Noname manuscript No.
(will be inserted by the editor)

A containerized task clustering for scheduling workflows to utilize processors and containers on clouds

Hidehiro Kanemitsu · Kenji Kanai · Jiro Katto · Hidenori Nakazato

Received: date / Accepted: date

Abstract Recent advancements of virtualization technologies for parallel processing involve scheduling containerized tasks in a workflow. Since a container can include multiple tasks, it can be reused or shared among applications. If every task in a workflow uses its dedicated container without sharing among any tasks, each container image must be downloaded for each task. As a result, many computational resources are required to process and the communication latency related to container image downloading can become a bottleneck for the makespan. In task scheduling algorithms for workflows, this characteristic produces a new challenging issue that how effectively shares containers among tasks to avoid redundant container image download processes and redundant task allocations. One of the fundamental problems is that no policy has been established for simultaneously satisfying effective container sharing, maintaining the degree of task parallelism, and effective computational resource utilization.

In this paper, we propose a clustering-based containerized task scheduling algorithm for clouds, namely, shareable functional task clustering for utilizing

Hidehiro Kanemitsu
Room 1001(10F) Build. A, 1404-1, Katakuramachi, Hachioji City, Tokyo 192-0982, Japan.
Tel.: +81-42-637-2491
E-mail: kanemitsuh@stf.teu.ac.jp

Kenji Kanai
Waseda Research Institute for Science and Engineering, Waseda University, Tokyo, Japan.
E-mail: k.kanai@aoni.waseda.jp

Jiro Katto
Department of Computer Science and Communications Engineering, School of Fundamental Science and Engineering, Waseda University, Tokyo, Japan.
E-mail: katto@waseda.jp

Hidenori Nakazato
Department of Computer Science and Communications Engineering, School of Fundamental Science and Engineering, Waseda University, Tokyo, Japan.
E-mail: nakazato@waseda.jp

virtualized resources (SF-CUV). The objective of SF-CUV is to minimize the makespan with less computational resources and containers than other algorithms by clustering tasks and sharing each container among tasks. SF-CUV consists of two phases: (i)task clustering and pre-virtual CPU (vCPU) allocation phase to derive an accurate scheduling priority, and (ii)task ordering and actual task reallocation phase. Experimental results obtained via simulation and in a real environment show that SF-CUV can utilize both vCPUs and containers with a shorter makespan compared with other approaches.

Keywords Task scheduling · Task clustering · Workflow scheduling · Containerized task · Task clustering · Resource Utilization · Cloud

1 Introduction

Recent advancements in communication networks include virtualization of network equipment, such as routers, firewalls, and load balancers, to utilize information flows and satisfy quality-of-service (QoS) requirements from both users and system administrators. Such network equipment can be virtualized at a software level to provide the same functionality by network function virtualization (NFV) [1,2]. Each function is typically called a service function (SF) for a service-oriented processing or virtualized network function (VNF) for handling network flows. In particular, multiple functions can be chained to output the composite processing results by service function chaining (SFC) [3, 4], which can be implemented as a set of containers or virtual machines (VMs). A container can include multiple processes and executables, and the same container can be shared among "tasks"[1]. Sharing one container among multiple tasks (i.e., containerized tasks) can localize communications among them and redundant container image downloading can be avoided.

The objectives of conventional containerized task allocation algorithms for a chain-structured application are minimization of the communication latency among tasks [5,10,15,19], maximization of the service data rate [7,8], minimization of task allocation cost [10,12–17,20], minimization of the number of VM instances [6,11,20], minimization of the makespan [9,10,17,22], and optimizing the routing cost [7,13,15,16,19] by using a search algorithm, integer linear programming (ILP), etc. Though task allocation algorithms from the literature [6,7,10,11] allocate tasks with container or VM sharing, they do not consider the balance between the communication locality and the degree of parallelism. This is because they focus on only satisfying the capacity constraint when each task is allocated, rather than minimizing the makespan with effectively utilize computational resources. If a criterion for task allocation is to satisfy the capacity constraint, each task is allocated to a computational resource having a large residual capacity. As a result, many computational resources are required for task allocation, thereby computational resources may be exhausted.

[1] A "task" in this paper means a set of execution units that corresponds to one transactional command and is supposed to be atomic.

In a workflow where multiple containerized tasks have precedence relationships with others, the degree of parallelism, communication localization among tasks, and the required numbers of container image downloading are factors for determining the makespan. For conventional containerized tasks in a workflow, a task allocation algorithm with minimizing the embedding cost [18], a provisioning method [22], and a fair CPU resource sharing method [23] have been proposed. In a workflow, the literature [24] proposes a task scheduling algorithm to minimize the makespan by utilizing bandwidth. However, no criterion exists for suppressing the number of both computational resources and containers. In the above approaches for task allocation and scheduling for both a chain and a workflow, no policy has been established for simultaneously satisfying effective container sharing, maintaining the degree of task parallelism, and effective computational resource utilization.

In this paper, we propose a clustering-based containerized task scheduling algorithm for clouds, namely, sharable functional task clustering for utilizing virtualized resources (SF-CUV). The objective of SF-CUV is to minimize the makespan with less computational resources and containers than other algorithms by clustering tasks and sharing each container among tasks. SF-CUV consists of two phases; phase (i): the task clustering and pre-vCPU allocation phase and phase 2: task ordering and actual task reallocation phase. Thus, SF-CUV is applicable where multiple workflows having containerized tasks must be processed simultaneously in a limited number of computational resources. For example, a large image file must be transformed to be analyzed (e.g., to recognize specific objects in the image), it can be divided into sub-images to be processed in parallel. If no executable module has been installed on a computational resource, containers can be dynamically pulled and installed on it. In such a case, the makespan should be minimized by avoiding redundant container image downloading with SF-CUV. Since such requirements need a software-level control for communication and allocating each task to a processor, SF-CUV can be implemented on SDN or clouds. Experimental results obtained via simulation and in a real environment show that SF-CUV can utilize both vCPUs and containers with a shorter makespan compared to other approaches. The main contributions of this paper are as follows:

- We present that a container sharing approach in workflows for avoiding redundant container image downloading can lead to the makespan minimization.
- We present a two-phase approach for effectively utilize both vCPUs and containers for containerized task scheduling for workflows.

The remainder of this paper is organized as follows. Section 2 reviews related studies on conventional containerized task allocation or scheduling algorithms. Section 3 describes our assumed system and cost model. Section 4 introduces the proposed SF-CUV algorithm. Section 5 presents the experimental results. Finally, Section 6 concludes the paper.

2 Related work

In this section, we provide an overview of conventional approaches from three perspectives, i.e., containerized task allocation or scheduling algorithms for a chain, a workflow, and non-containerized task scheduling algorithms for a workflow.

2.1 Containerized task allocation algorithms for chain

A chain structured application is a simple form in a workflow. Since each task has only one predecessor task or successor task, only one task is processed for each time slot. Thus, once a task is allocated to a node, no scheduling policy is needed. In particular, this form is used for service function chaining (SFC) for both NFV and IoT applications. In conventional approaches for containerized task allocation algorithms for chain [5–17, 19, 20], several objectives can be considered from the network optimization perspective. Literature [5, 10, 15, 19] presents how to allocate each task to a node by which each link cost is optimized under the processing and communication capacity constraint. This objective is useful if the economic cost for the communication among nodes must be minimized. However, it does not lead to minimize the makespan and does not effectively utilize both containers and nodes. As for approaches to minimize the number of required containers or VMs [6, 11, 20], several containers can be shared tasks, but there is no guarantee to minimize the makespan. The literature [6] proposes a meta-heuristic based on "Variable Neighborhood Search" to minimize the number of VNF (Virtualized Network Function) instances with satisfying network flow requirements and constraints. The literature [11] proposes a greedy algorithm for VNF allocation for minimizing the number of VNF instances. The literature [20] proposes a VNF embedding algorithm based on Binary Integer Programming (BIP) to optimize the embedding cost and the number of VNF instances. The literature [7, 8] presents methods for accommodating multiple user requests. In particular, the literature [8], its objective is to minimize both the total traffic volume in all the tasks and the number of containers by distributing them over the network. However, since it does not take the degree of parallelism, several independent tasks may be processed on a node; thereby the throughput may be decreased owing to the reduction in the degree of parallelism.

In other approaches, realistic objectives such as operating expenditure (OPEX) and capital expenditure (CAPEX) optimization are considered [11, 13, 14, 16–18, 20]. In the literature [11], the objective is to minimize the number of containers by satisfying the flow rate constraint for each task. This objective was formulated by mixed-ILP (MILP), and the greedy-based task allocation algorithm is proposed. In the literature [13], the objective is to minimize the makespan using a column generation approach while assuming an SFC as only a chain structure. In the literature [14], a MILP-based optimization is proposed for optimizing both the number of VMs and the task request flow.

In the literature [16], the objective was to optimize the routing path among tasks by utilizing the bandwidth for each allocated node and to minimize the total consumed energy. In the literature [17], the objective is to find the optimal mapping for minimizing the rejection rate for allocating a task to a node. In the literature [20], the objective is to minimize the SFC embedding cost by satisfying the constraints in terms of the capacity, deadline, and the maximum number of allocated functions. If the deadline is imposed as a constraint, the makespan is not always minimized.

As for approaches for scheduling each function in an SFC, fair-weighted affinity-based scheduling (FWS) [9] algorithm attempts to select a task with the minimum possible completion time from the ready tasks; then, the task is allocated to the VM or the host with the maximum residual capacity. From the resource utilization perspective, FWS tries to allocate as many VMs as possible; therefore, it does not effectively utilize computational resources. CoordVNF [10] tries to find the optimal allocation target by a breadth-first search while considering the residual processing capacity for each node. Thus, it tries to allocate a task to the nearest node from the previously allocated node, i.e., the criterion of CoordVNF is data locality. If an SFC is data-intensive, the makespan can be effectively made smaller; otherwise, it tries to allocate a task to a remote node with a large processing capacity. Thus, the computational resources can be exhausted.

2.2 Contanerized and non-containerized task scheduling algorithms for workflow

Many workflow scheduling algorithms have been proposed for scientific applications, image processing, and analyzing for realistic phenomenons. With the recent spreading of container technologies, it is a natural strategy to handle a task as a container even in a workflow.

As for containerized task scheduling algorithms for a workflow, there are several approaches. In the literature [18], the objective is to minimize the total traffic flow cost and the execution cost in a workflow, not to minimize the makespan. In the literature [22], a workflow is assumed and its objective is to minimize the makespan and the task deployment time. Though the algorithm in the literature [22] provides a good makespan, it does not consider resource utilization such as the number of nodes. In the literature [23], a fair CPU sharing method for each container in Montage workflow is proposed. Each CPU computing power is allocated to each container in order to achieve fairness in terms of the runtime, CPU usage, and the number of tasks among containers. Though this method tries to optimize the response time for each container, it does not include the communication optimization among containers. Thus, the makespan is not always optimized in this method. The literature [24] proposes a task scheduling algorithm to minimize the makespan by utilizing bandwidth. In this method, the critical path that includes each container image downloading time is derived. As a result, the communication in terms of

container image downloading can be effectively suppressed by minimizing the makespan. However, no policy for suppressing the number of computational resources is included.

From the non-containerized task scheduling perspective, the primary objective of task scheduling is to minimize the makespan. Although list-based task scheduling algorithms [34,36] and clustering-based task scheduling algorithms [21,25–27] have been proposed for scheduling tasks in a heterogeneous system, they cannot be applied directly for container utilization. In particular, CMWSL [21] assumes the network where each node has only one processing unit, and it attempts to cluster several tasks until a lower bound in terms of the total workload is exceeded. Thus, if CMWSL is applied to a virtualized system such as multiple VMs in a cloud, the lower bound can be a bottleneck for makespan minimization. The reason is that the more VMs each node has, the more edges among tasks can be already localized by allocating them to such VMs; that is, imposing the lower bound for data localization degrade the degree of parallelism in a cloud. Though SF-CUV uses the same measure as CMWSL [21] in terms of the clustering metric, named as WSL presented in Section 4.2.1, the following points are different among them.

- Task allocation policy: SF-CUV uses a task allocation criterion for minimizing the makespan (defined at Eq. (16)) based on the actual mapping state, while CMWSL tries to allocate with assuming a homogeneous system that every task on the path dominating the WSL is supposed to be allocated to the specific node. Thus, CMWSL cannot derive the scheduling priority based on the actual mapping, while SF-CUV can.
- Task cluster size adjustment: In SF-CUV, each task cluster size, i.e., the sum of task workload in a task cluster, is determined based on the actual task clustering state. However, in CMWSL, the lower bound for the task cluster size is initially derived by assuming a homogeneous system. As a result, the degree of parallelism or the communication localization by the clustering steps in CMWSL may not be appropriate to minimize the makespan when each processing speed or each communication bandwidth for each node is varied widely.

In other clustering-based task scheduling algorithms in a cloud [25], it clusters tasks according to conditional probability in terms of the occurrence. Such a method is useful when the given workflow includes if/else edges. However, it cannot be applied directly to a workflow in which every task must be processed. The task clustering algorithm called ECOS [26] attempts to minimize the cost within the deadline constraint by vertical and horizontal clustering on a cloud. In vertical clustering, each sequential task is clustered to localize the communication among tasks, while parallel tasks are clustered to minimize the cost within the deadline. The factor for makespan minimization in ECOS is only data localization in sequential tasks. The task clustering algorithm in the literature [27] partitions into K task clusters, and in each task cluster with considering data locality and parallelism. Then in each task cluster, list-based scheduling is applied. Since the assumed environment is a homogeneous sys-

Table 1 Notations for SF-CUV algorithm

Notation	Explanation	Definition
$n_i \in N$	i-th node having VMs in the set of nodes N.	
$c_{i,j} \in C_i$	j-th CPU core in n_i.	
$c_{i,j,k} \in C_{vcpu}$	k-th vCPU in CPU core $c_{i,j}$	
$M_i \in M$	Set of VMs in n_i.	
$m_{i,j} \in M_i$	j-th VM in M_i.	
$\phi : M \mapsto C_{vcpu}$	Set of vCPUs in the VM.	
$S = (V, E)$	Workflow	
$v_i \in V$	task	
w_i	Workload of v_i	
$e_{i,j} \in E$	Data dependency from v_i to v_j	
$d_{i,j}$	Data size from v_i to v_j	
$pred(v_i)$	Set of predecessor tasks of v_i	
$suc(v_i)$	Set of successor tasks of v_i	
$D(v_i)$	Container image size that includes v_i	
$T_{DL}(D(v_i), m_{k,l})$	Container downloading time on $m_{k,l}$	(1)
$T_p(v_i, c_{k,l,m})$	Exec. time of v_i on $c_{k,l,m}$	
$T_c(d_{i,j}, L_{k,p})$	Commun. time of $d_{i,j}$ from $m_{k,l}$ to $m_{p,q}$	(2)
$U(c_{k,l,m}, v_i)$	vCPU usage rate (0 – 100 %)	
$U_{th}(c_{k,l})$	Maximum allowed usage rate in a core (0 – 100 %)	
$T_{dr}(v_j)$	Data ready time of v_j	(5)
$T_s(v_j, A(v_j))$	Start time of v_j on $A(v_j)$	(6)
$T_f(v_i, A(v_i))$	Finish time of v_j on $A(v_j)$	(7)
$WSL(cls(i))$	$TL(cls(i)) + BL(cls(i))$	(8)
$top(i)$	Set of tasks that begin execution first in $cls(i)$	
$in(i)$	Set of tasks in $cls(i)$ with incoming edges from other clusters	
$out(i)$	Set of tasks sending data to other clusters	
$desc(v_k, cls(i))$	Set of descendant tasks of v_k in $cls(h)$, including v_k itself	
$TL(cls(i))$	Latest start time in $top(i)$	(9)
$tlevel(v_k)$	Latest start time of v_k	(10)
$BL(cls(i))$	Longest remaining time of $out(i)$	(12)
$blevel(v_k)$	Longest remaining time of v_k	(13)

tem, it cannot be applied directly to a virtualized environment where each VM has various processing speeds and each node has different communication bandwidth.

2.3 Difference with conventional approaches

Though SF-CUV algorithm tries to minimize the makespan by utilizing both vCPUs and containers by clustering tasks and sharing containers among tasks, any one of above cited approaches do not have specific criteria to satisfy both conditions, i.e., vCPU and container utilization. If a conventional containerized task scheduling algorithm is applied for multiple workflows, more vCPUs must wait for exections than the case of SF-CUV. If a non-containerized task scheduling algorithm is applied for scheduling containerized tasks, more containers must be downloaded for tasks than the case of SF-CUV.

3 System model

In this section, we define the parameters of the system, network, workflow, and the cost model. Note that all notations are listed on Table 1.

