Computer Networks 112 (2017) 345–359

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

GreenMap: Green mapping of MapReduce-based virtual networks onto a data center network and managing incast queueing delay

Ebrahim Ghazisaeedi*, Changcheng Huang

Department of Systems and Computer Engineering, Carleton University, Ottawa, ON, Canada

ARTICLE INFO

Article history:
Received 21 March 2016
Revised 25 September 2016
Accepted 18 November 2016
Available online 25 November 2016

Keywords:
Energy-efficient data centers
MapReduce
Virtualized data centers
Incast queueing delay

ABSTRACT

Energy consumption is a first-order concern for today's data centers. MapReduce is a cloud computing approach that is widely deployed in many data centers. Toward virtualizing data centers, we consider MapReduce-based virtual networks that need to be embedded onto a data center network. In this paper, we propose GreenMap, a novel energy-efficient embedding method that maps heterogeneous MapReduce-based virtual networks onto a heterogeneous data center network. Besides, we introduce a new incast problem that specially may happen in Virtualized Data Centers (VDCs). GreenMap also controls the incast queueing delay. We formulate a Mixed-Integer Disciplined Convex Program (MIDCP) for this method. Because the formulated MIDCP is $\mathcal{NP}$-hard, we also propose a novel and scalable heuristic for GreenMap. Simulation results prove that both of the MIDCP and the heuristic reduce a data center network's energy consumption effectively, and control the incast queueing delay.

1. Introduction

Cloud computing is becoming widespread, and energy consumption of the physical infrastructure that provides resources for the cloud is growing [1]. Accordingly, energy management is a key challenge for data centers to reduce all their energy-related costs [2].

MapReduce [3] is a cloud computing approach that parallelizes a computation across large-scale cluster of servers. We target MapReduce, because it is widely deployed in many data centers like Yahoo!, Amazon, and Facebook [4]. In this framework, users specify the computation tasks by generating map and reduce functions. Nodes that perform a mapping job are called mappers. Mappers process input data and generate intermediate key and value pairs. The generated key and value pairs are shuffled through shuffler nodes to the other nodes that perform a reducing job, called reducers. Reducers aggregate the received intermediate key and value pairs from different mappers and compute the computational results for an application. Note that the original MapReduce framework in [3] does not have a shuffling node. However, when we map a MapReduce framework to substrate networks, the traffic generated during the shuffling process must go through a network in which the network elements can be virtualized as shuffling nodes.

Recently, virtualization has emerged in communication networks, so multiple Virtual Networks (VNs) may concurrently run over a single substrate network. Virtual Machines (VMs) also traditionally virtualize servers' resources. VNs together with VMs underpin Virtualized Data Centers (VDCs). A VDC in the virtual layer consists of virtual networks in which VMs are connected with virtual links by virtual switches/routers. In the substrate layer, it consists of a data center network in which physical servers are connected with physical links by physical switches/routers. A VDC embedding process maps requested virtual nodes and links onto physical nodes and paths of a data center network, respectively.

Traditional data centers are moving toward virtualized data centers in order to address their limitations regarding network performance, security, and manageability [5]. Accordingly, in this paper, we consider MapReduce-based virtual networks in VDCs. Each virtual network is heterogeneous, where a virtual node might be a mapper, a reducer, or a shuffler in a MapReduce framework. The data center network is also heterogeneous, with a multi-level topology such as tree or fat-tree. A substrate node could be a server, or a switch/router. In this case, a VDC embedding process requires to split computation-based virtual nodes (mappers/reducers) and embed them onto multiple physical servers, in order to parallelize the computation tasks. Therefore, every splitted and mapped virtual node acts a corresponding worker in the MapReduce framework, with the required network connectivity of the virtual network. The MapReduce framework handles the traf-

* Corresponding author.
E-mail addresses: eghazisaeedi@sce.carleton.ca (E. Ghazisaeedi), huang@sce.carleton.ca (C. Huang).

http://dx.doi.org/10.1016/j.comnet.2016.11.015
1389-1286/

fic's source and destination between Mapper and Reducer workers. Nevertheless, all the existing embedding methods for virtual networks assume a virtual node could be mapped only onto a single substrate node. Therefore, they are not able to map heterogeneous MapReduce-based VNs onto a heterogeneous data center network.

An example of the actual use-case for MapReduce-base virtual networks is a virtual network that includes two mapper, some shuffling, and two reducer nodes. The first mapper nodes processes a large collection of documents to find the pages that a specific set of words occurred in them. The generated intermediate key and value pairs by the mapper virtual node will be shuffled to a reducer virtual node in the virtual network, through the shuffler virtual nodes. Afterwards, the aggregated results by the reducer will be forwarded to another mapper virtual node through shuffler virtual nodes. The second mapper virtual node will process each found page to find the number of occurrence of each word in it. The results will be shuffled to another reducer virtual node in the virtual network, through the shuffler virtual nodes. The second reducer virtual node aggregates the received data and finds a pattern between the given set of words and the most repeated words in the relative pages of documents.

Incast traffic pattern in a data center network firstly has been introduced in [6]. In a MapReduce framework, multiple mappers may simultaneously send intermediate key and value pairs to a single reducer. Therefore, the bottleneck physical link over the physical path to the reducer will likely be congested. In non-virtualized data centers, the incast congestion normally happens in the bottleneck physical link that connects a shallow-buffered Top-of-Rack (ToR) switch to an end-server that hosts a reducer [7–11]. The converged traffic flows deplete either the switch memory or the maximum allowed for that interface, resulting in packet loss [8]. The incast traffic also may cause a long queueing delay called incast queueing delay [8]. MapReduce requirements for low latency are directly related to the quality of the result returned and thus revenue. Consequently, it is vital for providers to control the incast delay.

The incast problem in VDCs is different from the incast problem in non-virtualized (traditional) data centers. A virtual link in a VDC is commonly mapped onto a physical path rather than a physical link. During a VDC embedding process, a specific amount of bandwidth capacity is allocated to each virtual link in physical paths. The traffic flows in the allocated physical paths are limited to their assigned bandwidth capacity. Hence, the incast problem might happen over the bottleneck allocated physical paths to virtual links. In this case, the incast traffic may face the incast queueing delay in multiple physical links over the bottleneck allocated path, leading to a longer cumulative incast queueing delay in the virtual link. This problem is significantly different from the scenario when incast happens only in a single bottleneck physical link, which connects a ToR switch to an end-server, in the case of non-virtualized data centers. To the best of our knowledge this problem is not introduced in any existing research studies.

In this paper, we propose GreenMap, an energy-efficient embedding method for MapReduce-based virtual networks that also controls the incast queueing delay. We formulate a Mixed-Integer Disciplined Convex Program (MIDCP) for this problem. Since the formulated MIDCP is $\mathcal{NP}$-hard, it is not scalable to large network sizes. Therefore, we also propose a novel and scalable heuristic for GreenMap. Both of the MIDCP and the heuristic map heterogeneous MapReduce-based VNs onto a heterogeneous data center network. They minimize a VDC's total consumed energy by physical servers, physical switches/routers, and physical links. They also control the incast queueing delay.

In this regard, we may split computation-based virtual nodes and map them onto multiple server substrate nodes. Accordingly, we address the emerged challenge of splitting and mapping adjacent virtual links of splitted and mapped virtual nodes. We might also collocate multiple splitted computation-based virtual nodes of a VN in a single server substrate node. The shuffler virtual nodes are assumed to be core/aggregation level nodes in the data center network with a multi-level topology. Besides, shuffler virtual nodes do not perform large computation tasks, and therefore there is no need of parallel computation in this case. Thus, we do not split a shuffler virtual node. A shuffler virtual node is mapped only onto a single switch/router substrate node. But, we may collocate multiple shuffler virtual nodes of a VN in a single switch/router substrate node. Note that the substrate path that connects an allocated computation-based virtual node in a server substrate node to an allocated shuffler virtual node in a switch/router substrate node, or vice versa, may traverse through multiple intermediate switches/routers.

Our method also controls the introduced incast queueing delay in any adjacent virtual link of a mapped reducer virtual node. This is achieved by calculating the average end-to-end queueing delay for incast traffic pattern during the virtual link mapping process. The incast queueing delay in an adjacent virtual link of an embedded reducer virtual node is related to the mentioned challenge of virtual link mapping. We demonstrate how controlling the incast queueing delay impacts the embedding process, the level of energy saving, and the network's admittance ratio.

We evaluated both the MIDCP and the heuristic for GreenMap over randomly generated VDC scenarios. Simulations results prove that the MIDCP and the heuristic save larger amounts of energy in the data center network than an existing energy-efficient embedding method for VNs that do not allow virtual node splitting. Besides, the simulations confirm that the incast queueing delay is controlled, and illustrate the influence of controlling the incast queueing delay on energy saving rates and the network's admittance ratio. It is also demonstrated that the heuristic could achieve closely to the optimum points set by the MIDCP.

The rest of this paper is organized as follows: The related works and our contributions in this paper are discussed in Section 2. We define our network model in Section 3, and study related power models in Section 4. The MIDCP and the heuristic for GreenMap are formulated in Section 5, and Section 6, respectively. The performance of the solutions is evaluated in Section 7. The paper concludes in Section 8.

2. Related works

There are few research studies suggested energy-efficient embedding processes that map VNs onto a substrate network, energy-wise [5,12–15]. These have been performed by modifying the physical resources' weights based on their power consumption in [12], and consolidating VNs to the smallest number of substrate network elements in [5,13–16]. However, all of them assume a virtual node could be mapped only onto a single substrate node. This is also the case in other proposed VN embedding methods that do not concern about energy efficiency [17–21]. Moreover, none of these embedding methods provides a tool that could handle the heterogeneity of both of MapReduce-based VNs and a data center network, at the same time.

On the other hand, existing research studies on incast problem, *for non-virtualized networks*, are focused on creating new traffic control mechanisms, and updating Transmission Control Protocol (TCP). They either tried to decrease packet loss by increasing the memory of shallow-buffered switches [8,22,23], or increase recovery speed of TCP by decreasing the value of Retransmission Timeout (RTO) [10,22,24]. Nonetheless, the existing solutions for incast problem are not efficient, and traffic control algorithms alone are difficult to mitigate incast congestion [11]. Besides, they do not provide a solution for incast problem in VDCs.

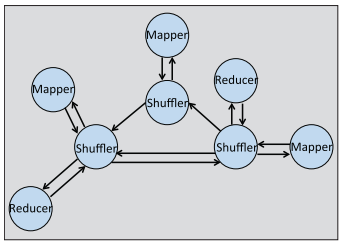

Fig. 1. A MapReduce-based VN's topology.

