution latency for each method.

conventional methods. The relationship between computational complexity and computation time is a limitation of RAMPA.

1) Computational Complexity of RAMPA: We verified the computational complexity of RAMPA by clarifying the computational complexity of each of the processes described in Section IV-C.

a) GPU search: In GPU search, the GPU corresponding to each pipeline stage is selected from the available GPUs in the GPU cluster. Therefore, the GPU selection is repeated for the number of pipeline stages, and all available GPUs in the cluster are checked in each iteration. When the number of pipeline stages is N^p and the number of available GPUs is $|G|$, the computational complexity is $O(N^p|G|)$.

b) Network link search: In this phase, we probabilistically select the transit link from the source GPU to the destination GPU. This process is repeated $N^p - 1$ times because it is performed to establish the path between GPUs when the number of pipeline stages is N^p. In addition, in the worst-case scenario, a path through all the nodes is established. Therefore, the computational complexity is $O(N^p + |G|)$.

c) Performance requirement check: In this phase, we check the performance requirements for each combination of operations executed for each pipeline stage. When the number of pipeline stages is N^p and that of operations is $|V_a|$, the computational complexity is $O(|V_a|N^p)$.

d) Pheromone update: Following GPU and path selection, the pheromones are updated for all GPUs and network links. Therefore, the computational complexity is $O(|G| + |L|)$.

The maximum number of pipeline stages is equal to the number of operations in the model. Therefore, the above process is repeated from 1 to $|V_a|$ at the maximum. Thus, the computational complexity of RAMPA is $O(|V_a|^2 \cdot (|G| + |L| + |S|) + |V_a|^3)$. From this perspective, the computational complexity of RAMPA depends on the scale of the cluster and DL model.

2) Discussion on Average Computational Time: We investigate the relationship between the computational complexity and computation time of RAMPA to discuss the practicality of RAMPA. RAMPA depends on the scale of the cluster and DL model. Therefore, we measure the computational time for the base cluster and large cluster in Fig. 6. In each cluster, the relationship between the number of operations and the average computational time for each model is shown in Fig. 11. Error bars represent 95% confidence interval.

From the results, in services 5 and 6, where the number of operations exceeds 40, the computational time was greater than in the other services. However, in the comparison of services 5 and 6 and services 3 and 4, more computation time was spent on services with fewer operations. This result does not match with the computational complexity $O(|V_a|^2 \cdot (|G| + |L| + |S|) + |V_a|^3)$. This is because the number of pipeline stages required to satisfy performance requirements is significantly constrained. Once the number of pipeline stages exceeds a certain level, it becomes impossible to satisfy the execution latency requirements, and the processes are quickly terminated. Unnecessary resource exploration can be avoided by considering performance requirements.

The results also demonstrate that it takes more time for large clusters with more than 1000 GPUs in all services. This result

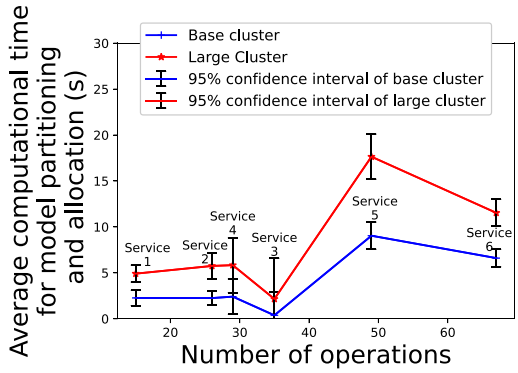


Fig. 11. Relationship between the number of operations and average computational time.

matches with the computational complexity $O(|V_a|^2 \cdot (|G| + |L| + |S|) + |V_a|^3)$. This means that the model partitioning and allocation take more time in proportion to the scale of the GPU cluster. However, in the cases evaluated in this study, the maximum is within approximately 20 s. This is considered acceptable for the time it takes to deploy the service prior to execution.