3.1 Network model

We assume that a network contains heterogeneous computational resources, including one or more cloud computing nodes with virtual machines (VMs) to process data from IoT devices, such as cameras and sensors. In Fig. 1, we assume that there are one or more networks including a container repository having container images, and several nodes having multiple VMs. Each VM has one or more virtual CPUs (vCPUs), each of which is mapped to one logical CPU (LCPU). From this state, each vCPU is an allocation target for tasks in workflows.

From logical points of view, let $N = \{n_1, n_2, \dots\}$ be the set of nodes with multiple VMs. The CPU in n_i is denoted as $C_i = \{c_{i,1}, c_{i,2}, \dots\}$, where $c_{i,k}$ is the k-th CPU core in C_i and $n_i \in N$. Let the set of VMs in n_i be $M_i = \{m_{i,1}, m_{i,2} \dots\}$. The processing speed (i.e., clock frequency) and communication bandwidth of $c_{i,j}$ are $\alpha_{i,j}$ and $\beta_{i,j}$, respectively. For each CPU core $c_{k,l} \in n_k$, we assume that $c_{k,l}$ has one or more vCPUs (where one vCPU corresponds to one logical CPU). Since each VM processes tasks on vCPUs, the allocation target for each task should be each vCPU. Each vCPU is denoted as $c_{k,l,m}$. Here, $1 \leq m \leq 2$ because of hyper-threading [30], which transforms one CPU core into two vCPUs. The set of vCPUs is defined as C_{vcpu}. Though the mapping between each CPU core and each vCPU is typically controlled by a hypervisor, we can define the actual mapping by CPU pinning [31]. Thus, we assume that the mapping can be known before scheduling tasks.

3.2 Workflow and task

We assume that workflow can be expressed in such a general form, i.e., DAG. We assume that there is at least one workflow to be deployed over the network. A workflow is defined as a graph $S = (V, E)$, where V is a set of tasks and E is a set of directed edges that show the data dependencies between the tasks. The i-th task is $v_i \in V$, and data transmission from v_i to v_j is denoted as $e_{i,j} \in E$. Further, w_i is the workload of v_i and $d_{i,j}$ is the data size of $e_{i,j}$. In a workflow, a task cannot begin execution until all the data from its preceding tasks arrive. Let $pred(v_i)$ and $suc(v_i)$ be the sets of predecessors and successors of v_i, respectively. If $pred(v_i) = \emptyset$, then v_i is called the START task, and if $suc(v_i) = \emptyset$, then v_i is called the END task.

3.3 Cost model

In this section, we define the cost for the execution time for each task and the communication time among tasks. In general, flow-level and instruction-level analyses are necessary to accurately estimate the execution time of unknown tasks. For known tasks, i.e., estimating the current execution time for already allocated tasks, empirical or regression-based analysis can be applied [28, 29].

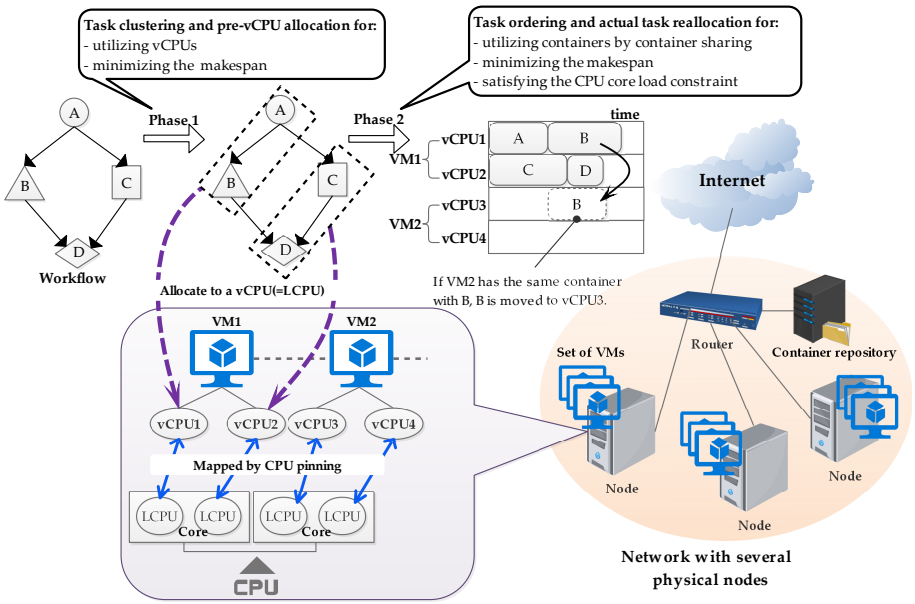


Fig. 1 Highlight of SF-CUV algorithm.

For complex workflows having many types of tasks and dependencies, an accurate estimation for both the task execution time and the communication time can be achieved to some extent by accumulating each analysis result. In this paper, we assume the situation that the execution time and the communication time for each task to be scheduled can be estimated from past execution results and input data. Let w_i be the workload of v_i. The size of the data that must be sent from v_i to v_j is denoted as $d_{i,j}$. Here, we assume that v_i belongs to a container and its image size is $D(v_i)$. Each container is supposed to run on a VM. Then the container image downloading time at the VM $m_{k,l}$ is defined by $T_{DL}(D(v_i), m_{k,l})$ as follows:

$$T_{DL}(D(v_i), m_{k,l}) = \begin{cases} 0, \text{ if } m_{k,l} \text{ has the container.} \\ \frac{D(v_i)}{\beta_k}, \text{ otherwise.} \end{cases} \tag{1}$$

The processing time of v_i on $c_{k,l,m}$ is defined as $T_p(v_i, c_{k,l,m}) = \frac{w_i}{\alpha_{k,l,m}}$, and the communication time of $d_{i,j}$ from $m_{k,l}$ to $m_{p,q}$ is defined as

$$T_c(d_{i,j}, L_{k,p}) = O_k + \frac{d_{i,j}}{\min\{\beta_k, \beta_p\}} = O_k + \frac{d_{i,j}}{L_{k,p}}, \tag{2}$$

where O_k is the setup time for the data transmission, and it is negligible compared to the actual communication time. Thus, O_k is supposed to be zero for any node in N. Further, β_k is the communication bandwidth specified by the NIC of n_k, i.e., the maximum bandwidth. If the data are transmitted between vCPUs that belong to the same host machine, we assume that the

communication time is negligible (equal to zero) compared to the network delay. If the objective of task allocation is to optimize CAPEX or OPEX, each vCPU has a threshold of the total processed workload that should not be exceeded in task allocation. As such a threshold can typically be specified by a user or some system management policies, the set of allocated tasks for each vCPU can be changed. Thus, the makespan can be varied depending on these policies. The computational load of a task, i.e., how long the process started by the task occupies the CPU per time unit depends on the structure of the task, operating system, and the hypervisor. In this paper, we assume that each task is processed without preemption by a vCPU per time unit to fully utilize each CPU capacity. Typically, the mapping between each vCPU and each logical CPU depends on the hypervisor on the cloud. However, depending on the mapping policy of the hypervisor, one logical CPU can be mapped to two or more vCPUs. In such a case, the economic cost regarding the number of vCPUs can be suppressed at the expense of the processing speed. Each vCPU can be fully utilized in a system such as an on-premise cloud, the processing speed for each vCPU should not be degraded if the total number of used vCPUs is less than that of logical CPUs. An application with a high degree of parallelism requires many vCPUs without degrading the performance to minimize the makespan. Thus, manually mapping between vCPUs and logical CPUs is needed such as CPU pinning. In this paper, an on-premise cloud is assumed and every allocated task is not preempted by other tasks at a time unit to finish its execution as early as possible. In the case of a high-load vCPU, the throughput is degraded. Thus, the threshold in terms of the processing load for each vCPU should be determined to avoid degrading its processing speed. We define the threshold of the processing load (60%, 80%, and so on). Let $U(c_{k,l,m}, v_i)$ be the vCPU usage rate when v_i is executed by $c_{k,l,m}$. Further, $|c_{k,l}|$ is the number of logical CPUs per CPU core $c_{k,l}$, e.g., $|c_{k,l}| = 2$ implies that hyper-threading is enabled in core $c_{k,l}$. If one logical CPU corresponds to one vCPU, two vCPUs run simultaneously for $|c_{k,l}| = 2$. As a capacity constraint in terms of parallel execution by vCPUs, we define the condition that vCPUs in a CPU core can simultaneously execute tasks without degrading their processing speed as follows:

$$\frac{1}{|c_{k,l}|} \sum_{m=1}^{|c_{k,l}|} U(c_{k,l,m}, v_i) \leq U_{th}(c_{k,l}), \tag{3}$$

where $U_{th}(c_{k,l})$ is the maximum usage ratio in core $c_{k,l}$ as determined by the system administrator. Eq. (3) indicates that the average usage of a vCPU must not exceed the maximum usage ratio in a core for parallel execution.

3.4 Objective function

We define that the allocation target of v_i determined by a scheduling algorithm is $A(v_i)$, where $A(v_i)$ is a vCPU. Here, we present the process of determining

the start time of each task v_i on $A(v_i)$ when scheduling tasks and deriving the scheduled length. If $T_s(v_i, A(v_i))$ is the start time, $T_p(v_i, A(v_i))$ is the processing time, and $T_f(v_i, A(v_i))$ is the finish time of v_i on $A(v_i)$, then

$$T_f(v_i, A(v_i)) = T_s(v_i, A(v_i)) + T_{DL}(v_i, \phi^{-1}(A(v_i))) + T_p(v_i, A(v_i)), \quad (4)$$

where $A(v_i) \in M \cup C_{vcpu}$ and $\phi^{-1}(A(v_i))$ is the VM that $A(v_i)$ belongs to. (4) means that the container downloading is required even if every data has arrived at v_i. The set of free tasks (i.e., the set of tasks whose predecessors have been scheduled) is denoted as $fList$. For each task in $fList$, the data-ready time (DRT) can be derived. The DRT is the latest time of data arrival from all preceding tasks. Notably, the DRT is the earliest start time for each task; however, the actual start time may be later than the DRT. This is because v_j cannot start execution when an unfinished task $v_h \notin pred(v_j)$ such that $A(v_h) = A(v_j)$ is scheduled before v_j. The DRT of v_j at m_k is defined as follows:

$$T_{dr}(v_j) = \max_{v_i \in pred(v_j)} \{T_f(v_i, A(v_i)) + T_c(d_{i,j}, L_{k,p})\}, \quad (5)$$

where $T_c(d_{i,j}, L_{k,p}) = 0$ if $k = p$; $A(v_i) \in n_k$ and $A(v_j) \in n_p$. Then, we can calculate the start time $T_s(v_j, A(v_j))$ using the DRT:

$$T_s(v_j, A(v_j)) = \max \left\{ \max_{\substack{v_i \notin pred(v_j), \\ A(v_i) = A(v_j)}} \{T_f(v_i, A(v_i))\}, T_{dr}(v_j) \right\}. \quad (6)$$

The makespan is the finish time of the END task and is defined as follows:

$$T_f(v_{end}, A(v_{end})) = T_s(v_{end}, A(v_{end})) + T_p(v_{end}, A(v_{end})). \quad (7)$$

With the aforementioned definitions, the objective function is defined as follows:

Objective 1 *Minimize $T_f(v_{end}, A(v_{end}))$ by utilizing both vCPUs and containers with satisfying Eq. (3).*

4 SF-CUV algorithm

In this section, we present the SF-CUV algorithm. First, we present an overview of SF-CUV, and then we provide a detailed explanation of the algorithm with examples.

4.1 Overview of SF-CUV

As shown in Fig. 1, SF-CUV consists of two phases. In the first phase, the task clustering and pre-vCPU allocation are performed to suppress the number of allocated vCPUs and minimize the makespan. In the second phase, task ordering and actual task reallocation are performed to suppress the number of containers and minimize the makespan by container sharing.

Algorithm 1 presents the whole procedure of the SF-CUV algorithm. Lines 1–22 in Algorithm 1 correspond to phase (i) and lines 23–28 in Algorithm 1 correspond to phase (ii). A detailed description for each phase is given in the sections that follow.

4.2 Task clustering and pre-vCPU allocation phase

4.2.1 Clustering metric: WSL

In this phase, we assume that each task belongs to a "single task cluster" with only one task. Then, several task clusters are merged into a larger task cluster to minimize the possible makespan. The i-th task cluster is denoted as $cls(i)$, where the initial task cluster is denoted as $cls(i) = \{v_i\}$. We define the operation of "clustering $cls(i)$ and $cls(j)$ together" is the merging, that is as denoted as $cls(i) \leftarrow cls(i) \cup cls(j)$. If a task in $cls(i)$ is an immediate predecessor of a task in $cls(j)$, that communication is localized by the clustering. Since such a data localization is one of factors to minimize the makespan, a clustering step in this paper assumes at least one data transfer is required between $cls(i)$ and $cls(j)$. Thus, clustering from $cls(i)$ to $cls(j)$ as "downward clustering (presented in Section 4.2.3)" and vice versa as "upward clustering (presented in Section 4.2.4)" are possible. Notably, the makespan cannot be determined until both the allocation target and the execution order for each task have been determined. Thus, a measure for estimating the makespan is used in this phase, which is called the worst schedule length (WSL) [21]. The WSL is the maximum execution path length when each task is executed as late as possible, provided that there is no data-waiting time from other task clusters. Moreover, it has been shown that minimizing the WSL can minimize both the upper and the lower bounds of the makespan [21].

The main objective of this phase is to minimize WSL by suppressing the number of vCPUs through task clustering steps.

Here, we highlight the WSL derivation[2] and the list of notation used in the following sections is given in Table 1. WSL of $cls(i)$ is defined as

$$WSL(cls(i)) = TL(cls(i)) + BL(cls(i)), \tag{8}$$

where $TL(cls(i))$ is the latest data arrival time in the tasks that can execute first in $cls(i)$, and $BL(cls(i))$ is the longest "execution path" length in $cls(i)$

[2] Fore more details for WSL derivation, refer to the literature [21].

plus the longest remained path length until the END task finishes. That is, WSL is made larger if more tasks each of which is independent of each other are included in the same task cluster. Here $WSL(cls(i))$ consists of the "upper part: $TL(cls(i))$" and "lower part: $BL(cls(i))$" of $cls(i)$. First, we define the upper part, i.e., $TL(cls(i))$, as

$$TL(cls(i)) = \max_{v_k \in top(i)} \{tlevel(v_k)\}, \tag{9}$$

where $top(i)$ is the set of tasks that begin execution first in $cls(i)$. Next, we define a metric for the possible start time for each task in a cluster. Let $in(i)$ be the set of tasks with incoming edges from other task clusters, and let $out(i)$ be the set of tasks with outgoing edges to other task clusters. For each $v_k \in top(i), v_j \in pred(v_k), v_j \in out(h)$, we define

$$tlevel(v_k) = \max_{\substack{v_j \in pred(v_k),\\ v_j \in out(cls(h))}} \{tlevel(v_j) + T_p(v_j, A(h)) + T_c(d_{j,k}, L_{p,q})\}, \tag{10}$$

$$tlevel(v_j) = TL(cls(h)) + T_p(cls(h), A(j)) - T_p(desc(v_j, cls(h)), A(h))), \tag{11}$$

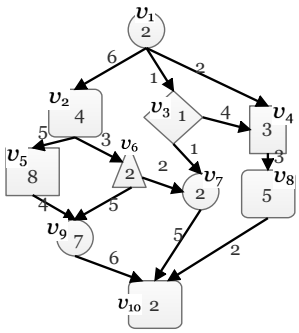
where we assume that $v_j \in cls(h)$, $v_k \in cls(i)$, and $A(h)$ is the vCPU to which $cls(h)$ is allocated. In Eqs. (10) and (11), we assume that $A(h) \in n_p,\ A(i) \in n_q,\ A(j) \in n_r$ and $desc(v_j, cls(h))$ is the set of descendant tasks of v_j in $cls(h)$, including v_j itself. For instance, if A, B, C, and D are included in an task cluster $cls(h)$ and we assume that $A \rightarrow B$, $A \rightarrow C$, $B \rightarrow D$, and $C \rightarrow D$, where "$\rightarrow$" is a precedence relationship, then $desc(B, cls(h)) = \{B, D\}$ and $desc(A, cls(h)) = \{A, B, C, D\}$. Finally, $tlevel(v_j)$ from Eq. (11) is the latest possible start time of v_j, provided that v_j is scheduled as late as possible when every task in $in(j)$ can begin execution without waiting for data arrival.