Our Contributions: (a) To the best of our knowledge, this is the first study on energy-efficient embedding of *MapReduce-based virtual networks* onto a data center network. *(b)* Different from [5,12–15,17–21], our approach makes it probable to split and map computation-based virtual nodes onto the data center network. Accordingly, it enables the providers to embed computation-based VNs onto the data center network. *(c)* Different from [5,12–15,17–21], our solution handles *heterogeneity* of MapReduce-based VNs and a data center network. *(d)* For the first time, we introduce a new incast problem for virtualized data centers. *(e)* We demonstrate a novel approach that controls the introduced incast queueing delay. *(f)* We tackle the incast problem during the provisioning process. So, it prevents the incast problem to happen at the first point. This is a much more efficient approach in comparison to the existing solutions for incast in [8,10,22,22–24] that try to recover the connection after that incast happens. *(g)* The problem is formulated as a MIDCP. *(h)* A novel and scalable heuristic is also proposed for the problem that could achieve closely to the optimum points. *(i)* We examine both the MIDCP and the heuristic through extensive simulations, and check the impacts of different factors. *(j)* We demonstrate how controlling incast queueing delay may affect the energy saving level and the network's admittance ratio.

3. Network model

We consider a heterogeneous data center network with a multi-level topology, such as tree or fat-tree, as the substrate network. It is modeled as a directed graph $G_s = (V_s, E_s)$. The directed graph provides higher level of flexibility in regard to routing traffic flows. V_s and E_s denote the set of substrate nodes and substrate links, respectively. The ith substrate node v_s^i could be a server, or a switch/router. The ith substrate node, which is a server, is represented by $\bar{v}_s^i$. $C_c(\bar{v}_s^i)$ is the Central Processing Unit (CPU) capacity of $\bar{v}_s^i$. $\bar{V}_s$ denotes the set of server substrate nodes. Moreover, the ith substrate node, which is a switch/router, is represented by $\tilde{v}_s^i$. $C_b(\tilde{v}_s^i)$ is the switching capacity of $\tilde{v}_s^i$. $\tilde{V}_s$ denotes the set of switch/router substrate nodes. Therefore, $V_s = \{\bar{V}_s \cup \tilde{V}_s\}$.

A substrate link $l_s^{i,j}$ in a data center network connects the ith substrate node to the jth substrate node. $C_b(l_s^{i,j})$ is the bandwidth capacity of $l_s^{i,j}$.

The nth virtual network is modeled as a directed graph $G_n = (V_n, E_n)$, where V_n and E_n denote the set of virtual nodes and virtual links in the nth VN, respectively. Φ is the set of all of the VNs, and $L_n = |E_n|$. Fig. 1 shows an example of a MapReduce-based VN's topology. In this framework, the kth virtual node v_n^k in the nth VN could be a mapper, a reducer, or a shuffler. The kth virtual node in the nth VN, which is a mapper, is represented by $\dot{v}_n^k$. $\hat{C}_c(\dot{v}_n^k)$ is the requested CPU capacity for $\dot{v}_n^k$. VN customers may specify the minimum $\hat{C}_c^{\check{m}}(\dot{v}_n^k)$ and the maximum $\hat{C}_c^{\hat{m}}(\dot{v}_n^k)$ amount of required CPU capacity per allocated physical machine for $\dot{v}_n^k$. The job arrival of a mapper could be modelled with Poisson process [25]. Besides, a mapper node that processes the jobs, is modelled as a M/M/1 queue [25]. Therefore, according to Jackson Networks theorem, the generated traffic of a mapper node could be modelled with Poisson process. In this regard, we assume a mapper virtual node $\dot{v}_n^k$ generates traffic that follows Poisson process with the mean rate of $\lambda(\dot{v}_n^k)$. Considering $\dot{V}_n$ as the set of mapper virtual nodes in the nth VN, $\lambda_n^{\mathcal{M}}$ is equal to $\sum_{k\in\dot{V}_n}\lambda(\dot{v}_n^k)$. Besides, the kth virtual node in the nth VN, which is a reducer, is represented by $\ddot{v}_n^k$. $\hat{C}_c(\ddot{v}_n^k)$ is the demanded CPU capacity for $\ddot{v}_n^k$. VN customers also may specify the minimum $\hat{C}_c^{\check{m}}(\ddot{v}_n^k)$ and the maximum $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$ amount of required CPU capacity per allocated physical machine for $\ddot{v}_n^k$. $\ddot{V}_n$ denotes the set of reducer virtual nodes in the nth VN. Furthermore, the kth virtual node in the nth VN, which is a shuffler, is represented by $\tilde{v}_n^k$. We assume a shuffler virtual node $\tilde{v}_n^k$ only has the switching capacity demand $\hat{C}_b(\tilde{v}_n^k)$, that is equal to the summation of its adjacent virtual links' bandwidth demands. $\tilde{V}_n$ denotes the set of shuffler virtual nodes in the nth VN. Therefore, $V_n = \{\dot{V}_n \cup \ddot{V}_n \cup \tilde{V}_n\}$.

The set of virtual links is presented as a set of ordered virtual node pairs $l_n^{a^m,b^m}, m = 1, 2, \ldots, L_n$. a^m and b^m are source and sink virtual nodes of the mth virtual link in the corresponding VN, respectively. $\hat{C}_b(l_n^{a^m,b^m})$ is the bandwidth demand of $l_n^{a^m,b^m}$. We presume computation-based virtual nodes are connected through shuffler virtual nodes in a virtual network's topology. In other words, there is no pair of computation-based virtual nodes that are connected directly to each other in a VN's topology. This is the real case in a MapReduce framework.

4. Power models

Considering a described heterogeneous data center network as the substrate network, we have three major power consumers, physical servers, physical switches/routers, and physical links. In this paper, we intend to minimize the total consumed energy by them in a VDC. We study a power model for each of these substrate elements.

$\tilde{p}(\bar{v}_s^i) = \alpha(\bar{v}_s^i)\tilde{p}^b(\bar{v}_s^i) + \frac{\check{\phi}(\bar{v}_s^i)}{C_c(\bar{v}_s^i)}\left(\tilde{p}^m(\bar{v}_s^i) - \tilde{p}^b(\bar{v}_s^i)\right)$ defines the actual power consumption $\tilde{p}(\bar{v}_s^i)$ for a server substrate node $\bar{v}_s^i$ [26]. $\alpha(\bar{v}_s^i)$ denotes the status of $\bar{v}_s^i$. It is 1, if $\bar{v}_s^i$ is active. Otherwise, it is 0. $\tilde{p}^b(\bar{v}_s^i)$ is the base power consumption of $\bar{v}_s^i$ required to keep it active. $\tilde{p}^m(\bar{v}_s^i)$ is the maximum power consumption of $\bar{v}_s^i$. The total allocated processing capacity $\check{\phi}(\bar{v}_s^i)$ to computation-based virtual nodes in $\bar{v}_s^i$ changes its actual power consumption between $\tilde{p}^b(\bar{v}_s^i)$ and $\tilde{p}^m(\bar{v}_s^i)$, linearly. $\tilde{p}^b(\bar{v}_s^i)$ and $\tilde{p}^m(\bar{v}_s^i)$ could be found through calibration experiments, e.g. in [27].

$\tilde{p}(\tilde{v}_s^i) = \alpha(\tilde{v}_s^i)\tilde{p}^b(\tilde{v}_s^i) + \frac{r(\tilde{v}_s^i)}{C_b(\tilde{v}_s^i)}\left(\tilde{p}^m(\tilde{v}_s^i) - \tilde{p}^b(\tilde{v}_s^i)\right)$ defines the actual power consumption $\tilde{p}(\tilde{v}_s^i)$ for a switch/router substrate node $\tilde{v}_s^i$ [28]. $\alpha(\tilde{v}_s^i)$ shows the status of $\tilde{v}_s^i$. $\tilde{p}^b(\tilde{v}_s^i)$ and $\tilde{p}^m(\tilde{v}_s^i)$ are the base and maximum power consumptions of $\tilde{v}_s^i$, respectively. The total traffic load $r(\tilde{v}_s^i)$ in $\tilde{v}_s^i$ changes its actual power consumption between its base and maximum power consumptions, linearly.

According to [28], $\tilde{p}^b(\tilde{v}_s^i)$ is $0.85C_b(\tilde{v}_s^i)^{\frac{2}{3}}$, and $\tilde{p}^m(\tilde{v}_s^i)$ is $C_b(\tilde{v}_s^i)^{\frac{2}{3}}$. Consequently, equation of the actual power consumption for a switch/router could be rewritten as $\tilde{p}(\tilde{v}_s^i) = 0.85\alpha(\tilde{v}_s^i)C_b(\tilde{v}_s^i)^{\frac{2}{3}} + \frac{0.15r(\tilde{v}_s^i)}{C_b(\tilde{v}_s^i)^{\frac{1}{3}}}$.

Similarly, $\tilde{p}(l_s^{i,j}) = \alpha(l_s^{i,j})\tilde{p}^b(l_s^{i,j}) + \frac{r(l_s^{i,j})}{C_b(l_s^{i,j})}(\tilde{p}^m(l_s^{i,j}) - \tilde{p}^b(l_s^{i,j}))$ defines the actual power consumption $\tilde{p}(l_s^{i,j})$ of a substrate link $l_s^{i,j}$ [28]. $\tilde{p}^b(l_s^{i,j})$ and $\tilde{p}^m(l_s^{i,j})$ are the base and maximum power consumptions of $l_s^{i,j}$, respectively. The total allocated traffic capacity $r(l_s^{i,j})$ to virtual links in $l_s^{i,j}$ varies its actual power consumption, linearly. Note that $\tilde{p}^b(l_s^{i,j})$ and $\tilde{p}^m(l_s^{i,j})$ are normally defined for different ranges of link bandwidth capacity, based on the link's length and the type of the cable. Some numerical amounts for them are given in [28].

According to the above defined power models, the most effective way of saving power in any of the mentioned substrate elements is shutting down the device. Note that the same amount of processing/traffic demand might cause different amounts of power consumption in distinct substrate elements, based on the element's processing/bandwidth capacity, its base power consumption, and its maximum power consumption.

5. Mixed-integer disciplined convex program

In this problem, the processing/bandwidth capacity of every substrate node, and the bandwidth capacity of every substrate link are given. Besides, each VN's topology, the processing/bandwidth demand of every virtual node and their minimum/maximum processing demands per physical server, the bandwidth demand of every virtual link, and $\lambda(\dot{v}_n^k)$ of each mapper virtual node are known. We need to find a mapping for every VN such that the data center network's total energy consumption by physical servers, physical switches/routers, and physical links, is minimized. We also require to control the incast queueing delay according to the given $\hat{D}$, which is the maximum tolerable queueing delay in a virtual link.