F. Discussion on the Feasibility of RAMPA

We demonstrated that many DL services can be executed simultaneously by RAMPA. RAMPA has limitations on the feasibility. RAMPA estimates the impact of resource allocation and model partitioning on performance such that services can be executed while satisfying performance requirements. In this estimation, the time required to execute an operation of that DL model on each GPU is required as a parameter. This means that for RAMPA to be effective, these parameters must be set accurately. Two approaches exist. First is pre-profiling for DL models and GPUs. If services can be identified in advance, RAMPA can be utilized by executing DL models on each GPU beforehand and leveraging their profiles. However, in cloud computing, services cannot always be predetermined. Therefore, an alternative approach involves the proposal of performance estimation methods in a GPU. Estimation methods based on GPU performance, such as FLOPS and memory bandwidth, and DL model execution information, such as FLOPs and data consumption, are considered. If an accurate performance estimation method is proposed, RAMPA will work well for any services. It is within the scope of our future work.

VI. Conclusion

We proposed RAMPA to execute additional DL services while satisfying the performance requirements in clusters of heterogeneous GPUs. RAMPA minimizes the allocation of important resources for future DL service execution to avoid inhibiting future DL service execution. We defined the resource allocation cost for GPU and network links in terms of resource importance. Furthermore, we formulated the impact of model partitioning, allocated GPUs, and paths on the execution latency and throughput. To comprehensively consider resource allocation and model partitioning, we defined an optimization problem to minimize

resource allocation costs while satisfying the service performance requirements. We evaluated the effectiveness of RAMPA by simulating the execution of DL services in a cluster of heterogeneous GPUs. The results demonstrated that more services can be executed while satisfying the performance requirements compared to the conventional method. By RAMPA, we achieved efficient GPU utilization to deliver many DL services in clusters with heterogeneous GPUs.

In the future, we plan to introduce a method for estimating the execution performance of DL services. In this study, we estimated performance using execution profiles to allocate DL services. However, obtaining execution information in advance for tasks is difficult, such as offloading inference tasks using the DL model. RAMPA is not a performance maximization method. Therefore, without accurate insights into the execution performance, allocations may degrade the performance of the service. Therefore, an estimation of the execution performance based on GPU performance, executable programs, and used DL model information is required.

References

[1] S. Ahmad, I. Shakeel, S. Mehfuz, and J. Ahmad, "Deep learning models for cloud, edge, fog, and IoT computing paradigms: Survey, recent advances, and future directions," *Comput. Sci. Rev.*, vol. 49, 2023, Art. no. 100568. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574013723000357

[2] A. Chowdhery et al., "Palm: Scaling language modeling with pathways," *J. Mach. Learn. Res.*, vol. 24, no. 1, pp. 1–113, Mar. 2024.

[3] M. Dehghani et al., "Scaling vision transformers to 22 billion parameters," in *Proc. 40th Int. Conf. Mach. Learn., Ser. Proc. Mach. Learn. Res.*, A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, Eds., 23–29 Jul. 2023, vol. 202, pp. 7480–7512. [Online]. Available: https://proceedings.mlr.press/v202/dehghani23a.html

[4] X. Zhang, "Mixtran: An efficient and fair scheduler for mixed deep learning workloads in heterogeneous GPU environments," *Cluster Comput.*, vol. 27, pp. 1–10, 2023.

[5] Z. Ye et al., "Astraea: A fair deep learning scheduler for multi-tenant GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 11, pp. 2781–2793, Nov. 2022.

[6] R. Cheng et al., "Towards GPU memory efficiency for distributed training at scale," in *Proc. ACM Symp. Cloud Comput.*, New York, NY, USA:ACM, 2023, pp. 281–297, doi: 10.1145/3620678.3624661.

[7] Z. Chen et al., "Deep learning research and development platform: Characterizing and scheduling with QoS guarantees on GPU clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 1, pp. 34–50, Jan. 2020.

[8] Y. Huang et al., "GPipe: Efficient training of giant neural networks using pipeline parallelism," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.*, Dec. 2019, pp. 103–112.

[9] Z. Li et al., "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation*, 23. Boston, MA, USA: USENIX Assoc., Jul. 2023, pp. 663–679. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[10] D. Narayanan et al., "Pipedream: Generalized pipeline parallelism for DNN training," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, New York, NY, USA:ACM, 2019, pp. 1–15, doi: 10.1145/3341301.3359646.

[11] Y. Hu et al., "PipeEdge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *Proc. 25th Euromicro Conf. Digit. Syst. Des.*, 2022, pp. 298–307.

[12] A. Ikoma, Y. Ohsita, and M. Murata, "Resource allocation considering impact of network on performance in a disaggregated data center," *IEEE Access*, vol. 12, pp. 67600–67618, 2024.