Next, we define the "lower part: $BL(cls(i))$" as

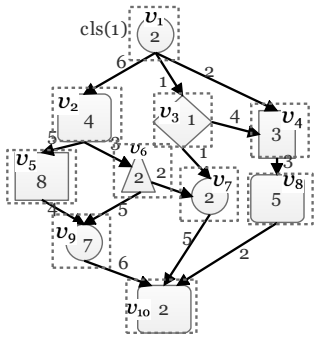
$$BL(cls(i)) = \max_{v_k \in out(i)} \{T_p(cls(i), A(i))) - T_p(desc(v_k, cls(i)), A(i)) + blevel(v_k)\}, \tag{12}$$

$$blevel(v_k) = \max_{\substack{v_l \in in(j),\\ v_l \in suc(v_k)}} \{T_p(v_k, A(i)) + T_c(d_{k,l}, L_{q,r}) + blevel(v_l)\}, \tag{13}$$

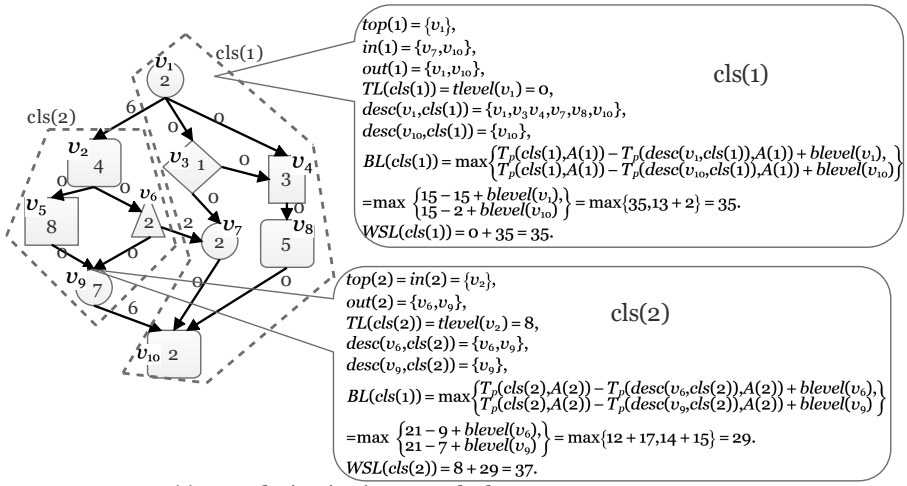
where we assume that $v_k \in cls(i),\ v_l \in cls(h)$, and $A(i) \in n_q, A(j) \in n_r$. Here $BL(cls(i))$ is the maximum possible residual time duration of the tasks in $cls(i)$, provided that each task can be scheduled as late as possible. Thus, the first and second terms in Eq. (12) imply the possible start time of a task in $cls(i)$ if it is scheduled as late as possible. Under such an assumption, $blevel(v_k)$ is the longest path length from v_k to the END task. Then $WSL(cls(i))$ is determined after each task is allocated (but before scheduling the tasks).



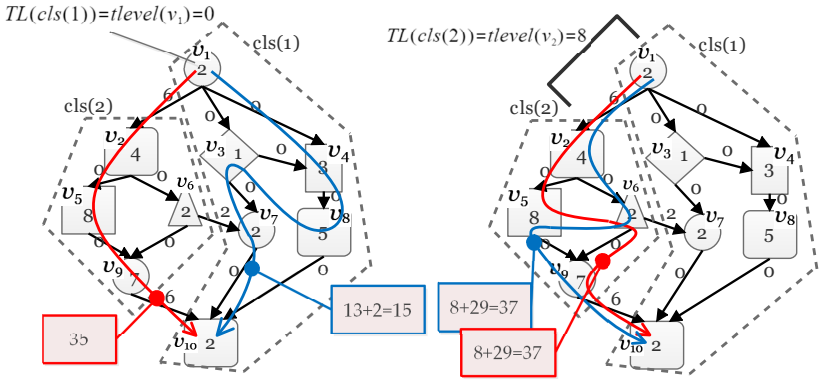
(a) Initial state.



(b) Set of single task clusters.



(c) WSL derivation in two task clusters.



(d) WSL(cls(1))=0+max{35, 15}=35.

(e) WSL(cls(2))=8+max{29,29}=37.

Fig. 2 Example of WSL derivation.

Example 1 *Fig. 2 shows an example of WSL derivation. In this figure, the shown graph is a DAG structured workflow where each edge defines the precedence relationship among tasks. Fig. 2(a) is the given workflow and (b) is the input workflow in which each task belongs to a respective single task cluster, and (c) is the status after two task clusters are generated. Fig. 2(d) and (e) are the paths of WSL for each task cluster, respectively. The cost value on every edge and task is the communication time and the processing time, respectively. For simplicity, in Fig. 2(c), we assume that* $cls(1)$ *and* $cls(2)$ *are allocated to different nodes distributed physically over the network. We assume that those nodes have the same processing speed and communication bandwidth equal to one, i.e., every cost value is not varied when each task is allocated to a node.*

In Fig. 2(c), $cls(1)$ *has only one top task, i.e.,* v_1 *because it is the START task. As for* $in(1)$*, because* v_7 *and* v_{10} *have incoming edges from* $cls(2)$*, they belong to* $in(1)$*. As for* $out(1)$*,* v_1 *has one outgoing edge to* $cls(2)$ *and* v_{10} *is the END task. Here* $desc(v_1, cls(1))$ *is the set of descendant tasks of* v_1 *in* $cls(1)$*, i.e.,* $v_1, v_3, v_4, v_7, v_8, v_{10}$ *belong to it. Thus,* $TL(cls(1))$ *and* $BL(cls(1))$ *are derived using the above sets, and* $WSL(cls(1))$ *is derived as 35. Similarly, we can derive* $WSL(cls(2))$ *as 37.*

In Fig. 2(d), the red arrow is the path of $WSL(cls(1))$*. Since* $BL(cls(1))$ *is derived from only the set of* $out(cls(1)) = \{v_1, v_{10}\}$*, WSL is derived by taking the largest one from* $tlevel(v_1) + blevel(v_1)$ *and* $tlevel(v_{10}) + blevel(v_{10})$*. Their values are* $0 + 35$ *and* $13 + 2$*, thus we have* $WSL(cls(1)) = 35$*. Similarly, in Fig. 2(e),* $WSL(cls(2))$ *is derived by taking the largest one from* $tlevel(v_6) + blevel(v_6) = 8 + 29$ *and* $tlevel(v_9) + blevel(v_9) = 8 + 29$ *because we have* $out(cls(2)) = \{v_6, v_9\}$*.* □

In the following sections, we describe details first step of the SF-CUV algorithm using WSL.

4.2.2 Summary of clustering in SF-CUV

First, we assume that every task cluster has only one task and each task cluster belongs to UEX, i.e., the set of unchecked task clusters (line 1 in Algorithm 1). task clusters are checked to see whether they satisfy the clustering condition, then are removed from UEX (lines 7 and 11 in Algorithm 1), and the algorithm finishes when $UEX = \emptyset$.

In line 3 in Algorithm 1, SF-CUV attempts to select the task cluster as the "pivot" with the maximum WSL by Eq. (8) from the task cluster list $FREE$ (line 3 in Algorithm 1). Here $FREE$ is the set of task clusters that are ready for the clustering procedures. The condition that a cluster $cls(i)$ belongs to $FREE$ is that all the predecessors of $top(cls(i))$ do not belong to UEX (defined in Eq. (17)). That is, all predecessor tasks of $top(cls(i))$ have been checked. From the pivot, the succeeding or preceding task cluster is selected as the clustering target. Clustering from *pivot* to a succeeding task cluster is defined as "downward clustering" (lines 5–8 in Algorithm 1), whereas clustering from *pivot* to a preceding task cluster is defined as "upward clustering" (lines 9–12 in Algorithm 1).

Algorithm 1: SF-CUV algorithm

Input: Set of vCPUs C_{vcpu} and workflow $S = (V, E)$, and set of task clusters $CLS = V$.
Output: The mapping from $cls(i) \in CLS$ to C_{vcpu} and the schedule.

```
/* UEX is the set of unclustered task clusters and FREE is the set of task
   clusters that are ready for clustering.                                  */
```
1 $UEX \leftarrow CLS$, $FREE \leftarrow$ START task clusters
2 **while** *($UEX \neq \emptyset$) OR ($\sharp$ of $cls(i) \geq 1$ s.t., $A(cls(i)) = \emptyset$)* **do**
3 Select $cls(i)$ having the maximum WSL in $FREE$ using the average processing speed α_{ave} and the average communication bandwidth β_{ave}
4 $pivot \leftarrow cls(i)$.
```
   /* Suppose that v_t ∈ top(pivot) dominates TL(pivot) and v_b ∈ out(pivot)
      dominates BL(pivot).                                                  */
   /* Suppose that v_c ∈ suc(v_b) dominates blevel(v_b) where v_c ∈ cls(j), and
      v_s ∈ pred(v_t) dominates tlevel(v_t), where v_s ∈ cls(h).             */
```
5 **if** *(14) is satisfied with pivot and $cls(j)$* **then**
6 $target \leftarrow cls(j)$
```
   /* Downward clustering.                                                  */
```
7 $UEX \leftarrow UEX - \{pivot, target\}$.
8 $pivot \leftarrow pivot \cup target$.
9 **else if** *(15) is satisfied with pivot and $cls(h)$* **then**
10 $target \leftarrow cls(h)$
```
   /* Upward clustering.                                                    */
```
11 $UEX \leftarrow UEX - \{target \cup pivot\}$.
12 $pivot \leftarrow target \cup pivot$.
13 **else**
```
   /*                                                                       */
```
14 **if** $A(pivot) = \emptyset$ **then**
15 Select $c_{q,l,m} \in C_{vcpu}$ satisfying (16) and allocate $c_{q,l,m}$ to $pivot$.
16 **else**
```
   /* Update FREE if one or more succeeding task clusters satisfy (17).
      */
```
17 **if** ($\sharp$ *of* $cls(j)) \geq 1$ *at (17) by tracing successors of* $out(pivot)$ **then**
18 $FREE \leftarrow FREE - \{pivot\}$.
19 Put each task cluster $cls(j)$ into $FREE$.
20 **else**
```
   /* Search for the next pivot candidate that does not effect the
      maximum WSL among FREE by the clustering step.                        */
```
21 $pivot \leftarrow cls(x) \in FREE$ having the next largest WSL value s.t., $A(cls(x)) = \emptyset$, then go to line 5 or 9 or 13. If no such task cluster $cls(x)$ is found, remove $pivot$ from $FREE$.
```
   /* Update Process.                                                       */
```
22 $pivot \leftarrow Update(pivot)$.
```
/* task ordering and actual allocation.                                    */
/* fList is the set of tasks whose all predecessor tasks have been scheduled, i.e.,
   not included in UEX_task.                                               */
```
23 Let $UEX_{task} \leftarrow V$, $fList \leftarrow$ START tasks.
24 **while** $UEX_{task} \neq \emptyset$ **do**
25 Select the task v_i satisfying (18) from $fList$.
```
   /* Find the vCPU as the allocation target by considering minimization of
      T_f(v_i, A(v_i)) while satisfying (3) and container sharing.          */
   /* Call Algorithm 2.                                                     */
```
26 $A(v_i) \leftarrow RefinedAllocation(v_i)$
27 $fList \leftarrow fList - \{v_i\}, UEX_{task} \leftarrow UEX_{task} - \{v_i\}$, and add tasks $v_j \in suc(v_i)$ to $fList$ s.t., $\forall v_l \in pred(v_j), v_l \notin UEX_{task}$.
28 **return** V *and* C_{vcpu}.
29 **Function** `Update`($cls(i)$)**:**
30 Update $top(i), in(i), out(i)$.
```
   /* Update WSL of the pivot and the successor clusters in FREE.           */
```
31 Update $TL(i)$ and $BL(i)$.
32 Update $TL(j)$ by tracing $\forall v_b \in out(i)$ and $\forall v_c$ s.t., $v_c \in suc(v_b), v_c \in cls(j), cls(j) \in FREE$.
33 **return** *cls(i)*.

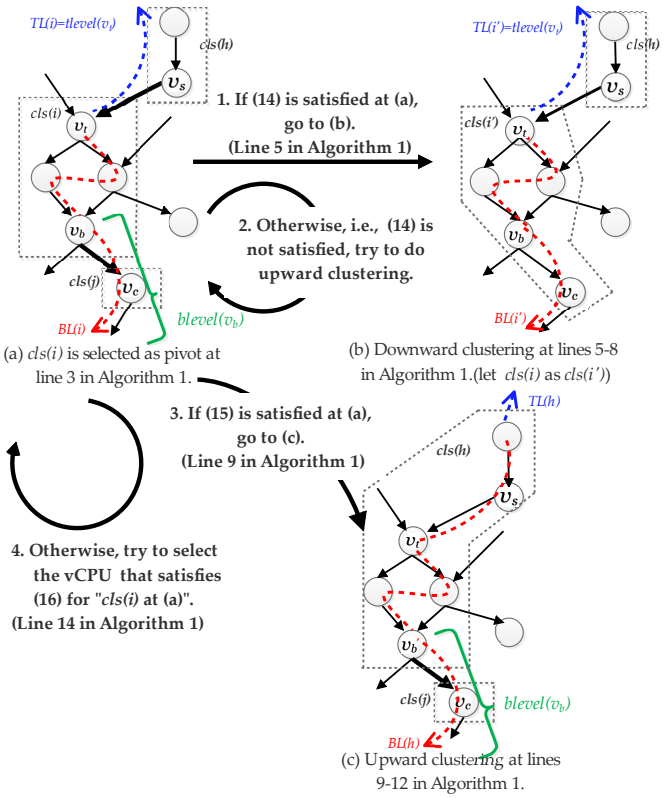


Fig. 3 Clustering criteria at (14) and (15).

4.2.3 Downward clustering

In line 5 in Algorithm 1, the condition for downward clustering from the pivot ($cls(i)$) to $cls(j)$ is accepted is that WSL is made smaller, that is defined as follows:

$$WSL(cls(i) \cup cls(j)) = TL(cls(i)) + T_p(cls(i) \cup cls(j), \alpha_{ave}) - desc(v_c, cls(i) \cup cls(j), \alpha_{ave}) + blevel(v_c) \leq WSL(cls(i)), \quad (14)$$

where we assume that $v_b \in out(i)$ dominates $BL(i)$, $v_c \in suc(v_b)$ dominates $blevel(v_b)$, and $v_c \in cls(j)$. Fig. 3 shows how $cls(i)$ is checked for the clustering step. Fig. 3(a) shows that $cls(i)$ is selected in line 3 in Algorithm 1, Fig. 3(b) is the result of a downward clustering where $cls(i)$ is clustered with $cls(j)$ in Fig. 3(a). Then the generated cluster is denoted as $cls(i')$ in Fig. 3(b). Fig. 3(c) is the result of an upward clustering where $cls(h)$ is clustered with $cls(i)$ in Fig. 3(a). If (14) is satisfied in Fig. 3(a), the process goes to Fig. 3(b).

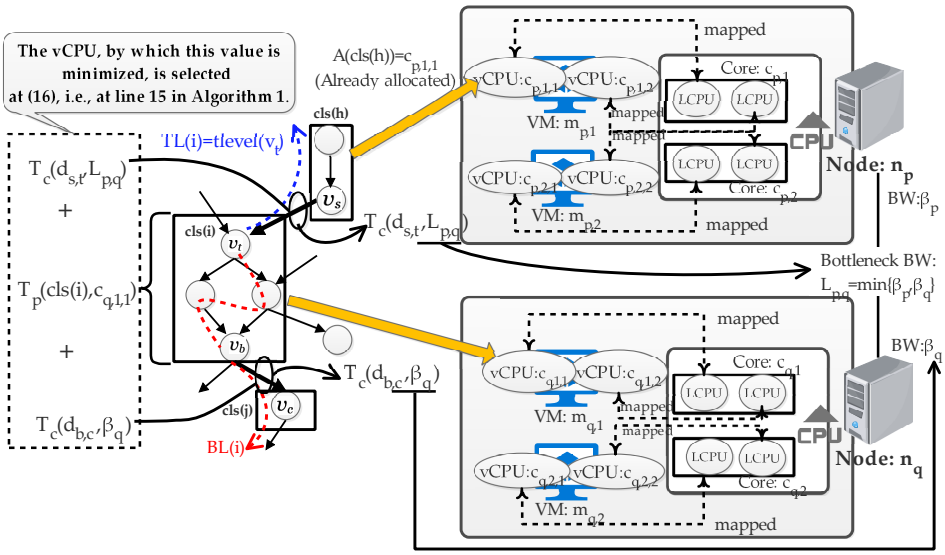


Fig. 4 vCPU selection for $cls(i)$ at (16), where it is assumed that $c_{q,l,m}$ at (16) is $c_{q,1,1}$.