In this regard, we may split the computation-based virtual nodes and map splitted virtual nodes onto multiple physical servers. But, we map a shuffler virtual node onto a single physical switch/router, as they do not need parallel computations. We also may collocate virtual nodes (mappers/reducers/shufflers) of a VN in a single relevant substrate node. A new challenge will emerge in this model. The issue is what is the bandwidth capacity that needs to be allocated to an adjacent virtual link of a splitted and mapped computation-based virtual node. Likely, a splitted computation-based virtual node processes proportional traffic to its assigned processing capacity. Therefore, the amount of bandwidth we allocate to an adjacent virtual link of a splitted and mapped computation-based virtual node is proportional to its assigned processing capacity. Thus, in this model, more likely the introduced incast problem may arise in an adjacent virtual link of a splitted reducer virtual node. Note that in order to avoid out of order packet delivery, we do not split generated traffic of an allocated virtual node.

In order to control the introduced incast queueing delay, it is required to find the end-to-end queueing delay for incast traffic pattern in the substrate path allocated to every virtual link that terminates at a splitted and mapped reducer virtual node. In this case the end-to-end delay is the total delay that a packet takes to travel from an output port of a Mapper virtual node to an input port of a Reducer virtual node, which is the total delay the networking elements impose to the traffic. So, the delay at servers' input ports does not have any effect on the end-to-end queueing delay for incast traffic pattern. In incast traffic pattern, $\lambda_n^{\mathcal{M}}$ is the mean traffic rate in the substrate path. Most of today's switches/routers are internally non-blocking (i.e. the internal switch fabric speed is much faster than each output port). Therefore, traffic can only be blocked by limited bandwidths of output ports which are defined earlier by the link bandwidth capacity. We model the queue of an allocated bandwidth capacity to a virtual link in a substrate link by M/M/1 queue. According to Jackson Networks theorem and because we do not split generated traffic of an allocated virtual node, the end-to-end incast queueing delay in the substrate path could be calculated by knowing the amount of allocated traffic capacity to the virtual link in each physical link over the substrate path, and $\lambda_n^{\mathcal{M}}$. Since the amount of bandwidth we allocate to the virtual links are proportional to the assigned capacity of its end virtual nodes, the way we split reducer virtual nodes impacts the incast queueing delay. Besides, the substrate node which we map the splitted reducer virtual node onto, and accordingly the allocated substrate path to each of the adjacent virtual links, also may influence the incast queueing delay. Clearly, this limits the level of freedom regarding energy-efficient embedding of the VNs, and may affect the energy saving rate and the network's admittance ratio.

Considering the discussed definitions, we formulate the following MIDCP as a solution for GreenMap:

5.1. Optimization variables

- $\check{\phi}(v_n^k, v_s^i)$ is a real variable. It represents the fraction of processing/switching demand of a virtual node v_n^k, that is allocated in a substrate node v_s^i. v_n^k could be $\dot{v}_n^k$, $\ddot{v}_n^k$, or $\tilde{v}_n^k$. v_s^i could be $\bar{v}_s^i$ or $\tilde{v}_s^i$.
- $\alpha(\bar{v}_s^i)$, $\alpha(\tilde{v}_s^i)$, and $\alpha(l_s^{i,j})$ are binary variables. They denote the status of respective substrate node/link. The variable is 1 in the case the device is active. Otherwise, it is 0.
- $\alpha(\dot{v}_n^k, \bar{v}_s^i)$, $\alpha(\ddot{v}_n^k, \bar{v}_s^i)$, $\alpha(\tilde{v}_n^k, \tilde{v}_s^i)$ are binary variables denote whether the virtual node is allocated in the substrate node (the variable is 1), or not (the variable is 0).
- $\alpha(l_n^{x,y}(m))$ is a binary variable. It is 1 if the whole or a fraction of $v_n^{a^m}$'s processing/switching demand is allocated in v_s^x, and the whole or a fraction of $v_n^{b^m}$'s processing/switching demand is allocated in v_s^y. Otherwise, it is 0. $l_n^{x,y}(m)$ is a sub virtual link of $l_n^{a^m,b^m}$ that connects the allocated source virtual node in v_s^x to the allocated sink virtual node in v_s^y.
- $z^{i,j}(l_n^{x,y}(m))$ is a binary variable. It is 1 if the allocated substrate path for $l_n^{x,y}(m)$ passes through $l_s^{i,j}$. Otherwise, it is 0.
- $\dot{d}_n^{x,y}(m)$ is a real variable. It is the fraction of $\hat{C}_b(l_n^{a^m,b^m})$ that needs to be allocated to $l_n^{x,y}(m)$.
- $\ddot{d}^{i,j}(l_n^{x,y}(m))$ is a real variable. It shows the amount of allocated traffic capacity to $l_n^{x,y}(m)$ in $l_s^{i,j}$.

5.2. Objective function

Our objective is minimizing the total consumed energy by physical servers, physical switches/routers, and physical links, in a VDC. Eq. (1) maintains this objective.

$$\begin{aligned} Minimize \Bigg\{ & \sum_{i\in\bar{V}_s} \alpha(\bar{v}_s^i)\tilde{p}^b(\bar{v}_s^i) + \frac{\check{\phi}(\bar{v}_s^i)}{C_c(\bar{v}_s^i)}\left(\tilde{p}^m(\bar{v}_s^i) - \tilde{p}^b(\bar{v}_s^i)\right) \\ & + \sum_{i\in\tilde{V}_s} 0.85\alpha(\tilde{v}_s^i)C_b(\tilde{v}_s^i)^{\frac{2}{3}} + \frac{0.15r(\tilde{v}_s^i)}{C_b(\tilde{v}_s^i)^{\frac{1}{3}}} \\ & + \sum_{(i,j)\in E_s} \alpha(l_s^{i,j})\tilde{p}^b(l_s^{i,j}) + \frac{r(l_s^{i,j})}{C_b(l_s^{i,j})}\left(\tilde{p}^m(l_s^{i,j}) - \tilde{p}^b(l_s^{i,j})\right) \Bigg\} \end{aligned} \tag{1}$$

5.3. Constraints

Every virtual node must be mapped onto one or multiple substrate nodes. This is ensured by the first constraint in Eq. (2). Note that this constraint allows collocation of multiple virtual nodes of a VN onto a single substrate node. The constraint in Eq. (3) prevents a computation-based virtual node to be mapped onto a switch/router substrate node. Besides, the constraint in Eq. (4) forbids a shuffler virtual node to be mapped onto a server substrate node.

$$\sum_{i\in V_s} \check{\phi}(v_n^k, v_s^i) = 1\,, \quad \forall n \in \{n|G_n \in \Phi\}, \forall k \in V_n \tag{2}$$

$$\sum_{n\in\{n|G_n\in\Phi\}} \left(\sum_{k\in\dot{V}_n} \check{\phi}(\dot{v}_n^k, \tilde{v}_s^i) + \sum_{k\in\ddot{V}_n} \check{\phi}(\ddot{v}_n^k, \tilde{v}_s^i) \right) = 0,\ \forall i \in \tilde{V}_s \tag{3}$$

$$\sum_{n\in\{n|G_n\in\Phi\}}\sum_{k\in\tilde{V}_n}\check{\phi}(\tilde{v}_n^k,\tilde{v}_s^i)=0\,,\quad \forall i\in\tilde{V}_s \tag{4}$$

The allocated processing capacity to a computation-based virtual node in a server substrate node must be equal or greater than the requested minimum CPU capacity per physical machine for the virtual node, and equal or less than the requested maximum CPU capacity per physical machine for the virtual node. This is confirmed in the constraint in Eq. (5) for mapper virtual nodes, and in the constraint in Eq. (6) for reducer virtual nodes. Note that the given ratio must be feasible. For example, $\frac{\hat{C}_c^{\dot{m}}(\dot{v}_n^k)}{\hat{C}_c(\dot{v}_n^k)}$ could not be greater than 0.5.

$$\frac{\hat{C}_c^{\dot{m}}(\dot{v}_n^k)}{\hat{C}_c(\dot{v}_n^k)}\alpha(\dot{v}_n^k,\tilde{v}_s^i)\le\check{\phi}(\dot{v}_n^k,\tilde{v}_s^i)\le\frac{\hat{C}_c^{\dot{m}}(\dot{v}_n^k)}{\hat{C}_c(\dot{v}_n^k)}\alpha(\dot{v}_n^k,\tilde{v}_s^i)\,,\quad \forall i\in\tilde{V}_s,\forall n\in\{n|G_n\in\Phi\},\forall k\in\dot{V}_n \tag{5}$$

$$\frac{\hat{C}_c^{\ddot{m}}(\ddot{v}_n^k)}{\hat{C}_c(\ddot{v}_n^k)}\alpha(\ddot{v}_n^k,\tilde{v}_s^i)\le\check{\phi}(\ddot{v}_n^k,\tilde{v}_s^i)\le\frac{\hat{C}_c^{\ddot{m}}(\ddot{v}_n^k)}{\hat{C}_c(\ddot{v}_n^k)}\alpha(\ddot{v}_n^k,\tilde{v}_s^i)\,,\quad \forall i\in\tilde{V}_s,\forall n\in\{n|G_n\in\Phi\},\forall k\in\ddot{V}_n \tag{6}$$

The constraint in Eq. (7) restricts the program to map a shuffler virtual node only onto one switch/router substrate node.

$$\check{\phi}(\tilde{v}_n^k,\tilde{v}_s^i)=\alpha(\tilde{v}_n^k,\tilde{v}_s^i),\ \forall i\in\tilde{V}_s,\forall n\in\{n|G_n\in\Phi\},\forall k\in\tilde{V}_n \tag{7}$$

Besides, total allocated processing capacities to computation-based virtual nodes in a server substrate node must be equal or less than the substrate node's CPU capacity, as shown in the constraint in Eq. (8). Total allocated switching capacities to shuffler virtual nodes in a switch/router substrate node also must be equal or less than the substrate node's switching capacity, as indicated in the constraint in Eq. (9).

$$\sum_{n\in\{n|G_n\in\Phi\}}\left(\sum_{k\in\dot{V}_n}\check{\phi}(\dot{v}_n^k,\tilde{v}_s^i)\hat{C}_c(\dot{v}_n^k)+\sum_{k\in\ddot{V}_n}\check{\phi}(\ddot{v}_n^k,\tilde{v}_s^i)\hat{C}_c(\ddot{v}_n^k)\right)\le C_c(\tilde{v}_s^i)\,,\quad \forall i\in\tilde{V}_s \tag{8}$$

$$\sum_{n\in\{n|G_n\in\Phi\}}\sum_{k\in\tilde{V}_n}\check{\phi}(\tilde{v}_n^k,\tilde{v}_s^i)\hat{C}_b(\tilde{v}_n^k)\le C_b(\tilde{v}_s^i)\,,\quad \forall i\in\tilde{V}_s \tag{9}$$