[13] X. Jin, Z. Bai, Z. Zhang, Y. Zhu, Y. Zhong, and X. Liu, "Distmind: Efficient resource disaggregation for deep learning workloads," *IEEE/ACM Trans. Netw.*, vol. 32, no. 3, pp. 2422–2437, Jun. 2024.

[14] I. Fajjari, N. Aitsaadi, G. Pujolle, and H. Zimmermann, "VNE-AC: Virtual network embedding algorithm based on ant colony metaheuristic," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2011, pp. 1–6.

[15] M. Dorigo and T. Stützle, "Ant colony optimization: Overview and recent advances," in *Handbook of Metaheuristics*. Boston, MA, USA: Springer, 2010, pp. 227–263, doi: 10.1007/978-1-4419-1665-5_8.

[16] "Nvidia l4 tensor core GPU," 2023. Accessed: Jul. 24, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/l4/

[17] "Nvidia v100 tensor core GPU," 2017. Accessed: Jul. 24, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/v100/

[18] "Nvidia a30 tensor core GPU," 2021. Accessed: Jul. 24, 2024. [Online]. Available: https://www.nvidia.com/ja-jp/data-center/products/a30-gpu/

[19] "Nvidia t4," 2018. Accessed: Jul. 24, 2024. [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla-t4/

[20] "Calient fs optical circuit switch," 2022. Accessed: Jul. 24, 2024. [Online]. Available: https://www.calient.net/wp-content/uploads/2022/06/Datasheet_Calients-Optical-Circuit-Switches.pdf

[21] Y. Fang et al., "You only look at one sequence: Rethinking transformer in vision through object detection," in *Proc. 35th Int. Conf. Neural Inf. Process. Syst.*, Dec. 2021, pp. 26183–26197.

[22] Z. Tong, Y. Song, J. Wang, and L. Wang, "Videomae: Masked autoencoders are data-efficient learners for self-supervised video pre-training," in *Proc. 36th Int. Conf. Neural Inf. Process. Syst.*, Red Hook, NY, USA: Curran Associates Inc., 2022, pp. 10078–10093.

[23] A. Yang et al., "Qwen2.5 technical report," 2024, *arXiv:2412.15115*.

[24] X. J. Ye, "calflops: A FLOPs and params calculate tool for neural networks in PyTorch framework," 2023. [Online]. Available: https://github.com/MrYxJ/calculate-flops.pytorch

[25] B. Wu et al., "Visual transformers: Token-based image representation and processing for computer vision," [cs.CV], 2020, *arXiv:2006.03677*. [Online]. Available: https://arxiv.org/abs/2006.03677

[26] G. Team, "Gemma," 2024. [Online]. Available: https://www.kaggle.com/m/3301

Akishige Ikoma received the ME and PhD degrees in information science and technology from Osaka University, Japan, in 2022 and 2025, respectively. His research focuses on network architecture for a disaggregated data center.

Yuichi Ohsita (Member, IEEE) received the ME and PhD degrees in information science and technology from Osaka University, Japan, in 2005 and 2008, respectively. From 2006 to 2012, he was an assistant professor with the Graduate School of Economics, Osaka University. In 2012, he moved to the Graduate School of Information Science and Technology, Osaka University, where he became an associate professor with the Institute for Open and Transdisciplinary Research Initiatives, in 2019. In 2023, he joined Cybermedia Center, The University of Osaka (formerly, Osaka University). He is currently an associate professor with D3 Center, The University of Osaka. His research interests include traffic engineering, traffic prediction, and network security. Dr. Ohsita is a member of IEICE and Association for Computing Machinery (ACM).

Masayuki Murata (Life Member, IEEE) received the ME and DE degrees in information and computer science from Osaka University, Japan, in 1984 and 1988, respectively. In 1984, he joined Tokyo Research Laboratory, IBM Japan, as a researcher. From 1987 to 1989, he was an assistant professor with Computation Center, Osaka University. In 1989, he joined the Department of Information and Computer Sciences, Faculty of Engineering Science, Osaka University, where he became a professor with the Graduate School of Engineering Science, in 1999. He has been with the Graduate School of Information Science and Technology, since 2004. His research interests include information network architecture, performance modeling, and evaluation. Prof. Murata is a member of ACM and IEICE.