4.2.4 Upward clustering

If downward clustering is not accepted, i.e., (14) is not satisfied, the algorithm attempts to perform upward clustering (Fig. 3(c)). In line 9 in Algorithm 1, the condition for upward clustering is accepted is that WSL is made smaller, that is defined as follows:

$$WSL(cls(h) \cup cls(i)) = TL(cls(h)) + T_p(cls(h) \cup cls(i), \alpha_{ave}) - desc(v_b, cls(h) \cup cls(i), \alpha_{ave}) + blevel(v_b) \leq WSL(cls(i)), \quad (15)$$

where $cls(i)$ is the pivot and we assume that $v_t \in top(i)$ dominates $TL(cls(i))$, $v_s \in pred(v_t), v_s \in cls(h)$ dominates $tlevel(v_t)$, and v_b dominates $BL(i)$, as described in Fig. 3(a). From this state, if (15) is satisfied in Fig. 3(a), the process goes to Fig. 3(c). This corresponds to the upward clustering in Fig. 3(c) being accepted if (15) is satisfied. If either condition (Eq. (14) or Eq. (15)) is satisfied, both clusters are removed from UEX.

4.2.5 vCPU allocation

Assuming that both Eqs. (14) and (15) are not satisfied in lines 13–21 in Algorithm 1, if no vCPU has been allocated to the pivot (line 14), the vCPU satisfying the following condition is selected for allocating to the pivot (let the pivot be $cls(i)$):

$$\min_{c_{q,l,m} \in \ unallocated \ vCPUs} \{T_c(d_{s,t}, L_{p,q}) + T_p(cls(i), c_{q,l,m}) + T_c(d_{b,c}, \beta_q)\} \quad (16)$$

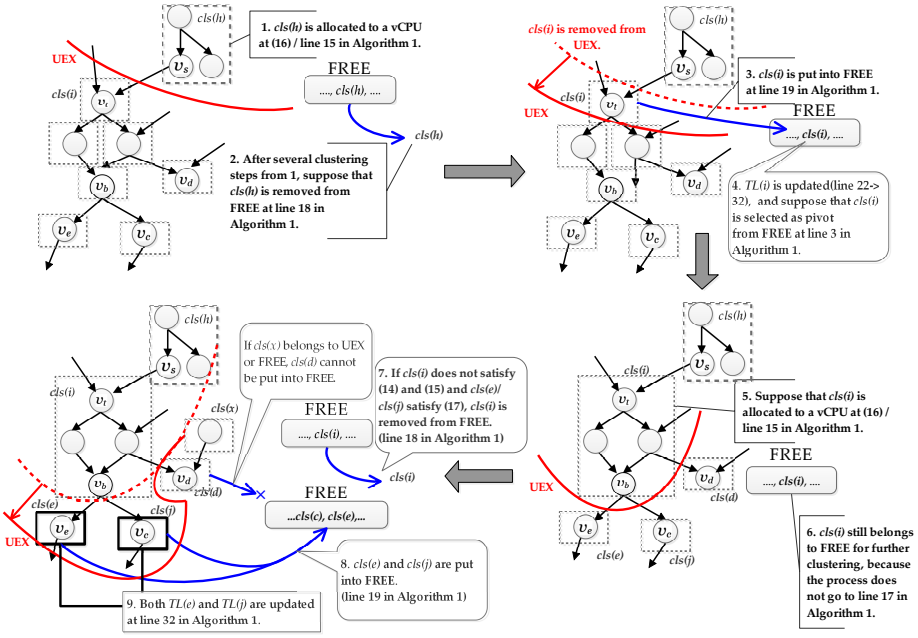


Fig. 5 Example of update process.

where $v_s \in cls(h)$ dominates $tlevel(v_t)$ and $v_t \in top(i)$ dominates $TL(cls(i))$. Moreover, $v_b \in cls(i)$ dominates $BL(cls(i))$ and v_c dominates $blevel(v_b)$, where $v_c \in cls(j)$ (see Fig. 4). Eq. (16) implies that SF-CUV tries to find the vCPU by which the time duration from the incoming communication start time to the outgoing communication finish time is minimized. Fig. 4 shows one example of the vCPU selection criteria at (16), where notation in this figure corresponds to that in (16). In Fig. 4, it is assumed that $A(cls(h)) = c_{p,1,1} \in n_p$, and $c_{q,1,1}$ is selected at (16), which corresponds to line 15 in Algorithm 1. We assume that $T_c(d_{s,t}, L_{p,q}) + T_p(cls(i), c_{q,1,1}) + T_c(d_{b,c}, \beta_q)$ is minimum in the set of unallocated vCPUs. If $p = q$, the communication of $d_{s,t}$ is localized, we have $T_c(d_{s,t}, L_{p,q}) = 0$. However, whether the communication time for $d_{b,c}$ is zero or not cannot be determined because the allocation target vCPU of task cluster $cls(j)$ in Fig. 4 has not been determined until $A(cls(j)) \neq \emptyset$. Thus, the communication time for $d_{b,c}$ is $T_c(d_{b,c}, \beta_q)$ by taking the bottleneck bandwidth as β_q. If both the processing speed and the communication bandwidth among n_p and n_q are the same, the allocation target for $cls(i)$ is selected from vCPUs in n_p for communication localization, though it is selected from n_q in Fig. 4.

4.2.6 Update procedures

After the allocation target vCPU has been determined for $cls(i)$, the WSL of $cls(i)$ is updated with the allocated vCPU (line 22 in Algorithm 1). This update procedure includes the update of $in(cls(i))$, $out(cls(i))$, $top(cls(i))$,

$TL(cls(i))$, and $BL(cls(i))$ to derive the new $WSL(cls(i))$ for the next pivot selection in line 3 of Algorithm 1. If a vCPU has been allocated to the pivot (line 15), the pivot still belongs to $FREE$ for further clustering steps, to suppress the number of required vCPUs and to minimize the makespan. This corresponds to line $15 \rightarrow 22 \rightarrow 3$ in Algorithm 1.

In line 3 of Algorithm 1, if the selected pivot has already been allocated to a vCPU and both the conditions for downward clustering and upward clustering are not satisfied, then the process goes to line 17 in Algorithm 1. This means that the process attempts to search for the new pivot candidates from succeeding task clusters of the pivot. The new task cluster ($cls(j)$) satisfying the following condition is put into $FREE$:

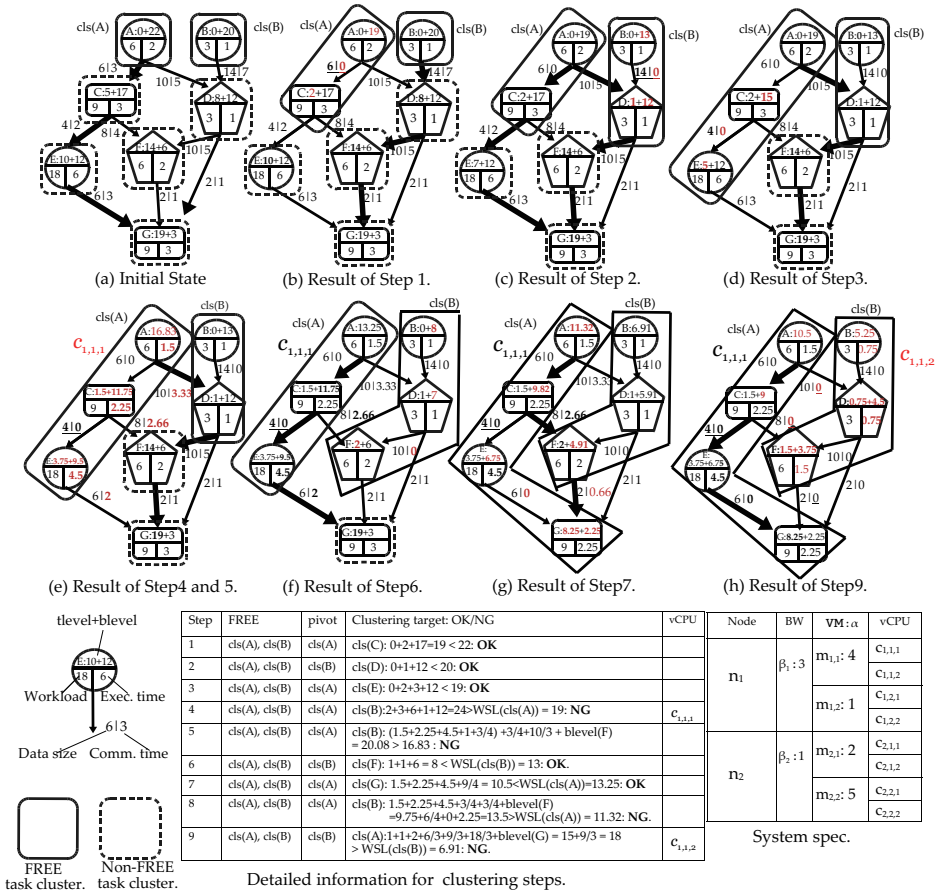
$$\{cls(j) | \forall v_t \in top(j), v_s \in pred(v_t),\ v_s \in cls(i) \\ s.t.\ \ A(cls(i)) \neq \emptyset\}. \tag{17}$$

In $cls(j)$, all the predecessor tasks of $top(j)$ are assumed to belong to traced task clusters. If such an task cluster $cls(j)$ exists, it is put into $FREE$ and the current pivot is removed from $FREE$ (line 17 in Algorithm 1). If no such task cluster exists, the current pivot still belongs to $FREE$ because the pivot can be selected and clustered depending on the subsequent clustering steps (line 20 in Algorithm 1). Then an task cluster having the next largest WSL value is selected as *pivot* (line 21 in Algorithm 1).

Example 2 *Fig. 5 shows an example of the update procedures in lines 29–33 in Algorithm 1. In 1 in this figure, $cls(h)$ is assumed to be allocated to a vCPU satisfying (16), then we assume that $cls(h)$ is selected as the pivot in a subsequent loop. However, we assume that $cls(j)$ does not satisfy both (14) and (15) and in 2 it is removed from $FREE$ in line 18 of Algorithm 1. In 3 in Fig. 5, $cls(i)$ is put into $FREE$, then we assume that WSL of $cls(i)$ is maximized in $FREE$. Thus, $cls(i)$ is selected as pivot for the clustering step. In 5 in Fig. 5, i.e., after several clustering steps, we assume that $cls(i)$ is allocated to a vCPU and $cls(i)$ still belongs to $FREE$ (lines 14–16 in Algorithm 1) . In 7, $cls(e)$ and $cls(j)$ are put into $FREE$. However, we assume that $cls(d)$ still belongs to UEX because $cls(x)$ is assumed to belong to UEX or $FREE$. Then in 8 in Fig. 5, both $TL(e)$ and $TL(j)$ are updated; that is, their WSL values are updated for further pivot selection from $FREE$. As parts in $cls(e)$ and $cls(j)$ affected by 5 in Fig. 5 are $TL(e)$ and $TL(j)$, only (i.e., BL values are not affected), and only TL values of task clusters in $FREE$ are updated.* □

In the next example, we present the whole procedures in the task clustering and pre-vCPU allocation phase.

Example 3 *Fig. 6 shows an example of whole procedures in the task clustering and pre-vCPU allocation phase of the SF-CUV algorithm. In this figure, the bold arrows indicate the execution path determining the maximum WSL among task clusters. From the initial state (a), each cost value is allocated according to the average processing speed ($(4+1+2+5)/4 = 3$) and the average bandwidth*



Step	FREE	pivot	Clustering target: OK/NG	vCPU
1	cls(A), cls(B)	cls(A)	cls(C): 0+2+17=19 < 22: **OK**	
2	cls(A), cls(B)	cls(B)	cls(D): 0+1+12 < 20: **OK**	
3	cls(A), cls(B)	cls(A)	cls(E): 0+2+3+12 < 19: **OK**	
4	cls(A), cls(B)	cls(A)	cls(B):2+3+6+1+12=24>WSL(cls(A)) = 19: **NG**	$c_{1,1,1}$
5	cls(A), cls(B)	cls(A)	cls(B): (1.5+2.25+4.5+1+3/4) +3/4+10/3 + blevel(F) = 20.08 > 16.83 : **NG**	
6	cls(A), cls(B)	cls(B)	cls(F): 1+1+6 = 8 < WSL(cls(B)) = 13: **OK**.	
7	cls(A), cls(B)	cls(A)	cls(G): 1.5+2.25+4.5+9/4 = 10.5<WSL(cls(A))=13.25: **OK**	
8	cls(A), cls(B)	cls(A)	cls(B): 1.5+2.25+4.5+3/4+3/4+blevel(F) =9.75+6/4+0+2.25=13.5>WSL(cls(A)) = 11.32: **NG**.	
9	cls(A), cls(B)	cls(B)	cls(A):1+1+2+6/3+9/3+18/3+blevel(G) = 15+9/3 = 18 > WSL(cls(B)) = 6.91: **NG**.	$c_{1,1,2}$

Detailed information for clustering steps.

Node	BW	VM : α	vCPU
n_1	β_1 : 3	$m_{1,1}$: 4	$c_{1,1,1}$
			$c_{1,1,2}$
		$m_{1,2}$: 1	$c_{1,2,1}$
			$c_{1,2,2}$
n_2	β_2 : 1	$m_{2,1}$: 2	$c_{2,1,1}$
			$c_{2,1,2}$
		$m_{2,2}$: 5	$c_{2,2,1}$
			$c_{2,2,2}$

System spec.

Fig. 6 Example of the clustering phase.

$((3+1)/2 = 2)$ from the "System spec." table. Each red- colored cost value is that updated from the previous clustering step by Update procedures in line 29 of Algorithm 1. The table located at the bottom in this figure indicates the state of $FREE$, how pivot is selected, and whether the clustering step is accepted.

In Fig. 6(a), $cls(A)$ and $cls(B)$ belong to $FREE$. Then $cls(A)$ is selected as pivot, because in line 3 of Algorithm 1, we have

$$WSL(cls(A)) = 22 > WSL(cls(B)) = 20.$$

Then, $cls(C)$ is selected as the target because C dominates $blevel(A)$ and (14) in this case is $0 + (2+3) - 3 + blevel(C) = 2 + 17 = 19 < WSL(cls(A)) = 22$. Fig. 6(b) is the resultant state from the first clustering step in (a), where $cls(A)$ and $cls(C)$ are merged into the new cluster, i.e., $cls(A)$ (line 8 of Algorithm 1). In Fig. 6(b), the update process is performed as $top(A) = \{A\}$ and $out(A) = \{A, C\}$ by Update procedures in line 22. Then $WSL(cls(A))$ is

made smaller from 22 to 19. As $TL(B)$ is not affected by the clustering in (a), $TL(B)$ does not vary.

In Fig. 6(b), $cls(B)$ is selected as pivot from $FREE$ in step 2. Here D dominates $blevel(B)$, and (14) is satisfied by $cls(D)$, $cls(B)$, and $cls(D)$ are clustered. In line 8, $cls(D)$ are clustered, thus ($WSL(cls(B))$ is made smaller from 20 to 13 in Fig. 6(c). Then $cls(A)$ and $cls(E)$ are clustered in step 3, the result is shown in Fig. 6(d).

In Fig. 6(d), i.e., at step 4, pivot is $cls(A)$ and $cls(B)$ is selected as the target because $WSL(cls(A)) = 19$, that is larger than $WSL(cls(B)) = 13$ (lines 3 and 4 in Algorithm 1). However, the condition in (14) is not satisfied, i.e., $0 + (2 + 3 + 6 + 1 + 1 - 1) + blevel(D) = 12 + 12 = 24 > WSL(cls(A)) = 19$. Thus, the process goes to line 9, i.e., whether the condition in (15) is satisfied is checked. As $top(A) = \{A\}$ and A has no predecessor task, upward clustering is rejected and the process goes to line 14. In $A(cls(i)) = \emptyset$ and in line 15, the vCPU $c_{1,1,1}$ is selected for (16). Then the resultant state is shown in Fig. 6(e), where $cls(A)$ is still selected as pivot at step 5, but the process goes to line 13 $\rightarrow$ 16 $\rightarrow$ 20 because $A(cls(A)) \neq \emptyset$ in step 5. As $cls(B)$ has the next largest WSL in $FREE$, it is selected as pivot for step 6. Then, $cls(B)$ and $cls(F)$ are clustered and the resultant state is shown in Fig. 6(f).

In Fig. 6(f), i.e., at step 7, $cls(A)$ is selected as pivot and $cls(A)$ and $cls(G)$ are clustered, whose result is shown in (g).

At (g), the maximum WSL path is A$\rightarrow$C$\rightarrow$F$\rightarrow$G, and $cls(A)$ is selected as pivot. However, (15) is not satisfied and A has no predecessor task with $A(cls(A)) \neq \emptyset$. Thus, the process goes to line 21 and then $cls(B)$ is selected as pivot. As $cls(B)$ does not satisfy both (14) and (15), the process goes to line 15, and we have $A(cls(B)) = c_{1,1,2}$ for (16). As every task cluster is allocated to a vCPU, the process break the loop at line 2. □

4.3 Task ordering and actual vCPU allocation phase

In this phase, each task is scheduled by considering the makespan, the number of containers, and number of allocated vCPUs by satisfying Eq. (3). Notably, in line 23 of Algorithm 1, each task has been allocated to a vCPU. In this phase, it can be moved to another vCPU to minimize the makespan by reducing both the number of containers and vCPUs. Because each task is allocated to a vCPU by task clustering and pre-vCPU allocation in the previous section, the scheduling priority is based on actual vCPU allocation and not on the average performance as in conventional task scheduling algorithms [34,36]. As can be seen in line 23 of Algorithm 1, we define UEX_{task} and $fList$ for this second phase. This phase corresponds to lines 23–28 in Algorithm 1.