In the next step, the program needs to map adjacent virtual links of allocated virtual nodes. We discussed that computation-based virtual nodes are connected through shuffler virtual nodes in a VN's topology. So, we have two types of virtual links in a VN. First, a virtual link that connects a computation-based virtual node to a shuffler virtual node, or vice versa. Second, a virtual link that connects a shuffler virtual node to another shuffler virtual node. Therefore, $\check{\phi}(v_n^{a^m},v_s^x)\check{\phi}(v_n^{b^m},v_s^y)\hat{C}_b(l_n^{a^m,b^m})$ is the fraction of $\hat{C}_b(l_n^{a^m,b^m})$ that needs to be allocated to $l_n^{x,y}(m)$. This amount is proportional to the assigned processing/switching capacity to the mapped source and sink virtual nodes of the virtual link $l_n^{a^m,b^m}$. This is reflected in the constraint in Eq. (10). This constraint could be replaced by the linear constraints in Appendix B.

$$\dot{d}_n^{x,y}(m)=\check{\phi}(v_n^{a^m},v_s^x)\check{\phi}(v_n^{b^m},v_s^y),\quad \forall x\in V_s,\forall y\in V_s,\forall n\in\{n|G_n\in\Phi\},m=1,\dots,L_n \tag{10}$$

B_1, B_2, B_3, and B_4 are large integer numbers. They must be large enough to be greater than the largest amount of the left-hand side of their respective inequality. If there is a non-zero bandwidth demand for $l_n^{x,y}(m)$, the first constraint in Eq. (11) forces $\alpha(l_n^{x,y}(m))$ to be 1. Then, the second constraint in Eq. (11) needs to route a single unit of data from the xth substrate node to the yth substrate node. Because the variable $z^{i,j}(l_n^{x,y}(m))$ is binary, the unit of data could not be splitted. Besides, the third constraint in Eq. (11) limits the program routing, so the maximum number of incoming and outgoing flows for a sub virtual link, in any substrate node, is two. This maintains a single loopless path. The driven route will be used as the substrate path for $l_n^{x,y}(m)$. Note that if $\alpha(l_n^{x,y}(m))=0$, no substrate path is allocated to $l_n^{x,y}(m)$.

$$\begin{aligned}&\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})\le B_1\alpha(l_n^{x,y}(m))\,,\\&\sum_{\{j|(i,j)\in E_s\}}z^{i,j}(l_n^{x,y}(m))-\sum_{\{j|(j,i)\in E_s\}}z^{j,i}(l_n^{x,y}(m))\\&=\begin{cases}\alpha(l_n^{x,y}(m)) & \text{if}\quad i=x\\ -\alpha(l_n^{x,y}(m)) & \text{if}\quad i=y\\ 0 & \text{otherwise}\end{cases},\\&\sum_{\{j|(i,j)\in E_s\}}z^{i,j}(l_n^{x,y}(m))+\sum_{\{j|(j,i)\in E_s\}}z^{j,i}(l_n^{x,y}(m))\le 2,\\&\forall i\in V_s,\forall x\in V_s,\forall y\in V_s,\forall n\in\{n|G_n\in\Phi\},m=1,\dots,L_n\end{aligned} \tag{11}$$

Furthermore, total allocated traffic capacity $r(l_s^{i,j})$ in a substrate link $l_s^{i,j}$ must be less than its physical bandwidth capacity $C_b(l_s^{i,j})$, as expressed in the constraint in Eq. (12). $r(l_s^{i,j})$ is the summation of every allocated traffic capacity $\ddot{d}^{i,j}(l_n^{x,y}(m))$ to a sub virtual link $l_n^{x,y}(m)$ in $l_s^{i,j}$. $\ddot{d}^{i,j}(l_n^{x,y}(m))$ is equal to $z^{i,j}(l_n^{x,y}(m))\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})$. If the allocated substrate path to $l_n^{x,y}(m)$ passes through $l_s^{i,j}$, $z^{i,j}(l_n^{x,y}(m))$ is 1, and therefore $\ddot{d}^{i,j}(l_n^{x,y}(m))$ is $\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})$. Otherwise, $\ddot{d}^{i,j}(l_n^{x,y}(m))$ is 0. Note that the constraint in Eq. (13) could be replaced by the linear constraints in Appendix C.

$$\begin{aligned}&r(l_s^{i,j})\le C_b(l_s^{i,j})\,,\quad \forall(i,j)\in E_s,\quad \text{where:}\\&r(l_s^{i,j})=\sum_{x\in V_s}\sum_{y\in V_s}\sum_{n\in\{n|G_n\in\Phi\}}\sum_{m=1}^{L_n}\ddot{d}^{i,j}(l_n^{x,y}(m))\\&\ddot{d}^{i,j}(l_n^{x,y}(m))=z^{i,j}(l_n^{x,y}(m))\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})\,,\end{aligned} \tag{12}$$

$$\forall(i,j)\in E_s,\forall x\in V_s,\forall y\in V_s,\forall n\in\{n|G_n\in\Phi\},\ m=1,\dots,L_n \tag{13}$$

Moreover, the total incoming and outgoing traffic of a switch/router substrate node must be less than its switching capacity. This is confirmed in the constraint in Eq. (14).

$$\begin{aligned}&r(\tilde{v}_s^i)\le C_b(\tilde{v}_s^i)\,,\forall i\in\tilde{V}_s\,,\quad \text{where:}\\&r(\tilde{v}_s^i)=\sum_{(i,j)\in E_s}r(l_s^{i,j})+\sum_{(j,i)\in E_s}r(l_s^{j,i})\end{aligned} \tag{14}$$

Considering M/M/1 queue and Jackson Networks theorem, and because we do not split generated traffic by an allocated virtual node, the average end-to-end incast queueing delay of a sub virtual link $l_n^{x,y}(m)$ could be calculated by the sum of $\frac{1}{\mu(l_n^{x,y}(m))-\lambda_n^M}$ for every substrate link $l_s^{i,j}$ that a traffic capacity is allocated to $l_n^{x,y}(m)$ in it ($z^{i,j}(l_n^{x,y}(m))=1$). The mean service rate $\mu(l_n^{x,y}(m))$ for $l_n^{x,y}(m)$ is equal to $\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})$.

Thus, the left-hand side of the first inequality constraint in Eq. (15) calculates the end-to-end incast queueing delay in the allocated substrate path to a sub virtual link that connects a mapped shuffler virtual node to a mapped and splitted reducer virtual node. Note that if no traffic capacity is allocated to the sub virtual link $l_n^{x,y}(m)$, then $z^{i,j}(l_n^{x,y}(m))$ is 0, and therefore the delay is 0. $2(1-\alpha(l_n^{x,y}(m)))$ is added to ensure the denominator is never 0. According to the first constraint in Eq. (11), if $\dot{d}_n^{x,y}(m)$ is greater

than 0, then $\alpha(l_n^{x,y}(m))$ is 1, and therefore the denominator is $\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m}) - \lambda_n^{\mathcal{M}}$. Otherwise, if $\dot{d}_n^{x,y}(m)$ is 0, then $z^{i,j}(l_n^{x,y}(m))$ of any substrate link is 0, and the denominator is $\lambda_n^{\mathcal{M}}$.

Hence, according to the proof in Appendix A, the first inequality constraint in Eq. (15) is a disciplined convex constraint [29], which verifies the incast queueing delay in the allocated substrate path to every virtual link that terminates at a mapped reducer virtual node, is less than $\hat{D}$. Besides, the second constraint in Eq. (15) confirms that every queue in the network is stable.

$$\sum_{(i,j)\in E_s} \frac{z^{i,j}(l_n^{x,y}(m))^2}{\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m}) - \lambda_n^{\mathcal{M}} + 2\big(1-\alpha(l_n^{x,y}(m))\big)\lambda_n^{\mathcal{M}}} \le \hat{D},$$
$$\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m}) > \alpha(l_n^{x,y}(m))\lambda_n^{\mathcal{M}},$$
$$\forall n \in \{n|G_n \in \Phi\}, m \in \{m|m = 1,\dots,L_n; b^m \in \ddot{V}_n\}, \forall x \in V_s, \forall y \in V_s \quad (15)$$

The constraints in Eq. (16) check the respective device's status.

$$\check{\phi}(\bar{v}_s^i) \le B_2\alpha(\bar{v}_s^i), \forall i \in \bar{V}_s, \quad \text{where:}$$
$$\check{\phi}(\bar{v}_s^i) = \sum_{n\in G_n}\left(\sum_{k\in\dot{V}_n}\check{\phi}(\dot{v}_n^k, \bar{v}_s^i) + \sum_{k\in\ddot{V}_n}\check{\phi}(\ddot{v}_n^k, \bar{v}_s^i)\right);$$
$$r(\tilde{v}_s^i) \le B_3\alpha(\tilde{v}_s^i), \quad \forall i \in \tilde{V}_s;$$
$$r(l_s^{i,j}) \le B_4\alpha(l_s^{i,j}), \quad \forall (i,j) \in E_s \quad (16)$$

Besides, the variables have to hold the following bounds:

$$0 \le \check{\phi}(\dot{v}_n^k, \bar{v}_s^i) \le 1, \quad \forall i \in V_s, \forall n \in \{n|G_n \in \Phi\}, \forall k \in V_n$$
$$\alpha(\bar{v}_s^i), \alpha(\tilde{v}_s^i) \in \{0,1\}, \quad \forall i \in V_s$$
$$\alpha(l_s^{i,j}) \in \{0,1\}, \quad \forall (i,j) \in E_s$$
$$\alpha(\dot{v}_n^k, \bar{v}_s^i) \in \{0,1\}, \quad \forall i \in \bar{V}_s, \forall n \in \{n|G_n \in \Phi\}, \forall k \in \dot{V}_n$$
$$\alpha(\ddot{v}_n^k, \bar{v}_s^i) \in \{0,1\}, \quad \forall i \in \bar{V}_s, \forall n \in \{n|G_n \in \Phi\}, \forall k \in \ddot{V}_n$$
$$\alpha(\tilde{v}_n^k, \tilde{v}_s^i) \in \{0,1\}, \quad \forall i \in \tilde{V}_s, \forall n \in \{n|G_n \in \Phi\}, \forall k \in \tilde{V}_n$$
$$\alpha(l_n^{x,y}(m)), z^{i,j}(l_n^{x,y}(m)) \in \{0,1\}, \quad \forall x \in V_s, \forall y \in V_s, \forall n \in \{n|G_n \in \Phi\}, m = 1,\dots,L_n$$

The formulated MIDCP is a type of virtual network embedding problems. A virtual network embedding problem is $\mathcal{NP}$-hard [30]. In consequence, the defined MIDCP is $\mathcal{NP}$-hard, and it is not scalable to large network sizes.