In line 25 of Algorithm 1, the task $v_i \in fList$ that satisfies the following condition is selected from $fList$:

$$tlevel(v_i) + blevel(v_i) = \max_{v_k \in fList} \{tlevel(v_k) + blevel(v_k)\}. \tag{18}$$

Algorithm 2: RefinedAllocation algorithm

Input: $v_i \in V$.
Output: The allocated vCPU, i.e., $A(v_i)$.
1 Let $\Psi(v_i) = \{v_j | v_j$ *belongs to the same container as* $v_i.\}$.
/* From the set of already allocated tasks, find the subset $\Psi(v_i)$ that belong to the same container as v_i (i.e., the same executable module). */
2 **if** $\Psi(v_i) \neq \emptyset$ **then**
/* Find the set of vCPUs, where they have tasks that belong to the same container as v_i. */
3 $C_{vcpu}(v_i) \leftarrow$ the set of vCPUs to which $\exists v_j \in \Psi(v_i)$ is allocated.
/* Allocate v_i to an idle time slot of a vCPU by insertion policy. Note that no container download is required and $T_{DL}(v_i, m_{p,q}) = 0$. */
4 $A(v_i) \leftarrow c_{k,l,m} \in C_{vcpu}(v_i)$ *s.t.,* (3) is satisfied and $T_f(v_i, c_{k,l,m}) = \min_{c_{p,q,r} \in C_{vcpu}(v_i)} \{T_s(v_i, c_{p,q,r}) + T_p(v_i, c_{p,q,r})\}$.
5 **else**
/* If no same-type tasks have been allocated in this phase, it tries to find the vCPU by which the finish time of v_i is minimized by insertion policy. */
6 $A(v_i) \leftarrow c_{k,l,m} \in C_{vcpu}$ *s.t.,* (3) is satisfied and $T_f(v_i, c_{k,l,m}) = \min_{c_{p,q,r} \in C_{vcpu}} \{T_s(v_i, c_{p,q,r}) + T_{DL}(v_i, m_{p,q}) + T_p(v_i, c_{p,q,r})\}$.
/* Update the scheduling priority values of successor tasks. */
7 Update $tlevel(v_j)$, where $v_j \in suc(v_i)$.
8 **return** $A(v_i)$.

Notably, both $tlevel(v_i)$ and $blevel(v_i)$ are derived using the actually allocated vCPU, not the average processing speed or the average bandwidth. Thus, accurate scheduling priorities are used in this phase.

In line 26 of Algorithm 1, SF-CUV tries to find the new allocation target vCPU by calling Algorithm 2. In line 1 of Algorithm 2, the set of tasks that belong to the same container as v_i, is defined as $\Psi(v_i)$. If such tasks exist (line 2 of Algorithm 2), $C_{vcpu}(v_i)$ is the set of vCPUs to which a task with the same container as v_i is allocated (line 3 of Algorithm 2). Then, the vCPU by which the finish time of v_i is minimized in $C_{vcpu}(v_i)$ is selected as $A(v_i)$. In line 4 of Algorithm 2, each task is allocated by tracing already allocated vCPUs for every task in $\Psi(v_i)$. Note that no container downloading time is required in this case. This procedure aims to suppress the number of containers and to minimize the makespan by container sharing among tasks. In line 5 of Algorithm 2, when there are no tasks in $\Psi(v_i)$, the vCPU that minimizes the finish time of v_i in the entire set of vCPUs is selected as $A(v_i)$. In this case, the container downloading time is required when the finish time is derived at line 5. After v_i is allocated to an idle time slot of a vCPU, *tlevel* of successor tasks of v_i is updated for next task selection from $fList$. Then, in line 27 of Algorithm 1, both $fList$ and UEX_{task} are updated. Then tasks $v_j \in suc(v_i)$ are put into $fList$ if all their predecessor tasks have been scheduled.

Example 4 *Fig. 7 shows an example of the task ordering and actual vCPU allocation phase. Fig. 7(a) corresponds to the state of (h) in Fig. 6. From this state, in step 1 of (d), we select A from fList, and the tasks of the same type as A are* $\{B, E\}$. *As* $Psi(v_i) = \{B, E\} \neq \emptyset$, *the process goes to line 2 of Algorithm 2. Here, we suppose that* $U_{th}(c_{1,1}) = 60$, $U(c_{1,1,1}, A) = 40$, *and* $U(c_{1,1,2}, B) = 20$*; then, we have* $\frac{60}{\frac{1}{2}(40+20)} = 2$. *Thus, A is scheduled as* $c_{1,1,1}$

in line 4 of Algorithm 2. Then, A is removed from fList and C is selected from fList in line 27 of Algorithm 1. Similarly, C, E, and B are assumed to be scheduled at $c_{1,1,1}$, $c_{1,1,1}$, *and* $c_{1,1,2}$ *by satisfying lines 3 and 4 of Algorithm 2, respectively. In step 5, when D is scheduled to* $c_{1,1,2}$, *SF-CUV attempts to allocate D to an idle time slot of* $c_{1,1,2}$. *Here, in (b), we assume that* $U(c_{1,1,1}, C) = 80$, $U(c_{1,1,2}, D) = 60$, *and* $U_{th}(c_{1,1}) = 60$. *In Eq. (3), we have* $\frac{80+60}{2} = 70 > U_{th}(c_{1,1}) = 60$. *Thus, D must be allocated to another vCPU in* n_1. *The alternative vCPU by which the finish time of D is minimized is* $c_{1,2,1}$ *because we assume that* $m_{1,2}$ *has the same container as the one that D belongs to. In such a case, no container downloading is required. Then, D is moved from* $c_1, 1, 2$ *to* $c_{1,2,1}$. *In this manner, every task is re-allocated to each vCPU.* □

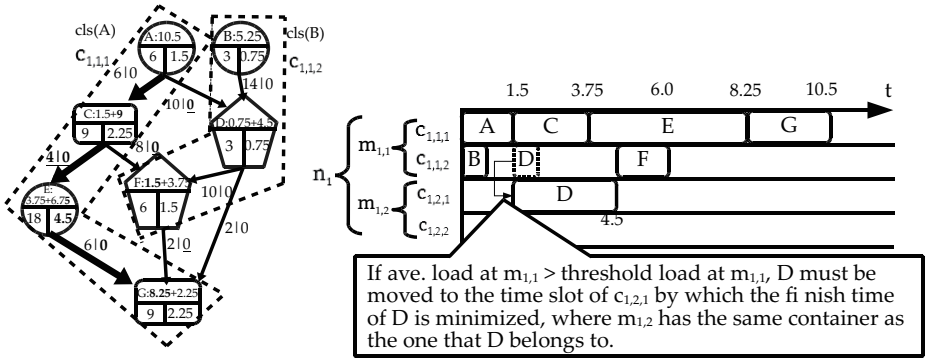
4.4 Time complexity of SF-CUV algorithm

To evaluate the time complexity of the SF-CUV algorithm, we start with the first phase, i.e., the task clustering and pre-vCPU allocation phase. In line 3 of Algorithm 1, $\log |FREE|$ steps are required for one pivot selection; thus this procedures costs $O(|V| \log |V|)$ in total. Once pivot is selected, the process goes to find a target for the clustering step in line 6 or 10. In line 6, target is the dominant task of $blevel(v_b)$ in (14), and it requires $|suc(v_b)| \log |suc(v_b)|$ steps by tracing every edge from v_b. As it requires one target task for each clustering step, a total of $|V|$ steps are required to trace all tasks. We have $\sum_{n_b \in V} |suc(n_b)| = |E|$ and $|suc(n_b)| \leq |V|$. Thus, line 6 costs $O(|E| \log |V|)$. Similarly, line 10 costs $O(|E| \log |V|)$. The dominant part for the time complexity in this phase is $Update(cls(i))$, as seen in lines 29–33 of Algorithm 1. In this part, all the edges and tasks in $cls(i)$ should be traced to update the information of $cls(i)$. This requires $(|V'| + |E'|)$ steps, where $|V'|$ and $|E'|$ are the number of tasks and edges within $cls(i)$, respectively. Then, the $blevel$ of each task in $out(i)$ must be updated; this requires $\log |suc(v_k)|$ steps, where $v_k \in out(i)$. Thus, in total, the time complexity of the first phase is $O(|V|(|V| + |E|) \log |E|)$.

For the second phase, i.e., the task ordering and actual vCPU allocation phase, the dominant part is as shown in line 26 of Algorithm 1, i.e., Algorithm 2. In line 3 of Algorithm 2, generating $C_{vcpu}(v_i)$ requires $|\Psi(v_i)|$ steps and the total time complexity is $O(|V|^2)$. In lines 4 and 6 of Algorithm 2, the time slot by which the makespan is minimized can be found by tracing each time slot for each vCPU. Thus, it requires $|C_{vcpu}(v_i)||V|$ steps and its time complexity is $O(|C_{vcpu}||V|^2)$.

Therefore, the time complexity of the SF-CUV algorithm is

$$\max\{O(|V|(|V| + |E|) \log |E|), O(|C_{vcpu}||V|^2)\}. \tag{19}$$



(a)State after the task clustering.

(b) Resultant Gantt chart by the task ordering phase.

Host	BW	VM : α	vCPU
n_1	$\beta_1 : 3$	$m_{1,1} : 4$	$c_{1,1,1}$
			$c_{1,1,2}$
		$m_{1,2} : 1$	$c_{1,2,1}$
			$c_{1,2,2}$
n_2	$\beta_2 : 1$	$m_{2,1} : 2$	$c_{2,1,1}$
			$c_{2,1,2}$
		$m_{2,2} : 5$	$c_{2,2,1}$
			$c_{2,2,2}$

(c) System spec.

Step	fList	Selected task	Allcation target
1	A, B	A	$c_{1,1,1}$
2	B, C	C	$c_{1,1,1}$
3	B, E	E	$c_{1,1,1}$
4	B	B	$c_{1,1,2}$
5	D	D	$\frac{U(C,c_{1,1,1})+U(D,c_{1,1,2})}{2} > U_{th}(m_{1,1})$ then move D from $c_{1,1,2}$ to $c_{1,2,1}$.
6	F	F	$c_{1,1,2}$
7	G	G	$c_{1,1,2}$

(d) Detailed information for task ordering steps.

Fig. 7 Example of task ordering and actual vCPU allocation.

5 Experimental results

In this section, we present and compare the results obtained via simulation and those obtained in a real environment. We compared the makespan, number of allocated computational resources, and number of containers to verify the effect of SF-CUV on resource utilization.

5.1 Relationship of objectives and evaluation items

Table 2 indicates the relationship between objectives of SF-CUV and evaluation items. SF-CUV attempts to minimize the makespan by the combination of task clustering and pre-vCPU allocation phase (i) and task ordering by the scheduling priority phase (ii). In the experimental comparison investigated in the simulation, Section 5.2.2 corresponds to the makespan minimization. In the real environment, Sections 5.3.6 and 5.3.7 correspond to it.

In phase (i) of the SF-CUV algorithm, the number of required vCPUs is suppressed by each clustering step; that is, the resultant set of vCPUs is that which minimizes WSL. Thus, the evaluation point is how each vCPU is

Table 2 Mapping of objectives and comparison items.

Objectives in SF-CUV	Phase	Simulation	Real environment
SL minimization	(i), (ii)	Sec. 5.2.2	Sec. 5.3.6, 5.3.7
vCPU utilization	(i), (ii)	Sec. 5.2.3	Sec. 5.3.6, 5.3.7
container utilization	(ii)	Sec. 5.2.3	Sec. 5.3.5, 5.3.6,

effectively utilized to minimize the makespan, i.e., the degree of contribution of each vCPU to minimizing the makespan, which is presented in Sections 5.2.3, 5.3.6, and 5.3.7.

In phase (ii), SF-CUV attempts to reallocate a task to a vCPU with the same type, but only if the makespan can be made smaller. Thus, the evaluation point is how effectively each task is shared to minimize the makespan. The corresponding parts are described in Sections 5.2.3, 5.3.5, and 5.3.6.

5.1.1 Comparison targets

Because the assumed workflow in this study consists of batch-type tasks, not all conventional approaches [5–15] can be compared with the SF-CUV algorithm. We categorized workflow algorithms for batch-type tasks as follows.

- SF-CUV without reallocation (CUV-FIX):
  In this approach, each allocated vCPU in the clustering phase is fixed in the task ordering phase. Thus, the allocation is completed in the clustering phase. This approach is required to confirm the effect of actual vCPU allocation in the second phase in SF-CUV.
- Capacity-based approach (CAP-based):
  In this approach, the order for task selection is based on the increasing order of the finish time with the average processing speed and the average communication bandwidth. Then, the selected task is allocated to the idle time slot of the vCPU with the largest residual capacity. Capacity in this simulation implies the clock frequency of each vCPU, i.e., the amount that a vCPU can process in a time unit. A vCPU having a higher processing speed has a higher task allocation priority than the nodes with lower processing speeds. The adoption of such capacity-based task allocation has been presented in the literature [6, 9].
- Communication locality-based approach (COM-based): This approach tries to minimize the communication time among tasks, i.e., the output data from one task is sent to the nearest node having sufficient capacity to accommodate the target task. Such data locality-based task allocation has been presented in the literature [5, 10].
- List-based task scheduling algorithm (HEFT [34] and PEFT [36]):
  Heterogeneous earliest finish time (HEFT) is a well-known task scheduling algorithm that is widely employed in real systems such as ASKALON [35]. Thus, in this comparison, we adopted HEFT as a criterion from the task scheduling perspective. Predict-EFT (PEFT) is a HEFT alternative that

outputs a better makespan than HEFT [36]. Thus, we adopted PEFT as a state-of-the-art task scheduling algorithm. The objective is to minimize the makespan, and no other criterion is considered. Both HEFT and PEFT use the average processing speed and the communication bandwidth for deriving the scheduling priority. If the constraint is given in Eq. (3) is considered, these task scheduling algorithms have no criteria for selecting the second allocation candidate that does not violate the constraint. Thus, in such a case, we modified these algorithms to allocate the selected task to the vCPU that does not violate Eq. (3) and achieves the second-lowest finish time. In this comparison, the reason for using them as a comparison target is to confirm the effect of step 1 of SF-CUV on the makespan, i.e., we confirm whether task clustering as pre-allocation to derive an accurate scheduling priority effect has a good effect on the makespan.

– Clustering-based task scheduling algorithm (CMWSL [21]):
  As mentioned in Section 2, CMWSL attempts to cluster several tasks until its cluster size exceeds the lower bound over the "single CPU" network. Although the assumed system is different from that of SF-CUV, we investigate how the degree of parallelism is degraded by imposing the lower bound in a cloud. As for other task clustering algorithms [25–27], the objective or the assumed environment is quite different; the algorithm [25] assumes a workflow having if/else edges, thus we cannot observe the effect through task occurrence-based clustering in the assumed system in this experiment. The objective of the algorithm in [26] is to minimize the cost, and the makespan can be made smaller by only the vertical clustering for data localization among sequential tasks. As such a data localization concept is included in the "COM-based" approach, we excluded the algorithm in [26] as a comparison target. The algorithm in [27] assumes only a homogeneous system and thus we excluded it from the comparison.
– A Docker container-based workflow scheduling(SOC: Stretch Out and Compact [24])
  In this algorithm, the critical path is derived by considering the container image download time, and then each task is allocated to vCPU. Then each allocated task that does not belong to the critical path is allocated in order not to affect the makespan.