6. Heuristic

Today's data centers include hundreds to thousands of physical servers. Therefore, we need to develop a heuristic algorithm for GreenMap that is scalable to large network sizes. The algorithm also has to achieve closely to the optimum points set by the formulated MIDCP. In this section, we propose such a heuristic for GreenMap.

The heuristic embeds MapReduce-based VNs onto a data center network, one by one, as the VN requests are received during the time. It maps a VN onto a data center network with the minimum additional energy consumption, and the hope of minimizing a VDC's total consumed energy at the end.

Four algorithms form our proposed heuristic for GreenMap. Each algorithm maps a part of the nth MapReduce-based virtual network onto a data center network. Algorithm 1 maps shuffler virtual nodes onto switch/router substrate nodes. Algorithm 2, and Algorithm 3 map reducer virtual nodes, and mapper virtual nodes onto server substrate nodes, respectively. Besides, Algorithm 4 maps virtual links onto substrate paths.

Algorithm 1 Mapping shuffler virtual nodes of nth VN.

1: **Input:** Given data for the VN and the data center network
2: **for all** unallocated $\tilde{v}_n^k$ such that $k \in \tilde{V}_n$ **do**
3: **for all** $\tilde{v}_s^i$ from top of S_L_1 **do**
4: **if** $\check{C}_b(\tilde{v}_s^i) \ge \hat{C}_b(\tilde{v}_n^k)$ **then**
5: $\check{C}_b(\tilde{v}_n^k, \tilde{v}_s^i) = \hat{C}_b(\tilde{v}_n^k)$
6: $\check{C}_b(\tilde{v}_s^i) = \check{C}_b(\tilde{v}_s^i) - \check{C}_b(\tilde{v}_n^k, \tilde{v}_s^i)$
7: **for all** mth virtual link such that: its source virtual node is $\tilde{v}_n^k$ and its sink virtual node is $\tilde{v}_n^l, l \in \tilde{V}_n$, **or** its source virtual node is $\tilde{v}_n^l, l \in \tilde{V}_n$ and its sink virtual node is $\tilde{v}_n^k$ **do**
8: **if** $\tilde{v}_n^l$ is unallocated **then**
9: **for all** $\tilde{v}_s^j$ from top of S_L_1 **do**
10: **if** $\check{C}_b(\tilde{v}_s^j) \ge \hat{C}_b(\tilde{v}_n^l)$ **then**
11: $\check{C}_b(\tilde{v}_n^l, \tilde{v}_s^j) = \hat{C}_b(\tilde{v}_n^l)$
12: $\check{C}_b(\tilde{v}_s^j) = \check{C}_b(\tilde{v}_s^j) - \check{C}_b(\tilde{v}_n^l, \tilde{v}_s^j)$
13: **Algorithm4**$(l_n^{i,j}(m))$, or **Algorithm4**$(l_n^{j,i}(m))$
14: **if** virtual link mapping was not successful **then**
15: $\check{C}_b(\tilde{v}_n^l, \tilde{v}_s^j) = 0$
16: $\check{C}_b(\tilde{v}_s^j) = \check{C}_b(\tilde{v}_s^j) + \check{C}_b(\tilde{v}_n^l, \tilde{v}_s^i)$
17: undo the modifications, check next $\tilde{v}_s^j$
18: **else**
19: break, and check the next virtual link in line 7
20: **end if**
21: **end if**
22: **end for**
23: **else**
24: **Algorithm4**$(l_n^{i,j}(m))$, or **Algorithm4**$(l_n^{j,i}(m))$
25: **if** this virtual link mapping was not successful **then**
26: undo the modifications
27: break, and check the next $\tilde{v}_s^i$ in line 3
28: **end if**
29: **end if**
30: **end for**
31: update S_L_1, break, and check the next $\tilde{v}_n^k$ in line 2
32: **end if**
33: **end for**
34: undo the modifications, and reject nth VN
35: **end for**

We first run Algorithm 1 to prepare the transit network for transferring generated intermediate data of computation-based virtual nodes. Afterwards, we run Algorithm 2 and map reducer virtual nodes onto server substrate nodes. Similar to the formulated MIDCP, the heuristic controls the incast queueing delay in every virtual link that terminates in a splitted and mapped reducer virtual node. Therefore, as it is discussed, we are limited regarding splitting and allocating reducer virtual nodes. However, we do not have such a limitation in embedding mapper virtual nodes. In consequence, in order to have more available resources in mapping reducer virtual nodes, we map reducer virtual nodes before embedding mapper virtual nodes. Afterwards, we run Algorithm 3 to allocate mapper virtual nodes in server substrate nodes. Algorithm 4 is called during the running process of the first three algorithms to embed the corresponding virtual links onto substrate paths.

6.1. Mapping shuffler virtual nodes

A shuffler virtual node is allocated in a single switch/router substrate node. Multiple shuffler virtual nodes might be collocated in a single switch/router. In order to minimize a VDC's energy consumption, our priority is to map a shuffler virtual node onto an already active switch/router with the minimum power consumption for the requested switching capacity. In this regard, we sort switch/router substrate nodes in a list called

Algorithm 2 Mapping reducer virtual nodes of nth VN.

Input: Mapping outcome of Algorithm 1
for all unallocated $\dot{v}_n^k$ from top of S_L_3 **do**
  $\mathcal{V}'^k_n = 0$
  for all $\bar{v}_s^i$ from top of S_L_2 **do**
    if $\hat{C'}_c(\dot{v}_n^k) \geq max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i))$ **then**
      if $\check{C}_c(\bar{v}_s^i) \geq min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$ **and** $\hat{C}_c^{\dot{m}}(\dot{v}_n^k)) \geq C_i(\dot{v}_n^k, \bar{v}_s^i))$ **then**
        $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$
      else if $max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i)) \leq \check{C}_c(\bar{v}_s^i) < min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$ **then**
        $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = \check{C}_c(\bar{v}_s^i)$
      end if
      if $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) > 0$ **then**
        $\check{C}_c(\bar{v}_s^i) = \check{C}_c(\bar{v}_s^i) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
        $\hat{C'}_c(\dot{v}_n^k) = \hat{C'}_c(\dot{v}_n^k) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
        $\mathcal{V}'^k_n(i) = \check{C}_c(\dot{v}_n^k, \bar{v}_s^i) - max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i))$
        $\mathcal{V}'^k_n = \mathcal{V}'^k_n + \mathcal{V}^k_n(i)$
        for all adjacent virtual link m of $\dot{v}_n^k$ that its shuffler virtual node is mapped on $v_s^j, j \in \tilde{V}_n$ **do**
          Algorithm4($l_n^{i,j}(m)$), or **Algorithm4**($l_n^{j,i}(m)$)
          if the virtual link mapping was not successful **then** undo the modifications, break, and check the next $\bar{v}_s^i$ in line 4
        end for
      end if
    else if $\hat{C'}_c(\dot{v}_n^k) \neq 0$ **and** $\check{C}_c(\bar{v}_s^i) \geq max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i))$ **and** $\mathcal{V}'^k_n \geq max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i)) - \hat{C'}_c(\dot{v}_n^k)$ **then**
      $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i))$
      $\check{C}_c(\bar{v}_s^i) = \check{C}_c(\bar{v}_s^i) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
      for all adjacent virtual link m of $\dot{v}_n^k$ that its shuffler virtual node is mapped on $v_s^j, j \in \tilde{V}_n$ **do**
        Algorithm4($l_n^{i,j}(m)$), or **Algorithm4**($l_n^{j,i}(m)$)
        if the virtual link mapping was not successful **then** undo the modifications, break, and check the next $\bar{v}_s^i$ in line 4
      end for
      for all xth v_s^x going backward from $x = i$ to the top of S_L_2 **do**
        $\check{C}_c(\dot{v}_n^k, \bar{v}_s^x) = \check{C}_c(\dot{v}_n^k, \bar{v}_s^x) - min(\mathcal{V}'^k_n(x), max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i)) - \hat{C'}_c(\dot{v}_n^k))$
        $\check{C}_c(\bar{v}_s^x) = \check{C}_c(\bar{v}_s^x) + min(\mathcal{V}'^k_n(x), max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i)) - \hat{C'}_c(\dot{v}_n^k))$
        update the allocated traffic capacities to the corresponding virtual links
        if total reduced allocated processing capacities is equal to $max(\hat{C}_c^{\dot{m}}(\dot{v}_n^k), C_i(\dot{v}_n^k, \bar{v}_s^i)) - \hat{C'}_c(\dot{v}_n^k)$ **then** update S_L_2, break, and check the next $\dot{v}_n^k$ in line 2
      end for
    else if $\hat{C'}_c(\dot{v}_n^k) = 0$ **then**
      update S_L_2, break, and check the next $\dot{v}_n^k$ in line 2
    end if
  end for
  undo the modifications, break, and reject nth VN
end for

Algorithm 3 Mapping mapper virtual nodes of nth VN.