5.2 Comparison in simulation

5.2.1 Simulation setup

In this subsection, we describe the simulation environment; the real environment is described in Section 5.3. We developed the simulator [38], where one or more workflow requests are processed among multiple cloud sites, and each cloud site has one or more nodes over the network. The i-th workflow request is denoted as $S_i = (V_i, E_i)$, and the entire workflow is denoted as

$$S = (V, E) = (V_1 \cup V_2 \cup \cdots \cup V_n, E_1 \cup E_2 \cup \cdots \cup E_n).$$

Table 3 Setup parameters of the simulation.

workflow(Random)	♯ of workflows	$\{1, 5, 10, 20\}$
	$\|V\|$ per workflow	$\{10, 30, 50, 100, 500\}$
	♯ of container types	$\{3, 5, 10, 20\}$
	Out degree of task	1 to 5
	Workload of task	1000 to 10000 (MI)
	Container image size	Randomly chosen in 50 – 1000 (MB)
	$\frac{\max_{d_{i,j} \in E} \{d_{i,j}\}}{\min_{d_{i,j} \in E} \{d_{i,j}\}}$	100
	CPU usage for each task	30% to 90%
workflow (Montage)	♯ of workflows	$\{1, 5, 10, 20\}$
	$\|V\|$ per workflow	$\{25, 50, 100, 1000\}$
	♯ of container types	$\{3, 5, 10, 20\}$
	Container image size	Randomly chosen in 50 – 1000 (MB)
	CPU usage for each task	10% to 90%
workflow (Epigenomics)	♯ of workflows	$\{1, 5, 10, 20\}$
	$\|V\|$ per workflow	$\{24, 46, 100, 997\}$
	♯ of container types	$\{3, 5, 10, 20\}$
	Container image size	Randomly chosen in 50 – 1000 (MB)
	CPU usage for each task	10% to 90%
System	♯ of cloud sites	5
	♯ of hosts per cloud	1 to 10
	♯ of cores per node	2 to 18
	♯ of vCPUs per core	2
	Max. usage per core	$\{70\%, 80\%, 90\%, 100\%\}$
	BW among clouds	$\{100, 300, 500, 1024\}$ (Mbps)
	BW within cloud	$\{1, 10\}$ (Gbps)
	Frequency at 1 vCPU	2.0 to 3.0 (GHz)

Then, S is the input for the simulation. Each task can perform various types of processing, e.g., encoding, rendering, and matrix operation, i.e., each task requires input data or files to output data or files as batch processing. In this context, we assume that several clients submit their own workflow requests. The simulation environment was developed and run via jdk1.8.0_191 on an Intel® Xeon® E-2176M CPU (2.70 GHz) with 32 GB of RAM.

Table 3 summarizes the setup parameters used in the simulation. We generated two types of workflows, i.e., randomly generated workflow (random workflow) to handle a general workflow structure, and real-world applications such as the Montage workflow and Epigenomics workflow expressed as a directed acyclic graph in XML (DAX) [37]. In particular, the Montage workflow is used for image transformation and image synthesis mainly for astronomical image processing. The Epigenomics workflow is used in various genome sequencing operations for genome analysis. Both workflows typically require large amounts of computational power and data exchanged among tasks, i.e., tasks. Thus, they are well suited for the simulation to verify the effectiveness in terms of resource utilization in clouds. The number of workflow requests is randomly chosen from $\{1, 5, 10, 20\}$ and the number of tasks in a random workflow request is randomly chosen from $\{10, 30, 50, 100, 500\}$, i.e., each workflow request initially consists of a workflow $S = (V, E)$, and multiple workflows are assumed to be scheduled simultaneously. In both the Montage workflow and the Epigenomics workflow, we used four patterns of DAX files for the simu-

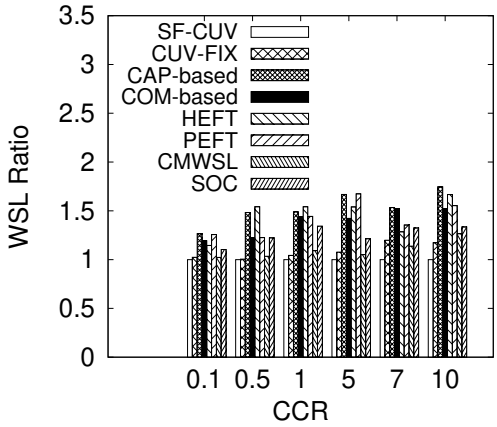
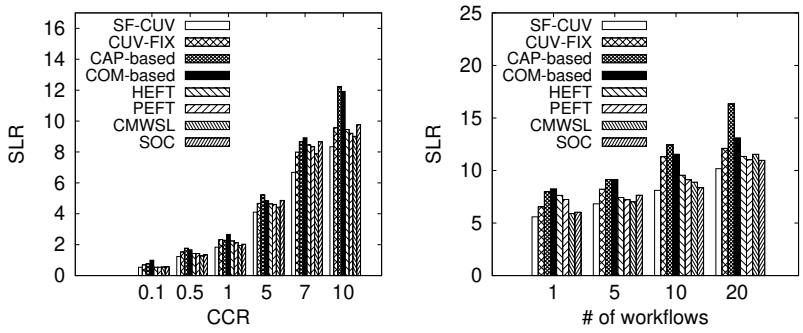


Fig. 8 Comparison of WSL.



(a) By varying CCR (b) By varying ♯ of workflows

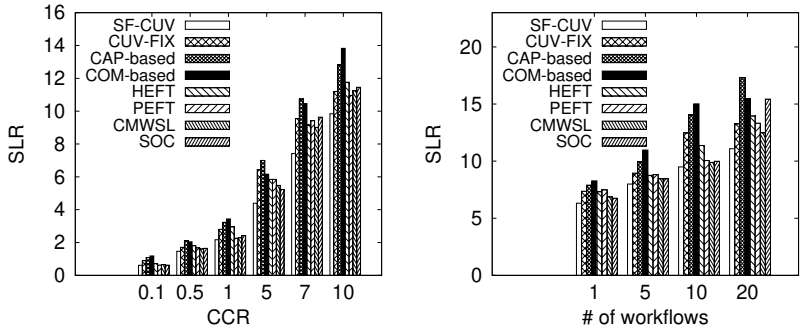
Fig. 9 Comparison of makespan for a random workflow.

lation. In this context, the simulation shows how each task in each workflow should be allocated and scheduled to share a function among workflows.

For the system parameters, we assume that there are 5 cloud sites, each of which has 1–10 computing hosts. Each host has 1 CPU socket, and the number of cores is randomly chosen from $\{2, 4, 6, \ldots, 18\}$. Then, each core has 2 vCPUs with hyper-threading and the ratio of the number of logical CPUs to the number of vCPUs is assumed to be 1.0 so that each vCPU provides the full capacity for each allocated task. Next, each task is assumed to be allocated to a vCPU. For the communication bandwidth, we assume that the bandwidth between clouds is smaller than that between hosts in a cloud.

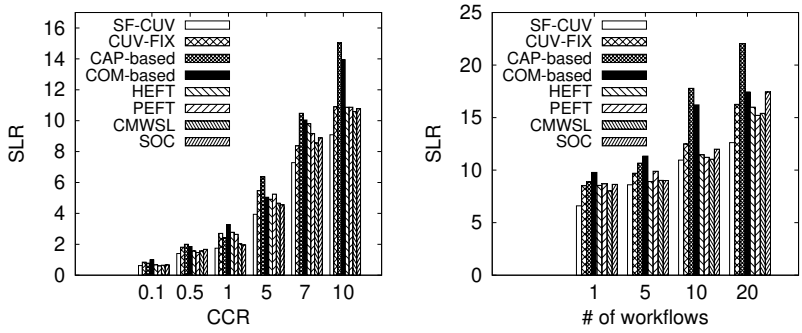
5.2.2 Comparison of makespan

In our experiments, we compared the makespan for varying communication-to-computation ratios (CCRs) [34,21]. We assumed $0 < CCR \leq 10$ to cover



(a) By varying CCR (b) By varying ♯ of workflows

Fig. 10 Comparison of makespan for the Montage workflow.



(a) By varying CCR (b) By varying ♯ of workflows

Fig. 11 Comparison of makespan for the Epigenomics workflow.

various applications. If the CCR of a workflow is high, the workflow is said to be data-intensive. As the clustering metric in SF-CUV is WSL to minimize the makespan after every task is scheduled, we first validated whether WSL in SF-CUV is smaller than in other approaches. Fig. 8 shows the comparison results of WSL in a random workflow, where the y-axis indicates the WSL ratio when SF-CUV is set to 1.0, and WSL is derived after every task is scheduled. WSL shown in this figure is based on the averaged WSL by 100 trials for each CCR. SF-CUV outperforms the other approaches in every CCR. In particular, from the comparison between WSL of SF-CUV and CUV-FIX, it is observed that WSL is improved from phase (i) to phase (ii) because the difference between them is whether phase (ii) is performed. For the comparison between SF-CUV and CMWSL, we conclude that imposing the lower bound by CMWSL does not always lead to WSL minimization on a cloud where multiple processing units are included in a node.

For the metric of the makespan for the comparison, we used the schedule length ratio (SLR) [34] to measure the performance of each task allocation

approach. The SLR is defined as follows:

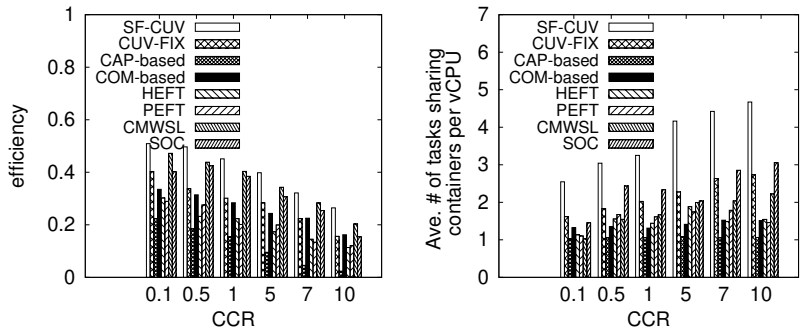
$$SLR = \frac{T_f(v_{end}, A(v_{end}))}{\sum_{v_k \in CP}\{\frac{D(v_k)}{\beta_{ave}} + T_p(v_k, \alpha_{ave})\}}, \tag{20}$$

where α_{ave} is the average processing speed and β_{ave} is the average communication bandwidth. CP is the critical path in the initial state with the average processing speed and average communication bandwidth. Note that SLR at (20) includes the container downloading time because every task needs its container downloading before the actual processing. Figs. 9, 10, and 11 show the comparison results of the SLR of three types of workflows derived as the average value in 100 trials, where (a) shows the results obtained by varying CCR and (b) shows the results obtained by varying the number of workflows. We averaged SLR by 100 for each CCR in (a) and each number of workflows in (b). From Fig. 9(a), we can observe that SF-CUV outperforms all the other approaches in every CCR. By comparing SF-CUV and CUV-FIX, we can conclude that the allocation target should be determined in the set of vCPUs in the same node as the vCPU that was determined in the clustering phase. For the comparison with the conventional task scheduling algorithms, i.e., HEFT, PEFT, CMWSL and SOC, SF-CUV outperforms these approaches, whereas these approached outperform CUV-FIX when CCR ranges from 0.1 to 5.0. In Fig. 9(b), we observe that SLR increases with the number of workflows in all the approaches. This is because a higher number of workflows means a larger-scale problem, although the dominant path length such as the critical path depends on each workflow. The result is that the task with the highest scheduling priority must be selected from the many free tasks. Thus, the accuracy in terms of the scheduling priority becomes crucial to the makespan. Therefore, the average-based scheduling priority derived by HEFT and PEFT tends to be degraded with a higher number of workflows. SF-CUV outperforms all the other approaches in all cases, whereas SLR by CUV-FIX is worse than SF-CUV owing to inaccurate actual task allocation. In Fig. 10(a) and (b), SLR by SF-CUV outperforms the other approaches in every CCR. Also, SLR by SF-CUV is the best among all the approaches in Fig. 11(a) and (b).

Thus, from these results, we can conclude that (i) task clustering itself does not always derive a good makespan in a computation-intensive workflow, and (ii) an accurate scheduling priority is necessary for a data-intensive workflow. Therefore, an task scheduling method that adopts a clustering policy and an accurate scheduling policy outputs a good makespan in every CCR. As for CMWSL, we cannot observe a positive effect by imposing the lower bound in a cloud, where multiple VMs belong to a node. From the obtained results, we can conclude that SF-CUV outputs a good makespan even if CCR becomes higher or the problem scale becomes larger.

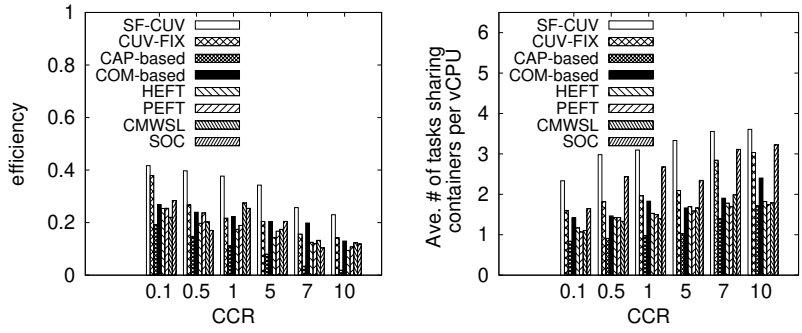
5.2.3 Comparison of resource utilization

To compare resource utilization, we used two measures: (i) efficiency [21] and (ii) average number of tasks of the same type per vCPU. Efficiency is the



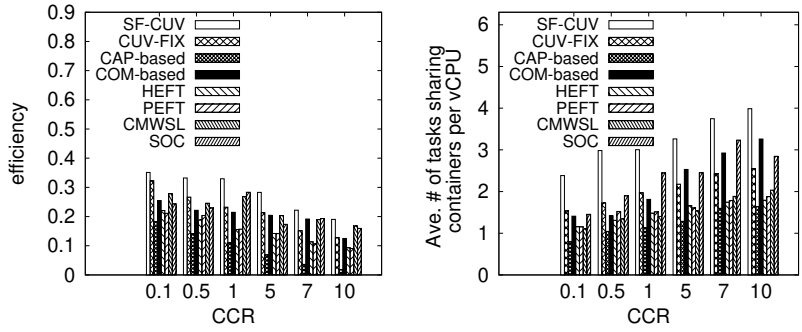
(a) Efficiency (b) Degree of container sharing

Fig. 12 Comparison in terms of resource utilization for a random workflow.



(a) Efficiency (b) Degree of container sharing

Fig. 13 Comparison in terms of resource utilization for the Montage workflow.



(a) Efficiency (b) Degree of container sharing

Fig. 14 Comparison in terms of resource utilization for the Epigenomics workflow.

degree of speedup ratio per vCPU, i.e., how each vCPU is utilized, which is defined as follows:

$$Efficiency = \frac{\sum_{v_i \in V} T_p(w_i, \alpha_{ave})}{|C'_{vcpu}| \times T_f(v_{end}, A(v_{end}))}, \tag{21}$$

where C'_{vcpu} is the set of actually allocated vCPUs. For (ii), the measure is defined as

$$\frac{|V|}{\sum_{c_{k,l,m} \in C_{vcpu}'} N_{type}(c_{k,l,m})}, \tag{22}$$

where $N_{type}(c_{k,l,m})$ is the number of tasks that shares containers in the vCPU $c_{k,l,m}$, that we call the degree of container sharing.

Figs. 12, 13, and 14 show the comparison results in terms of (a) efficiency and (b) degree of container sharing. In Fig. 12 (a), SF-CUV has the highest efficiency for all CCRs, whereas the CAP-based method has the lowest efficiency. This is because the CAP-based approach tries to find a vCPU having the highest residual capacity. Thus, one of the unallocated vCPUs can be an allocation candidate and the number of allocated vCPUs increases. For CUV-FIX, the number of allocated vCPUs is the same as that in the clustering phase, whereas SF-CUV tries to move an task to another vCPU while considering the minimum finish time in the task ordering and actual vCPU allocation phase. As for HEFT and PEFT, they derive a better makespan than CUV-FIX when CCR ranges from 0.1 to 5.0 in Fig. 9(a), but they do not derive better efficiency in every CCR in Fig. 12(a). One of the reasons for the inefficiency is that both try to allocate an task to an idle time slot of a node having no task to achieve a good makespan. As a result, many nodes can become task allocation targets. CMWSL outperforms both HEFT and PEFT in efficiency because more tasks are allocated to a vCPU by imposing the lower bound; that is, the required number of vCPUs in CMWSL is smaller than in HEFT and PEFT. In Fig. 12(b), we can see that SF-CUV has the highest degree of container sharing for all CCRs, then SOC has the next-highest value because it tries to localize the container downloading time by using the "stretch out" based critical path. In both the CAP-based and the COM-based approaches, the degree of container sharing is lower than that in CUV-FIX in every CCR. As for HEFT, PEFT, and CMWSL, as they do not consider the number of sharing tasks, their degree of container sharing is lower than that of CUV-FIX in every CCR. Thus, each task can be effectively shared in SF-CUV and the number of containers can be decreased compared with the other approaches. In Fig. 13(a) and (b), SF-CUV outperforms the other approaches in every CCR, and the CUV-FIX and COM-based approaches obtain better results than the COM-based, HEFT, and PEFT approaches. SOC has the next-highest degree of container sharing in Fig. 13(b). In Fig. 14(a), SF-CUV outperforms the other approaches in terms of efficiency. In Fig. 14(b), the COM-based approach obtains a better degree of container sharing than CUV-FIX when CCR is 5.0 or higher. Thus, the data-locality-based task allocation scheme can contribute to improved resource utilization.