Input: Mapping outcome of Algorithm 1 and 2
for all unallocated $\dot{v}_n^k$ such that $k \in \dot{V}_n$ **do**
  for all $\bar{v}_s^i$ from top of S_L_2 **do**
    if $\hat{C'}_c(\dot{v}_n^k) \geq \hat{C}_c^{\dot{m}}(\dot{v}_n^k)$ **and** $\hat{C'}_c(\dot{v}_n^k) + \hat{C}_c^{\dot{m}}(\dot{v}_n^k) \neq 0$ **then**
      if $\check{C}_c(\bar{v}_s^i) \geq min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$ **then**
        $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$
      else if $\hat{C}_c^{\dot{m}}(\dot{v}_n^k) \leq \check{C}_c(\bar{v}_s^i) < min(\hat{C'}_c(\dot{v}_n^k), \hat{C}_c^{\dot{m}}(\dot{v}_n^k))$ **then**
        $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = \check{C}_c(\bar{v}_s^i)$
      end if
      if $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) > 0$ **then**
        $\check{C}_c(\bar{v}_s^i) = \check{C}_c(\bar{v}_s^i) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
        $\hat{C'}_c(\dot{v}_n^k) = \hat{C'}_c(\dot{v}_n^k) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
        for all adjacent virtual link m of $\dot{v}_n^k$ that its shuffler virtual node is mapped on $v_s^j, j \in \tilde{V}_n$ **do**
          Algorithm4($l_n^{i,j}(m)$), or **Algorithm4**($l_n^{j,i}(m)$)
          if the virtual link mapping was not successful **then** undo the modifications, break, and check the next $\bar{v}_s^i$ in line 3
        end for
      end if
    else if $\hat{C'}_c(\dot{v}_n^k) \neq 0$ **and** $\check{C}_c(\bar{v}_s^i) \geq \hat{C}_c^{\dot{m}}(\dot{v}_n^k)$ **then**
      $\check{C}_c(\dot{v}_n^k, \bar{v}_s^i) = \hat{C}_c^{\dot{m}}(\dot{v}_n^k)$
      $\check{C}_c(\bar{v}_s^i) = \check{C}_c(\bar{v}_s^i) - \check{C}_c(\dot{v}_n^k, \bar{v}_s^i)$
      for all adjacent virtual link m of $\dot{v}_n^k$ that its shuffler virtual node is mapped on $v_s^j, j \in \tilde{V}_n$ **do**
        Algorithm4($l_n^{i,j}(m)$), or **Algorithm4**($l_n^{j,i}(m)$)
        if the virtual link mapping was not successful **then** undo the modifications, break, and check the next $\bar{v}_s^i$ in line 3
      end for
      for all xth v_s^x going backward from $x = i$ to the top of S_L_2 **do**
        if $\check{C}_c(\dot{v}_n^k, \bar{v}_s^x) - \hat{C}_c^{\dot{m}}(\dot{v}_n^k) + \hat{C'}_c(\dot{v}_n^k) \geq \hat{C}_c^{\dot{m}}(\dot{v}_n^k)$ **then**
          $\check{C}_c(\dot{v}_n^k, \bar{v}_s^x) = \check{C}_c(\dot{v}_n^k, \bar{v}_s^x) - \hat{C}_c^{\dot{m}}(\dot{v}_n^k) + \hat{C'}_c(\dot{v}_n^k)$
          $\check{C}_c(\bar{v}_s^x) = \check{C}_c(\bar{v}_s^x) + \hat{C}_c^{\dot{m}}(\dot{v}_n^k) - \hat{C'}_c(\dot{v}_n^k)$
          update the allocated traffic capacities to the corresponding virtual links
          if total reduced allocated processing capacities is equal to $\hat{C}_c^{\dot{m}}(\dot{v}_n^k) - \hat{C'}_c(\dot{v}_n^k)$ **then** update S_L_2, break, and check the next $\dot{v}_n^k$ in line 2
        end if
      end for
    else if $\hat{C'}_c(\dot{v}_n^k) = 0$ **then**
      update S_L_2, break, and check next $\dot{v}_n^k$ in line 2
    end if
  end for
  undo the modifications, break, and reject nth VN
end for

S_L_1. Active switches/routers in S_L_1 have a higher priority than inactive switches/routers. This is because activating an inactive switch/router adds the large amount of base power consumption. Active switches/routers are sorted in ascending order based on their value of $\frac{\tilde{p}^m(\bar{v}_s^i) - \tilde{p}^b(\bar{v}_s^i)}{C_b(\bar{v}_s^i)}$. As a result, the same amount of switching demand adds less amount of power consumption in higher priority switches/routers, according to the defined power model. If the value of this fraction is equal for some switches/routers, we sort them in descending order based on their available switching capacity. A switch/router with the larger amount of available switching capacity may allow us to collocate a larger number of shuffler virtual nodes in it, and therefore allow us to save more energy. On the other hand, we sort inactive switches/routers in ascending order based on their base power consumption. Thus, if it is necessary to activate an inactive switch/router, the lowest possible amount of base power consumption will be required. We sort inactive switches/routers with the equal amount of base power consumption in ascending order based on $\frac{\tilde{p}^m(\bar{v}_s^i) - \tilde{p}^b(\bar{v}_s^i)}{C_b(\bar{v}_s^i)}$, because of the same discussed reason.

Algorithm 4 Virtual link mapping.

Input: $l_n^{i,j}(m)$, the mapping outcome
find K-shortest path from v_s^i to v_s^j in the data center network
for all K found paths with the minimum number of inactive intermediate physical nodes **do**
  for all (x, y) such that $l_s^{x,y}$ is on the alternative path **do**
    if $\check{C}_b(l_s^{x,y}) \geq \check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$ **then**
      $\ddot{d}^{x,y}(l_n^{i,j}(m)) = \check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$
      $\check{C}_b(l_s^{x,y}) = \check{C}_b(l_s^{x,y}) - \ddot{d}^{x,y}(l_n^{i,j}(m))$
    else
      undo the modifications, break, and check the next found path
    end if
  end for
  for all x such that $\tilde{v}_s^x$ is an intermediate physical node on the alternative path **do**
    if $\check{C}_b(\tilde{v}_s^x) \geq 2\check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$ **then**
      $\check{C}_b(\tilde{v}_s^x) = \check{C}_b(\tilde{v}_s^x) - 2\check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$
    else
      undo the modifications, break, and check the next found path
    end if
  end for
  the virtual link is **successfully mapped**
  break from checking the rest of found paths
end for

Algorithm 1 searches for a switch/router candidate $\tilde{v}_s^i$ from the top of S_L_1 to map an unallocated shuffler virtual node $\tilde{v}_n^k$ onto it. A switch/router $\tilde{v}_s^i$ is a candidate if its available switching capacity $\check{C}_b(\tilde{v}_s^i)$ is equal or greater than the switching capacity demand $\hat{C}_b(\tilde{v}_n^k)$ of the shuffler virtual node $\tilde{v}_n^k$. For such a candidate, the algorithm similarly maps every other unallocated shuffler virtual node that is connected to $\tilde{v}_n^k$ with a virtual link, onto a switch/router. In the next step, it calls Algorithm 4 to embed the candidate's adjacent virtual links, between the allocated shuffler virtual nodes, onto substrate paths. We describe Algorithm 4's process later in this section. If the algorithm successfully maps the adjacent shuffler virtual nodes and virtual links of the candidate, then the allocated switching capacity $\check{C}_b(\tilde{v}_n^k, \tilde{v}_s^i)$ to $\tilde{v}_n^k$ in $\tilde{v}_s^i$ is $\hat{C}_b(\tilde{v}_n^k)$. However, if it could not successfully map an unallocated shuffler virtual node onto a switch/router, then it rejects the VN.

6.2. Mapping reducer virtual nodes

In the next step, we run Algorithm 2 to map reducer virtual nodes onto server substrate nodes. The algorithm splits the processing demand $\hat{C}_c(\ddot{v}_n^k)$ of a reducer virtual node $\ddot{v}_n^k$ and map the splitted demands onto servers, while it minimizes the VDC's energy consumption and controls the incast queueing delay. The way it splits the processing demands and map them onto servers has critical impacts on the energy consumption and the incast queueing delay.

In order to reduce the VDC's energy consumption when we map reducer virtual nodes, we intend to use the minimum possible number of servers with the lowest power consumption for demanded processing capacities. In this regard, we sort server substrate nodes in a list called S_L_2. In S_L_2, active servers have a higher priority than inactive servers. Active servers are sorted in ascending order based on $\frac{\tilde{p}^m(\tilde{v}_s^i) - \tilde{p}^b(\tilde{v}_s^i)}{C_c(\tilde{v}_s^i)}$. This ensures that the same amount of processing demand adds less power consumption in servers with a higher priority, according to the defined power model. Servers with the equal value of $\frac{\tilde{p}^m(\tilde{v}_s^i) - \tilde{p}^b(\tilde{v}_s^i)}{C_c(\tilde{v}_s^i)}$ are sorted based on their available processing capacity. A larger amount of available processing capacity in servers may allow us to split processing demands in larger blocks. Therefore, we require a smaller number of servers to allocate them in, and the algorithm is more flexible regarding the incast constraint. Besides, we sort inactive servers in ascending order based on their base power consumption. Thus, if we need to activate an inactive server, the lowest possible amount of base power consumption is needed. Servers with the equal value of base power consumption are sorted in ascending order based on $\frac{\tilde{p}^m(\tilde{v}_s^i) - \tilde{p}^b(\tilde{v}_s^i)}{C_c(\tilde{v}_s^i)}$ for the same discussed reason. Note that we repeatedly update S_L_2 during Algorithm 2's process, upon a reducer virtual node is mapped.

Moreover, we sort reducer virtual nodes in a list called S_L_3, in descending order based on $\frac{\hat{C}_c(\ddot{v}_n^k)}{\hat{C}_c^{\tilde{m}}(\ddot{v}_n^k)}$. $\frac{\hat{C}_c(\ddot{v}_n^k)}{\hat{C}_c^{\tilde{m}}(\ddot{v}_n^k)}$ is the maximum number of distinct servers that may be required for mapping a reducer virtual node $\ddot{v}_n^k$. Algorithm 2 first maps reducer virtual nodes with the highest value of $\frac{\hat{C}_c(\ddot{v}_n^k)}{\hat{C}_c^{\tilde{m}}(\ddot{v}_n^k)}$ to increase the admittance ratio of the network.

In order to use the minimum number of servers with the lowest power consumption, we need to split the processing demand of a reducer virtual node $\ddot{v}_n^k$ into the largest possible processing blocks, and allocate them in the appropriate servers. The algorithm checks the ordered server substrate nodes in S_L_2 and try to allocate the maximum possible processing capacity to an ordered unallocated reducer virtual node $\ddot{v}_n^k$ in S_L_3 in a server. However, the allocated processing capacity $\check{C}_c(\ddot{v}_n^k, \bar{v}_s^i)$ to a reducer virtual node $\ddot{v}_n^k$ in a server substrate node $\bar{v}_s^i$ must satisfy two conditions. First, $\check{C}_c(\ddot{v}_n^k, \bar{v}_s^i)$ must be equal or greater than given $\hat{C}_c^{\tilde{m}}(\ddot{v}_n^k)$ and equal or less than known $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$. Second, $\check{C}_c(\ddot{v}_n^k, \bar{v}_s^i)$ must be greater than the minimum incast processing capacity $C_i(\ddot{v}_n^k, \bar{v}_s^i)$, which guarantees an in range (less than $\hat{D}$) end-to-end queueing delay for incast traffic pattern in the substrate path allocated to a virtual link terminates at $\ddot{v}_n^k$ in $\bar{v}_s^i$.

We assumed a reducer virtual node is connected to the rest of the network via a shuffler virtual node in a VN topology. Besides, shuffler virtual nodes are already mapped in the previous step. Therefore, it is possible to calculate the minimum incast processing capacity $C_i(\ddot{v}_n^k, \bar{v}_s^i)$ when we check the suitability of a server $\bar{v}_s^i$ for a $\ddot{v}_n^k$, analytically. In order to calculate $C_i(\ddot{v}_n^k, \bar{v}_s^i)$, we use Algorithm 4 to find substrate paths that will be allocated to each sub virtual link which connects an allocated shuffler virtual node $\tilde{v}_n^l$ to $\ddot{v}_n^k$. The mean end-to-end incast queueing delay in the longest found substrate path determines $C_i(\ddot{v}_n^k, \bar{v}_s^i)$. $\underline{L}(\ddot{v}_n^k, \bar{v}_s^i)$ denotes the number of physical links in the found longest substrate path. Note that we do not allocate any traffic capacity at this stage. According to M/M/1 queue and Jackson Networks theorem, and because we do not split generated traffic by an allocated virtual node, the minimum required bandwidth capacity that needs to be allocated to the virtual link $l_n^{l,k}$ with the longest substrate path, is:

$$\frac{\underline{L}(\ddot{v}_n^k, \bar{v}_s^i)}{\hat{D}} + \lambda_n^{\mathcal{M}} \tag{17}$$

As it is discussed before, we know this value is equal to $\check{\phi}(\ddot{v}_n^k, \bar{v}_s^i)\hat{C}_b(l_n^{l,k})$. Hence,

$$\begin{aligned}
\check{\phi}(\ddot{v}_n^k, \bar{v}_s^i) &= \frac{\frac{\underline{L}(\ddot{v}_n^k, \bar{v}_s^i)}{\hat{D}} + \lambda_n^{\mathcal{M}}}{\hat{C}_b(l_n^{l,k})}, \\
\frac{C_i(\ddot{v}_n^k, \bar{v}_s^i)}{\hat{C}_c(\ddot{v}_n^k)} &= \frac{\frac{\underline{L}(\ddot{v}_n^k, \bar{v}_s^i)}{\hat{D}} + \lambda_n^{\mathcal{M}}}{\hat{C}_b(l_n^{l,k})}, \\
C_i(\ddot{v}_n^k, \bar{v}_s^i) &= \frac{\hat{C}_c(\ddot{v}_n^k)}{\hat{C}_b(l_n^{l,k})}\left(\frac{\underline{L}(\ddot{v}_n^k, \bar{v}_s^i)}{\hat{D}} + \lambda_n^{\mathcal{M}}\right)
\end{aligned} \tag{18}$$

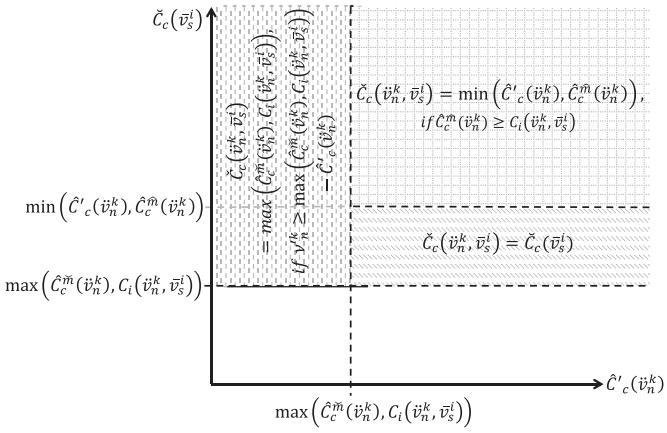

Fig. 2. The heuristic's decision making process.

Furthermore, $\hat{C}'_c(\ddot{v}_n^k)$ is the remained processing capacity that needs to be allocated to $\ddot{v}_n^k$. At the first, $\hat{C}'_c(\ddot{v}_n^k)$ equals to $\hat{C}_c(\ddot{v}_n^k)$. $\hat{C}'_c(\ddot{v}_n^k)$ might be changed, as the algorithm processes the ordered servers for the possible mapping. In this regard, we might face three conditions based on $\hat{C}'_c(\ddot{v}_n^k)$, at each time we check a $\bar{v}_s^i$ to map $\ddot{v}_n^k$ onto it.

First, $\hat{C}'_c(\ddot{v}_n^k)$ is larger than $max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i))$. In this case, allocating the remained processing demand to $\ddot{v}_n^k$ in $\bar{v}_s^i$ does not violate either the minimum requested processing capacity per physical machine $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$, or the maximum tolerable queueing delay $\hat{D}$ in the adjacent virtual links. Therefore, as Fig. 2 shows, we allocate the maximum processing capacity based on the available processing capacity $\check{C}_c(\bar{v}_s^i)$ in the server $\bar{v}_s^i$. If $\check{C}_c(\bar{v}_s^i)$ is equal or greater than $min(\hat{C}'_c(\ddot{v}_n^k), \hat{C}_c^{\hat{m}}(\ddot{v}_n^k))$, then the maximum processing capacity we could allocate to $\ddot{v}_n^k$ in $\bar{v}_s^i$ is the smaller value between $\hat{C}'_c(\ddot{v}_n^k)$ and $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$. This ensures we do not allocate a larger amount than the requested $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$, or $\hat{C}'_c(\ddot{v}_n^k)$. Of course, $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$ must be equal or greater than $C_i(\ddot{v}_n^k, \bar{v}_s^i)$. But, if $\check{C}_c(\bar{v}_s^i)$ is less than $min(\hat{C}'_c(\ddot{v}_n^k), \hat{C}_c^{\hat{m}}(\ddot{v}_n^k))$ and equal or greater than $max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i))$, then the maximum processing capacity we could allocate to $\ddot{v}_n^k$ in $\bar{v}_s^i$ is $\check{C}_c(\bar{v}_s^i)$. So, it does not violate the minimum/maximum requested processing capacity per physical machine, or the maximum tolerable queueing delay $\hat{D}$ in the adjacent virtual links. Otherwise, no processing capacity is allocated to $\ddot{v}_n^k$ in $\bar{v}_s^i$. If the algorithm allocates a processing capacity in a server, then it calls Algorithm 4 to map the corresponding sub virtual links.

Second, $\hat{C}'_c(\ddot{v}_n^k)$ is smaller than $max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i))$. This means if we allocate $\hat{C}'_c(\ddot{v}_n^k)$ to $\ddot{v}_n^k$ in $\bar{v}_s^i$, then either the minimum requested processing capacity per physical machine $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$, or the maximum tolerable queueing delay $\hat{D}$ in its adjacent virtual links, will be violated. Instead, the algorithm allocates $max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i))$ to $\ddot{v}_n^k$ in $\bar{v}_s^i$ and maps its corresponding sub virtual links, if $\bar{v}_s^i$ has enough available processing capacity. $max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i)) - \hat{C}'_c(\ddot{v}_n^k)$ is the extra processing capacity allocated to $\ddot{v}_n^k$ in $\bar{v}_s^i$. Therefore, the algorithm recursively updates the previously allocated processing capacities to $\ddot{v}_n^k$ in servers to decrease the extra allocated capacity, if the constraints (the requested range of processing capacity per physical machine, and the maximum tolerable delay in the adjacent virtual links) will not be violated.

In this regard, when a processing capacity is allocated to a virtual node in a server, we record the amount of processing capacity that if we subtract it from the allocated processing capacity, none of the constraints will be violated. This amount for a $\ddot{v}_n^k$ in a $\bar{v}_s^i$ is equal to $\check{C}_c(\ddot{v}_n^k, \bar{v}_s^i) - max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i))$, and it is represented by $\mathcal{V}_n^k(i)$. $\mathcal{V}_n'^k$ is total $\mathcal{V}_n^k(i)$ for $\ddot{v}_n^k$.

We decrease the previously allocated processing capacities to $\ddot{v}_n^k$ in servers (e.g. $\bar{v}_s^x$) by $min(\mathcal{V}_n^k(x), max(\hat{C}_c^{\hat{m}}(\ddot{v}_n^k), C_i(\ddot{v}_n^k, \bar{v}_s^i)) - \hat{C}'_c(\ddot{v}_n^k))$, until the extra allocated capacity is removed. This is the case if $\mathcal{V}_n'^k$ is equal or greater than the extra allocated capacity. Then, the algorithm updates the allocated traffic capacities to the corresponding sub virtual links.

Third, $\hat{C}'_c(\ddot{v}_n^k)$ is 0. This means $\ddot{v}_n^k$ is mapped successfully.

Algorithm 2 rejects the VN, if it could not successfully map a reducer virtual node and its adjacent virtual links onto servers and substrate paths, respectively.

6.3. Mapping mapper virtual nodes

We run Algorithm 3 as the last step, to embed mapper virtual nodes onto servers. Algorithm 3 works similarly to Algorithm 2. However, there are some minor differences.

Different from the mapping process of a reducer virtual node, we do not concern about the incast queueing delay when we map a mapper virtual node onto servers. Therefore, in contrary to Algorithm 2, Algorithm 3 does not check the minimum incast processing capacity during its allocation process. This simplifies the mapping procedure. As a result, the recursive process in Algorithm 3, which modifies the allocated capacities due to the extra allocated processing capacity, is slightly different from the recursive process in Algorithm 2. The extra allocated processing capacity to a $\ddot{v}_n^k$ in a $\bar{v}_s^i$ in Algorithm 3 is $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k) - \hat{C}'_c(\ddot{v}_n^k)$, where $\hat{C}'_c(\ddot{v}_n^k)$ is the remained processing capacity that needs to be allocated to $\ddot{v}_n^k$. This is because the minimum processing capacity that could be allocated to a $\ddot{v}_n^k$ in a $\bar{v}_s^i$ is only limited by $\hat{C}_c^{\hat{m}}(\ddot{v}_n^k)$. Besides, Algorithm 3 searches in the previously allocated processing capacities to find the one which if it subtracts the extra allocated processing capacity from the already allocated processing capacity, the result does not violate the minimum requested processing capacity per physical machine. Note that $\check{C}_c(\ddot{v}_n^k, \bar{v}_s^i)$ is the allocated processing capacity to $\ddot{v}_n^k$ in $\bar{v}_s^i$.

6.4. Mapping virtual links

We have seen Algorithms 1–3, might call Algorithm 4 to allocate a virtual link or a sub virtual link $l_n^{i,j}(m)$ in a substrate path. In this regard, Algorithm 4 finds K loopless shortest path from v_s^i to v_s^j in a data center network. If source and sink substrate nodes of a substrate link are active, the cost of the substrate link is 1. Otherwise, the substrate link has a large cost, e.g. 100. This helps to find the shortest paths with activating the minimum number of inactive substrate links and nodes. Our preferred routing algorithm to find K loopless shortest paths is the very well known Yen's algorithm [31]. It is also possible to consider more recent methods of finding K loopless shortest paths as proposed in [32,33]. Note that the value of K is adjustable. It will be more probable to find a capable substrate path, by incrementing the value of K. However, incrementing the value of K increases the time complexity of the algorithm, as will be discussed. The right value for K could be chosen according to the size of a data center network.