In this comparison, we can conclude that an task reallocation scheme such as SF-CUV leads to a suppression in the number of allocated vCPUs and higher container sharing. Thus, using only a clustering phase cannot always suppress the number of allocated vCPUs, which depends on the clustering policy as well as the performance of each computational resource.

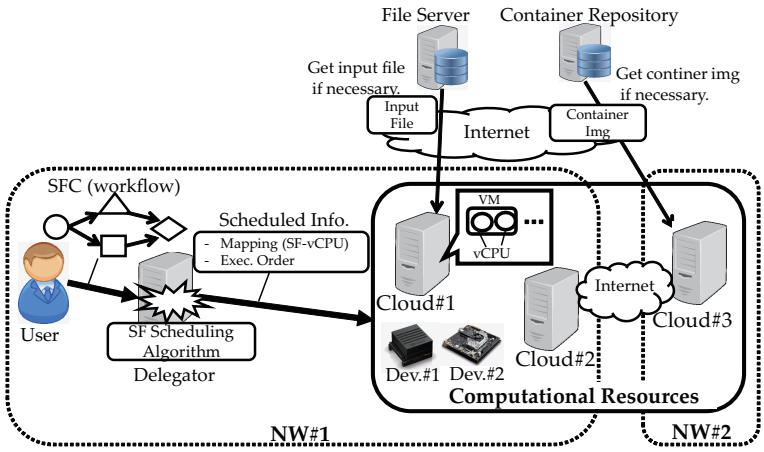
5.3 Comparison in a real environment

5.3.1 Objective

We implemented a workflow engine to incorporate task scheduling algorithms including SF-CUV for performance verification in a real environment. We assume that each task is performed in a heterogeneous distributed environment, where each node has different configurations, such as cloud-enabled and edge devices for processing data from IoT devices in a batch-processing manner. In this context, we conducted performance comparisons in terms of the makespan and resource utilization to verify the practicality of SF-CUV. If the degree of container sharing is higher, the number of sharing tasks becomes lower. Thus, the makespan can be made smaller. Therefore, we compared these metrics in two scenarios, i.e., "No task pre-deployment" presented in Section 5.3.6, where every computational resource has no task (container) before scheduling the tasks, and "task pre-deployment" presented in Section 5.3.7, where an task has been deployed on every computational resource before scheduling the tasks to verify the effect of container sharing in SF-CUV.

5.3.2 Real environment setup

Fig. 15 shows the environment in which we conducted a performance comparison for a real situation. In this environment, we set up a heterogeneous computing environment in two different networks, i.e., NW♯1 and NW♯2. In these networks, we deployed computer hosts on which VMs run through Apache CloudStack [40]. Each VM works on Ubuntu Desktop 18.04.3 LTS through KVM hypervisor. Then our developed workflow engine named SFlow [39] was installed on every VM. Although the mapping between each CPU core and each vCPU is typically controlled by a hypervisor, in this experiment, we manually mapped each CPU core and each vCPU by CPU pinning in advance. Then "taskset" command is set to run the specific vCPU for the allocated task. We assume that each task is allocated to each vCPU by an task scheduling algorithm. According to the table shown in Fig. 15, Cloud♯1 and Cloud♯2 in NW♯1 and Cloud♯3 in NW♯2 have 15 VMs, 6 VMs, and 8 VMs, respectively. Moreover, the bottleneck network bandwidth was set as 100 Mbps by deploying a router having 100 Mbps NIC between NW♯1 and NW♯2. In NW♯1, there are two computing devices denoted as Dev.♯1 and Dev.♯2, on which the container can be deployed while no VMs are deployed.



Hardware	Parameter	Value
NW#1 Router#1	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#1	# of VMs	15VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 11VMs
	vCPU Freq.	{2.0, 2.5, 3.0} GHz.
	BW to Router #1	1Gbps
Cloud#2	# of VMs	6VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 2VMs
	vCPU Freq.	{1.0,2.0, 2.5} GHz.
	BW to Router #1	1Gbps
Dev.#1	Type	NVIDIA Jetson AGX Xavier (2.26 GHz x 8Cores, 16GB RAM)
Dev.#2	Type	NVIDIA Jetson TX2 (2.0 GHz x 6Cores, 8GB RAM)

Hardware	Parameter	Value
NW#2 Router#2	Bottleneck BW to external hosts	100Mbps.
	Internal BW	1Gbps
Cloud#3	# of VMs	8VMs
	# of vCPUs	2vCPUs x 4VMs 4vCPUs x 4VMs
	vCPU Freq.	{1.0, 3.0, 3.5} GHz.
	BW to Router #2	1Gbps

Fig. 15 Real environment for the experiment.

As for the processing speed for each processor, we set it from 2.0 to 3.0 GHz. The maximum usage rate for each vCPU was set as 80% to avoid exceeding 100%, because, typically, user processes and kernel processes continuously occupy each CPU usage to a certain degree.

In this environment, we assume that one task corresponds to one Docker container [41] that is stored in the "Container Repository" in Fig. 15. Thus, if a node has no task for execution, it downloads the task from the Container Repository by SCP and then begins execution. If the START task requires one input file and the target node of the START task has no input file, it downloads the input file from the "File Server" by SCP. Thus, we assume that the file transfer involving the input file and Docker container may occur.

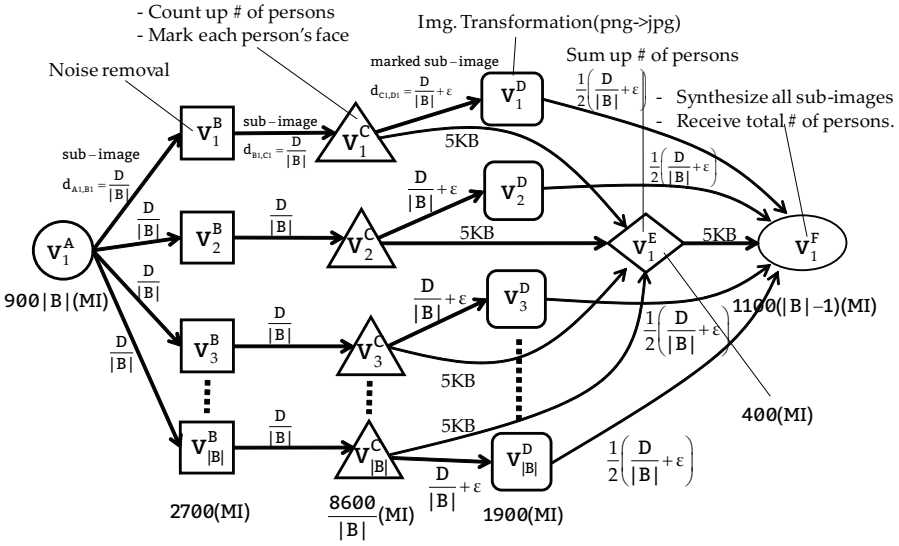


Fig. 16 Applied workflow structure.

5.3.3 Applied workflow for a real environment

In this real environment, we assume a workflow in which a large input file is processed in parallel among the networks, e.g., an image file is analyzed for IoT data processing. In particular, from an image captured by a camera, the number of faces (hereinafter, "faces") is counted to specify the degree of congestion in a specific purpose.

Fig. 16 shows the applied workflow in the real environment. The workflow has six types of task, i.e., A, B, C, D, E, and F, and every task is executed on a Docker in which OpenCV [42] is pre-installed. Then, we generated the OpenCV-enabled Docker image and compressed it as a 1.22 GB tar file that we name "BaseSF-tar," and it was stored in the "Container Repository" as in Fig. 15. As the workflow has six types of task, we generated six different Docker images based on "BaseSF-tar," i.e., every task Docker image has the same file size (1.22 GB).

In the workflow, v_1^A divides the input image into $|B|$ sub-images, where $|B|$ is a variable in the experiment; and we used the 5.45 MB (small image) and 61.7MB (large image) files as the input file. At v_k^B, the divided sub-image is processed for noise removal, and the output data size is set as $\frac{D}{|B|}$ in the configuration file for workflow submitted to "Delegator" in Fig. 15. Then, v_k^C processes the count of the number of faces and marks each face in the sub-image using "CascadeClassifier." For the data from v_k^C to v_k^D, its size is assumed to be added with ϵ (MB) that corresponds to the size of marked parts in the sub-image. As ϵ depends on the number of marked parts (i.e., the number of faces) and the marking structure, we cannot know it in advance.

Thus, in this experiment, we set ϵ to zero, i.e., the effect of marking on the resultant data size is negligible. At the same time, the counted number is sent from v_k^C to v_1^E, and every sub-image is assumed to be half of the data sent from v_k^C to v_k^D according to our measurement of their file sizes in the image format transformation (png $\rightarrow$ jpg) in advance. Then, each output sub-image is merged at v_1^F. The data sent from v_k^C to v_k^D includes a text message denoting the number of persons and additional information regarding the java object for object serialization. According to our measurement in advance, its size is 5 kB. In Fig. 16, the number of tasks in the workflow is defined as $1 + |B| + |B| + |B| + 1 + 1 = 3|B| + 3$.

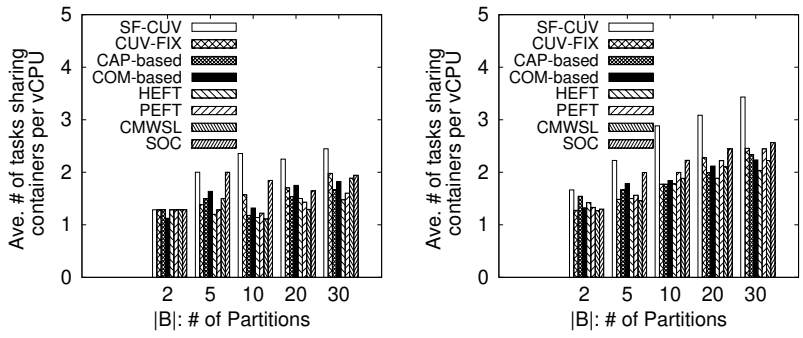
We measured the workload for each type of task by a test run on a 2.0 GHz vCPU; then, we set each workload from the time duration by the test run. If an task takes the time duration t, we define the workload as $1000t$. In Fig. 16, workloads of $v_1^A, v_k^B, v_k^C, v_k^D, v_1^E, v_1^F$ are defined in the unit of MI (million instructions).

5.3.4 Procedures

As for the processing flow, first, a user sends workflow information (workflow info.) as a JSON file to the "Delegator" that is in charge of performing task scheduling using the environmental information (Env. info) that includes every host information (IP address for each node, the clock frequency for each processor, a threshold value of the load for each CPU core, and communication bandwidth). Then, the Delegator derives the mapping and execution order as the scheduling result. Then, it sends the request for execution of the START task to the node that is specified by the scheduling result. At the START task, we assume that no input file has been deployed on the node that executes START task before the execution because there is no way to know where the input file should be located before scheduling the task. Thus, in every execution, the input file is first downloaded from the "File Server" in Fig. 15 to the node for executing the START task. The following executions are performed according to the scheduling result with the socket communication. When the execution of the END task is completed, its result is returned to the Delegator. The makespan in this case is defined as the time duration from the start time for downloading the input file to the finish time of the resultant data arrival at the user.

5.3.5 Comparison results of container sharing

Figs. 17(a) and (b) shows the comparison results of the degree of container sharing in the real environment. As the number of actually allocated vCPUs is derived after each task scheduling (i.e., the value is derived at "Delegator" in SFlow [39] in Fig. 15), it is the same in the cases of both no task pre-deployment and task pre-deployment. Fig. 17(a) is the result for the small input image size (5.45MB) and (b) is the result for the large input image size (61.7MB). In both figures, the horizontal axis represents the number of



(a) Degree of container sharing (small image)

(b) Degree of container sharing (large image)

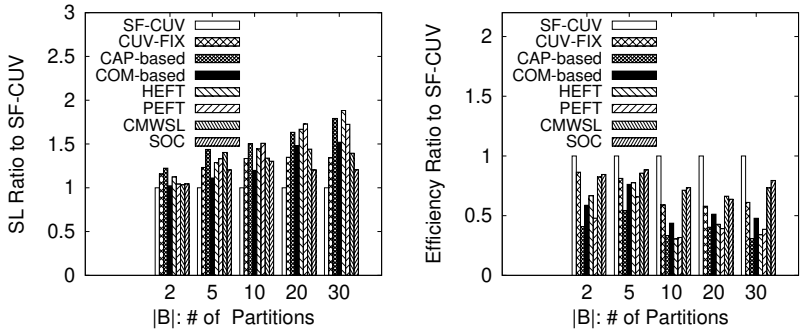
Fig. 17 Comparisons degree of container sharing.

partitions, i.e., $|B|$. In Fig. 17(a), at $|B| = 2$, the degree of container sharing is nearly the same among the algorithms. The reason is that the number of tasks ($|V| = 3(|B| + 1) = 9$ is not sufficient to provide the difference in the degree of container sharing, i.e., no task has not been moved in the second phase in SF-CUV, i.e., the "task ordering and actual vCPU allocation" phase. If $|B|$ is larger (5, 10, 20, 30), the difference increases and SF-CUV outperforms the other approaches in terms of the degree of container sharing. In particular, the difference between SF-CUV and CUV-FIX increases, i.e., more tasks have been moved to be shared in the second phase compared with the case of $|B| = 2$. We can see that SOC outperforms CAP-based, HEFT, PEFT, and CMWSL in every case. From this result, SF-CUV can effectively share tasks when $|B|$ is larger than 5. For Fig. 17(b), the difference in terms of the degree of container sharing is higher than that of (a), even at $|B| = 2$. The reason is that each data size to be exchanged among tasks is larger than the case of a small image, thereby more communications, i.e., tasks are localized in the same vCPU or a VM by SF-CUV. As a result, more containers are shared among one VM than the case of (a).

5.3.6 Comparison results in no task pre-deployment

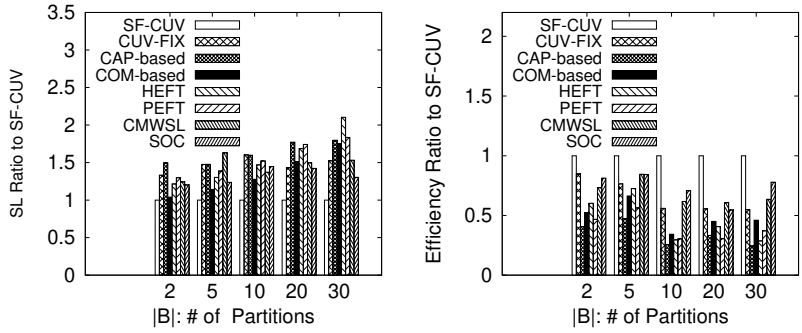
In this experiment, we conducted performance comparisons in terms of the makespan, efficiency defined in Eq. (21), and the degree of container sharing defined in Eq. (22). The actual performance depends on the dynamics of the nodes, such as fluctuation of computational and network load as well as on the static performance, such as CPU frequency and network bandwidth. Thus, we averaged these metrics over 10 trials for each algorithm for comparison.

Fig. 18 shows the comparison results in the case of no task pre-deployment, where Fig. 18(a) shows the comparison results for the schedule length ratio when SF-CUV is set to 1.0, and Fig. 18(b) shows the results for the efficiency ratio when SF-CUV is set to 1.0. In (a) and (b), in all cases of $|B|$, SF-CUV



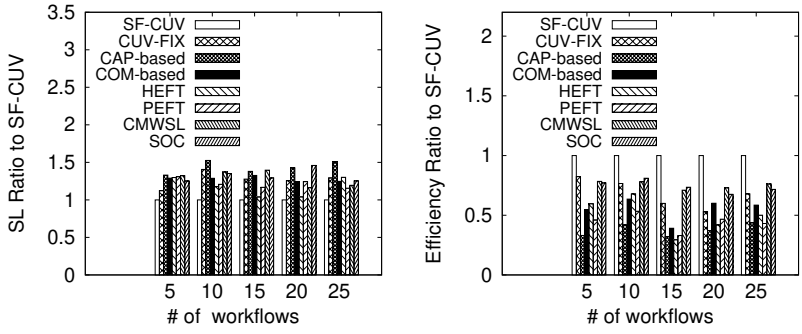
(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 18 Comparisons of no task pre-deployment for the small image file.



(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 19 Comparisons of no task pre-deployment for the large image file.



(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 20 Comparisons of no task pre-deployment for multiple workflows.

outperforms the other approaches in terms of the makespan and efficiency. In particular, in Fig. 17, the degrees of container sharing in HEFT and PEFT are worse than in the COM-based approach, and in Fig. 18(a) their makespans are also worse. Thus, their efficiencies are worse than in the COM-based approach, except in the case of $|B| = 5$. This is because more tasks (i.e., containers) should be downloaded from the Container Repository in HEFT and PEFT than in the COM-based approach. Thus, the downloading time becomes the bottleneck of the makespan. As for CMWSL, its makespan is better than that of the COM-based approach in Fig. 18(a), and efficiency in CMWSL is better than the COM-based approach in Fig. 18(b). The reason is that the required number of vCPUs is more suppressed by clustering with the lower bound than with the COM-based approach; thus, efficiency in CMWSL is higher than that in the COM-based approach.

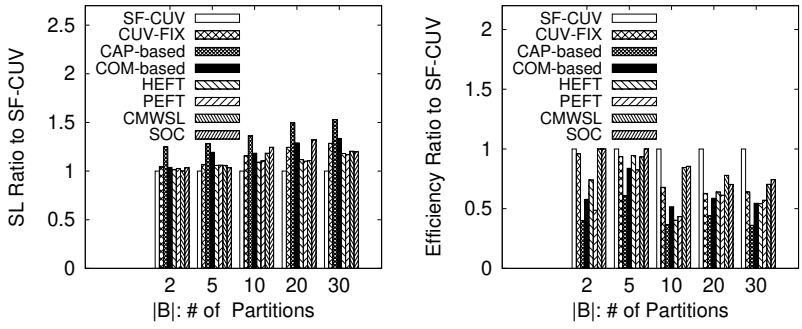
Fig. 19 shows the comparison results for the large image file (61.7MB), and it is observed that SF-CUV outperforms others in terms of the makespan and efficiency from (a) and (b). If the input image file size is large, the data size exchanged among tasks is larger than the case of small image file. As a result, such large data communications are localized within one vCPU or one VM by SF-CUV and both the makespan and efficiency are not degraded.

Fig. 20 shows the comparison results for multiple workflows, where mutiple image files (each file size is 61.7MB) are handled as input files. For each workflow, we uniformly distributed $|B|$ to make the application have variation in terms of the data size, i.e., if the number of workflows is $|N|$, each case of $|B|$ is $\frac{N}{5}$. For example, if $N = 10$, each two workflows try to divide the image file into $|B| = 2, 5, 10, 20, 30$ partitions. Then such N workflows are handled as the merged workflow for scheduling each task. In both results at (a) and (b) in Fig. 20, we can see that SF-CUV outperforms others in terms of the makespan and efficiency even if multiple workflows are scheduled simultaneously.

From the obtained results, it can be concluded that the degree of container sharing can strongly affect the makespan in the case of no task pre-deployment, and an algorithm must consider how each task should be shared among the computational resources.

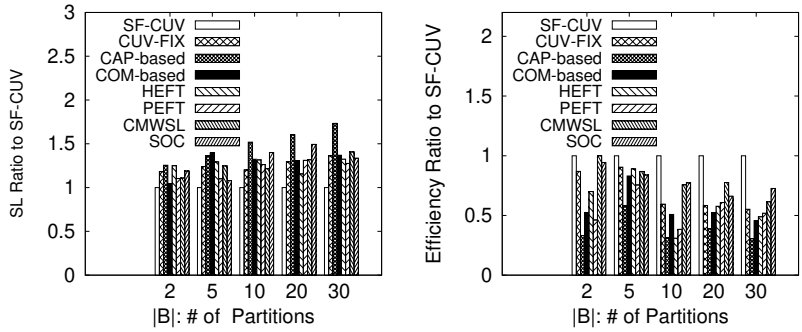
5.3.7 Comparison results in task pre-deployment

Fig. 21 shows the comparison results in terms of the makespan and efficiency in the case of task pre-deployment. In this case, every type of task is deployed before scheduling tasks, i.e., no task downloading is required. Thus, this case corresponds to the ideal situation in which we can obtain information about where and which task should be executed in advance. SF-CUV outperforms the others in Fig. 21(a) and (b). In contrast to Fig. 18(a), in Fig. 21(a), HEFT and PEFT obtain better makespans than the COM-based approach for each number of partitions. In Fig. 21(b), we cannot observe superiority in terms of efficiency among the COM-based, HEFT, PEFT, and CMWSL approaches. Therefore, we conclude that a list-based task scheduling algorithm can be applied to a workflow for minimizing the makespan if every computational



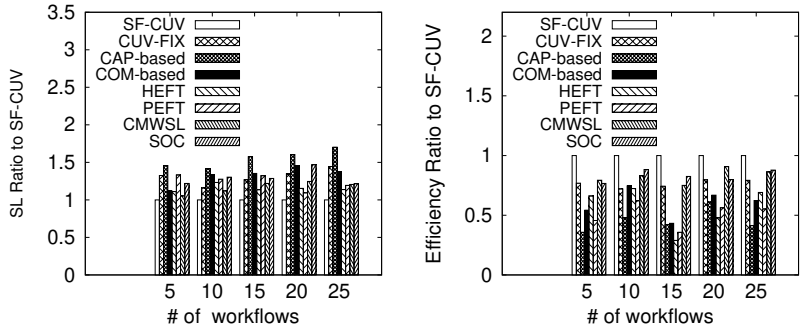
(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 21 Comparisons of task pre-deployment for the small image file.



(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 22 Comparisons of task pre-deployment for the large image file.



(a) Schedule Length Ratio (b) Efficiency Ratio

Fig. 23 Comparisons of task pre-deployment for multiple workflows.

resource is ready for executing the tasks; however, they still do not achieve good efficiency. SF-CUV satisfies the requirement of makespan minimization with a small number of computational resources with and without task predeployment.

Fig. 22 shows the comparison results for the large image file (61.7MB), and Fig. 23 shows the comparison results for multiple workflows. In both cases, we can see that SF-CUV outperforms others in terms of the makespan and efficiency. In particular, at Fig. 23(b) we can see the the difference of efficiency between SF-CUV, CMWSL, and SOC become smaller with more workflows. Since the total number of tasks becomes larger with increasing of the number of workflows, thereby every vCPU is allocated to tasks in all algorithms. As a result, efficiency depends on only the scheduling policy. This fact affects on the behavior of Fig. 23(b).

6 Conclusion

In this paper, we proposed an task clustering algorithm, namely SF-CUV, to minimize the makespan by effectively utilizing both vCPUs and containers. In the task clustering and pre-vCPU allocation phase, WSL was minimized to suppress the actual makespan in the task scheduling phase, and this phase derived an accurate scheduling priority. Also in this phase, the number of required vCPUs is effectively suppressed by clustering steps. In the task ordering and actual task reallocation phase, each task was scheduled with the accurate scheduling priority and moved to another vCPU for sharing containers among tasks for each vCPU to avoid redundant container downloading procedures. Both the simulation results and the real environmental results showed that SF-CUV outperforms other task scheduling algorithms, i.e., it provides a better makespan while effectively utilizing the vCPUs and containers. Furthermore, we explored a method for dynamically determining the CPU usage threshold for handling actual system dynamics. In the future, more realistic requirements, such as memory/hard disk capacities, should be considered for incorporating SF-CUV into a real environment.

Acknowledgement

Firstly, we would like to appreciate anonymous reviewers to their useful and valuable comments. This work is partially supported by the R&D contract for radio resource enhancement “Wired-and-Wireless Converged Radio Access Network for Massive IoT Traffic” by the Ministry of Internal Affairs and Communications, Japan. The research leading to these results has been supported by the EU-JAPAN initiative by the EC Horizon 2020 Work Programme (2018-2020) Grant Agreement No.814918 and Ministry of Internal Affairs and Communications “Strategic Information and Communications R&D Promotion Programme (SCOPE)” Grant no. JPJ000595, “Federating IoT and cloud

infrastructures to provide scalable and interoperable Smart Cities applications, by introducing novel IoT virtualization technologies (Fed4IoT)".This research is partially supported by Japan's Ministry of Internal Affairs and Communications and JSPS KAKENHI Grant Number 19K11910.

Compliance with ethical standards

The authors declared that they have no conflicts of interest to this work. We declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

References

1. R. Mijumbi, J. Serrat, J-L. Gorricho et al., "Network Function Virtualization: State-of-the-Art and Research Challenges," *IEEE Commun. Surv. & Tutorials*, vol. 18, no. 1, pp. 236–262, Sep., 2016.
2. A. Belbekkouche, M.M. Hasan, and A. Karmouch, "Resource Discovery and Allocation in Network Virtualization," *IEEE Commun. Surv. & Tutorials*, vol. 14, no. 4, pp. 1114–1128, Feb., 2012.
3. D. Bhamare, R. Jain, M. Samaka, et al., "A survey on service function chaining," *J. Netw. Comput. Appli.*, Vol. 75, pp. 138–155, Sep., 2016.
4. P. Quinn and J. Guichard, "Service Function Chaining: Creating a Service Plane via Network Service Headers," *Computer*, vol. 47, no. 11, pp. 38–44, Nov. 2014.
5. L. Wang, Z. Lu, X. Wen, et al., "Joint Optimization of Service Function Chaining and Resource Allocation in Network Function Virtualization," *IEEE Access*, vol. 4, pp. 8084–8094, Nov., 2016.
6. M. C. Luizelli, W. L. C. Cordeiro, L. S. Buriol, et al., "A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining," *Comput. Commun.*, vol. 102, 67–77, Nov., 2016.
7. T-W. Kuo, B-H. Liou, K. C-J Lin, et al., "Deploying Chains of Virtual Network Functions: On the Relation Between Link and Server Usage," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1562–1576, Aug., 2018.
8. M. Ghaznavi, N. Shahriar, S. Kamali, et al., "Distributed Service Function Chaining," *IEEE J. Sel. Areas Commun.*, vol. 35, no. 11. pp. 2479–2489, Nov., 2017.
9. D. Bhamare, M. Samaka, A. Erbad, et al., "Multi-Objective Scheduling of Micro-Services for Optimal Service Function Chains," *in Proc. IEEE ICC 2017 SAC Symp. Cloud Commun. Netw. Track*, pp. 1–6, May, 2017.
10. M. T. Beck and J. F. Botero, "Scalable and coordinated allocation of service function chains," *Comput. Commun.*, vol. 102, pp. 72–88, Oct. 2016.
11. Y. Sang, B. Ji, G. R. Gupta, et al., "Provably Efficient Algorithms for Joint Placement and Allocation of Virtual Network Functions," *in Proc. IEEE Int. Conf. Comput. Commun. (IEEE INFOCOM 2017)*, pp. 1–9, May, 2017.
12. S. Sahhaf, W. Tavernier, M. Rost, et al., "Network service chaining with optimized network function embedding supporting service decompositions," *Comput. Netw.*, vol. 93, pp. 492–505, Oct., 2015.
13. H. A. Alameddine, S. Sebbah, and C. Assi, "On the Interplay Between Network Function Mapping and Scheduling in VNF-Based Networks: A Column Generation Approach," *IEEE Trans. Netw. Serv. Managem.*, vol. 14, no. 4, pp. 860–874, Dec., 2017.
14. Z. Li and Y. Yang, "Placement of Virtual Network Functions in Hybrid Data Center Networks," *in Proc. IEEE Int. Symp. High-Performance Interconnects*, pp. 73–79, Aug., 2017.
15. S. Khebbache, M. Hadji, and D. Zeghlache, "Virtualized network functions chaining and routing algorithms," *Comput. Netw.*, vol. 114, pp. 95–110, Jan., 2017.

16. V. Eramo, E. Miucci, M. Ammar et al., "An Approach for Service Function Chain Routing and Virtual Function Network Instance Migration in Network Function Virtualization Architectures," *IEEE/ACM Trans. Netw.*, Vol. 25, No. 4, pp. 2008–2025, Aug., 2017.
17. O. Soualah, M. Mechtri, C. Ghribi et al., "An Effcient Algorithm for Virtual Network Function Placement and Chaining," *in Proc. IEEE Annu. Consumer Commun. & Netw. Conf. (CCNC)*, pp. 647–652, Jan., 2017.
18. X. Lin, D. Guo, Y. Shen et al., "DAG-SFC: Minimize the Embedding Cost of SFC with Parallel VNFs," *in Proc. Int. Conf. Parallel Processing (ICPP 2018)*, 10 pages, Aug. 2018.
19. H. K. Nguyen, Y. Zhang, Z. Chang et al., "Parallel and Distributed Resource Allocation With Minimum Traffc Disruption for Network Virtualization," *IEEE Trans. Commun.*, Vol. 64, No. 3, pp. 1162–1175, Mar., 2017.
20. J. Pei, P. Hong, K. Xue, and D. Li, "Efficiently Embedding Service Function Chains with Dynamic Virtual Network Function Placement in Geo-Distributed Cloud System," *IEEE Trans. Parallel Distrib. Syst.*, Vol. 30, No. 18, pp. 2179–2192, Oct., 2019.
21. H. Kanemitsu, M. Hanada, and H. Nakazato, "Clustering-based Task Scheduling in a Large Number of Heterogeneous Processors," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 11, pp. 3144–3157, Nov., 2016.
22. G. Sun, Y. Li, et al., "Low-latency orchestration for workflow-oriented service function chain in edge computing," *Future Generation Computer Systems*, vol. 85, pp. 116–128, Aug., 2018.
23. E. J. Alzahrani, Z. Tari, Y. C. Lee, D. Alsadie, and A. Y. Zomaya, "AdCFS: Adaptive completely fair scheduling policy for containerised workflows systems," *in Proc. 2017 IEEE 16th Int. Symp. Network Comput. Appl., NCA 2017*, pp. 1 ?, 2017.
24. W. Zhang, Y. Liu, L. Wang, Z. Li, and R. S. M. Goh, "Cost-Efficient and Latency-Aware Workflow Scheduling Policy for Container-Based Systems," *in Proc. Int. Conf. Parallel Distrib. Syst., ICPADS2019*, pp. 763–770, 2019.
25. E. E. Mong, M. M. Thein, M. T. Aung, "Clustering Based on Task Dependency for Data-Intensive Workflow Scheduling Optimization," *in Proc. Workshop Many-Task Computing on Clouds, Grids, and Supercomputers*, pp. 20—25, Nov., 2016.
26. M. Dong, L. Fan, C. Jing, "ECOS: An Efficient Task-clustering based Cost-effective aware Scheduling Algorithm for Scientific Workflows Execution on Heterogeneous Cloud Systems," *J. Syst. Software.*, Vol. 158, pp. 1–11, Sep., 2019.
27. M. Y. Ozkaya, A. Benoit, B. U?ar, J. Herrmann, and ?mit V. Cataly?rek , "A Scalable Clustering-Based Task Scheduler for Homogeneous Processors Using DAG Partitioning," *in Proc. Prallel Distrib. Proc. Symp. (IPDPS)* pp. 155-165, May, 2019.
28. A. M. Chirkin et al., "Eecution time estimation for workflow scheduling," Futur. Gener. Comput. Syst., vol. 75, pp. 376–387, 2017.
29. Phinjaroenphan P., Bevinakoppa S., Zeephongsekul P. "A Method for Estimating the Execution Time of a Parallel Task on a Grid Node," Lecture Notes in Computer Science, Vol. 3470. pp. 226–236, Springer, 2005.
30. D. Koufaty and D. T. Marr, "Hyperthreading technology in the netburst microarchitecture," *IEEE Micro*, vol. 23, no. 2, pp. 56–65, Apr., 2003.
31. A. Podzimek, L. Bulej et al., "Analyzing the Impact of CPU Pinning and Partial CPU Loadson Performance and Energy Efficiency," *in Proc. IEEE/ACM Int. Symp. Cluster, Cloud and Grid Comput.*, pp. 1–10, July, 2015.
32. R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya," CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Software: Practice and Experience*, vol. 41, no. 1, pp. 23–50, Jan., 2011.
33. W. Chen and E. Deelman, "WorkflowSim: A toolkit for simulating scientific workflows in distributed environments," *in Proc. IEEE Int. Conf. E-Sci.*, pp. 1–8, Oct., 2012.
34. H. Topcuoglu, S. Hariri, and M. Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 260–274, 2002.
35. T. Fahringer, A. Jugravu et al, "ASKALON: A Tool Set for Cluster and Grid Computing," *Concurrency Computat.: Pract. Exper., Wiley*, Feb., 2005.

36. H. Arabnejad and J. G. Barbosa, "List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table," *IEEE Trans. Parallel Distrib. Syst.*, Vol. 25, No. 3, pp. 682–694, 2014.
37. S. Bharathi et al, "Characterization of scientific workflows," *in Proc. Third Workshop on Workflows in Support of Large-Scale Science*, pp. 1–10, 2008.
38. ncl_sfcsim: URL:`https://github.com/ncl-teu/ncl_sfcsim`
39. SFlow: URL:`https://github.com/ncl-teu/SFlow`
40. Apache CloudStack: URL: `https://cloudstack.apache.org/`
41. Docker Web Site: URL: `https://www.docker.com/`
42. OpenCV Web Site: URL: `https://opencv.org/`