Algorithm 4 may check the capability of every found substrate path, regarding the required traffic capacity. We know the traffic capacity that needs to be allocated to $l_n^{i,j}(m)$ is equal to $\check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$. Therefore, a substrate path is a candidate if the available traffic capacity $\check{C}_b(l_s^{x,y})$ in every physical link $l_s^{x,y}$ along the path is equal or greater than $\check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$. Besides, the available switching capacity $\check{C}_b(\tilde{v}_s^x)$ in every intermediate substrate node $\tilde{v}_s^x$ along the candidate path also must be equal or grater than $2\check{\phi}(v_n^{a^m}, v_s^i)\check{\phi}(v_n^{b^m}, v_s^j)\hat{C}_b(l_n^{a^m,b^m})$. This considers incoming and outgoing traffic in an intermediate substrate node. Note that a server can not be an intermediate substrate node in our defined data center network topology. The virtual link or the sub virtual link is

successfully mapped onto a data center network, if the algorithm could find such a substrate path for it.

It is required to find the time complexity of the proposed heuristic to see if it is scalable to large network sizes. In this regard, we need to find the complexity of each of the described algorithms for mapping the nth MapReduced-based VN onto a data center network.

We first find the complexity of Algorithm 4, because it is called during the other algorithms. Algorithm 4 calls Yen's algorithm in line 2. The complexity of Yen's algorithm is $\mathcal{O}(K|V_s|(|E_s| + |V_s|log|V_s|)$. Besides, the loop that starts in line 3 and ends in line 21 is run for K times. This loop has two sub-loops. The first sub-loop that starts in line 4 and ends in line 11 is run for every substrate link in the worst-case. The algorithm inside this sub-loop might undo some modifications, that in the worst-case it checks every substrate link again. Hence, the complexity of this sub-loop is $\mathcal{O}(|E_s|^2)$. The second sub-loop that starts in line 12 and ends in line 18 is run for every switch/router substrate node in the worst-case. Inside this sub-loop the algorithm may also undo some modifications. So, it may need to update every substrate link and switch/router again. Thus, the complexity of the second sub-loop is $\mathcal{O}(|\tilde{V}_s|(|E_s| + |\tilde{V}_s|))$. All in all, considering only dominating factors, the time complexity of Algorithm 4 is $\mathcal{O}(K(|V_s|(|E_s| + |V_s|log|V_s|) + |E_s|^2))$.

The complexity of Algorithm 1 is determined by the main loop that starts in line 2 and ends in line 35. This loop is run for every shuffler virtual node in the VN. So, it is run for $|\tilde{V}_n|$ times. A sub-loop of the main loop starts in line 3 and ends in line 33. This sub-loop is run for $|\tilde{V}_s|$ in the worst-case. Besides, we have a sub-loop in Algorithm 1 which starts in line 7 and ends in line 30. This sub-loop maps every adjacent virtual link of a shuffler virtual node. In the worst-case, it is run for $|E_n|$ times. The other sub-loop that starts in line 9 and ends in line 22 may be run for $|\tilde{V}_s|$ times. The algorithm may call Algorithm 4 to map corresponding virtual links. We know the complexity of Algorithm 4 is $\mathcal{O}(K(|V_s|(|E_s| + |V_s|log|V_s|) + |E_s|^2))$. Besides, it might undo some modifications related to the shuffler virtual node and its mapped virtual links. So, the complexity of this function is $\mathcal{O}(|\tilde{V}_s| + |E_s||E_n|)$. Consequently, by taking into account the dominating factors, the time complexity of Algorithm 1 is $\mathcal{O}(|\tilde{V}_n||\tilde{V}_s|^2|E_n|(K|V_s||E_s| + K|V_s|^2log|V_s| + K|E_s|^2 + |E_s||E_n|))$.

The main loop that starts in line 2 and ends in line 30 of Algorithm 2 specifies its time complexity. This loops is run for every reducer virtual node in the VN, which is $|\ddot{V}_n|$ times. A sub-loop that starts in line 4 and ends in line 37 may check the capability of every server substrate node for the reducer virtual node. So, it is run for $|\bar{V}_s|$ times. We have another sub-loop that starts in line 16 and ends in line 19. A similar sub-loop also starts in line 24 and ends in line 27. Both of them are run for $|E_n|$ times in the worst-case. Algorithm 4 is called inside these sub-loops. Besides, the algorithm may undo some modifications inside these sub-loops. Here, the undoing function may check every server substrate node for the allocated processing capacities to the reducer virtual node, and each substate link for the allocated traffic capacities to the virtual links. Therefore, its complexity is $\mathcal{O}(|\bar{V}_s| + |E_s||E_n|)$. The other sub-loop that starts in line 28 and ends in line 33 recursively updates the previously allocated capacities. This sub-loop may be run for $|\bar{V}_s|$ times. The algorithm also updates the previously allocated traffic capacities to the corresponding virtual links inside this sub-loop. The complexity of this function is $\mathcal{O}(|E_s||E_n|)$. Hence, taking into consideration the dominating factors, the time complexity of Algorithm 2 is $\mathcal{O}(|\ddot{V}_n||\bar{V}_s||E_n|(K|\bar{V}_s||E_s| + K|\bar{V}_s|^2log|\bar{V}_s| + K|E_s|^2 + |E_s||E_n|))$.

The time complexity of Algorithm 3 could be derived similarly to the time complexity of Algorithm 2. The only difference is that the main loop in Algorithm 3 is run for every mapper virtual node instead of every reducer virtual node in Algorithm 2. Therefore, it is run for $|\dot{V}_n|$ times. In consequence, the time complexity of Algorithm 3 is $\mathcal{O}(|\dot{V}_n||\bar{V}_s||E_n|(K|\bar{V}_s||E_s| + K|\bar{V}_s|^2log|\bar{V}_s| + K|E_s|^2 + |E_s||E_n|))$.

We know Algorithms 1–3 are run in series to form the heuristic for GreenMap. Thus, the time complexity of the heuristic for mapping the nth VN is $\mathcal{O}((|\tilde{V}_n||\tilde{V}_s|^2|E_n|(K|V_s||E_s| + K|V_s|^2log|V_s| + K|E_s|^2 + |E_s||E_n|)) + (|\ddot{V}_n||\bar{V}_s||E_n|(K|\bar{V}_s||E_s| + K|\bar{V}_s|^2log|\bar{V}_s| + K|E_s|^2 + |E_s||E_n|)) + (|\dot{V}_n||\bar{V}_s||E_n|(K|\bar{V}_s||E_s| + K|\bar{V}_s|^2log|\bar{V}_s| + K|E_s|^2 + |E_s||E_n|)))$. This proves the heuristic could be solved in a polynomial time.

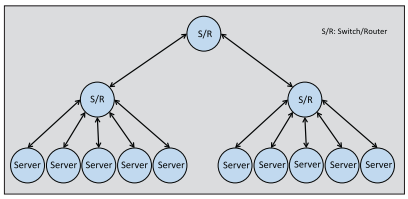

Fig. 3. Topology of the data center network in a small simulation setup.

7. Evaluation

GreenMap is supposed to map the heterogeneous MapReduce-based VNs onto a heterogeneous data center network, and minimize its total energy consumption. Besides, it needs to control the introduced incast queueing delay. We verify the performance of the formulated MIDCP and the proposed heuristic for GreenMap by generating and mapping random MapReduce-based VNs onto a data center network with multi-level topology.

As it is discussed, the formulated MIDCP is $\mathcal{NP}$-hard, so it is not scalable for large network sizes. Therefore, similar to the other related works in [15,16,28,34,35], we assess capability of the MIDCP on small random simulation setups. It is possible to solve Mixed-Integer Disciplined Convex Programs by combination of a continuous optimization algorithm and an exhaustive search method [29]. We solved the formulated MIDCP by MOSEK solver [36], with the tolerance of 1.49×10^{-8}. Nonetheless, the theoretical complexity analysis reveals that the proposed heuristic algorithm is much simpler, and therefore it is scalable for large network sizes. Hence, the performance of the suggested heuristic is examined on large random simulation setups. The setups for the proposed heuristic algorithm are simulated using MATLAB.

A small random simulation setup includes a heterogeneous data center network with a symmetric tree topology. Fig. 3 shows the topology of this network. It has 13 nodes including 10 servers and 3 switches/routers. We assume servers are blades with a CPU capacity of 2 GHz. The blade server is ideal for our study, because it is widely deployed in data centers and incorporates several power management techniques [27]. The base power consumption of a blade server is 213Watt, and its maximum power consumption is 319.5 W [27]. It is assumed a switch/router has 10Gbps switching capacity, and a physical link has 1Gbps bandwidth capacity. In this case, the base power consumption of a physical link is 1.7 W, and its maximum power consumption is 2 W [28].

Recently, Waxman algorithm [37] is widely used by the researchers to generate random virtual network topologies [15,16,35]. Therefore, in this paper, virtual networks' topologies are generated by Waxman algorithm. Waxman generates random network topologies based on two parameters. As the first parameter grows, the probability of having an edge between a pair of nodes in the topology is increased. We choose 0.4 for the first parameter. As the second parameter grows there is a larger ratio of long edges to short edges. We choose 0.2 for the second parameter.

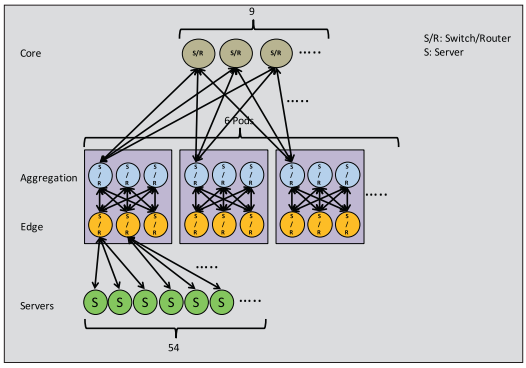

Fig. 4. Abstract topology of the data center network in a large simulation setup.

In small random simulation setups, a virtual link's bandwidth demand is generated randomly between 100Mbps and 200Mbps, following the uniform distribution. The switching demand of a shuffler virtual node is assumed to be equal to the summation of all its incoming and outgoing virtual links' bandwidth demands. Moreover, $\lambda(i_n^k)$ of a mapper virtual node i_n^k is generated with the uniform distribution between 1Mbps and 10Mbps. We also assume $\hat{D}$ is 25 ms. The round-trip times (RTTs) in data center networks are in the range of hundreds of microsecond [7]. Therefore, we assume the delay demands in the same range through this paper. Note that no restrictions is considered for the minimum/maximum processing capacity per physical machine for a computation-based virtual node, unless otherwise stated.

A large random simulation setup includes a heterogeneous data center network with a fat-tree topology. Fig. 4 shows the abstract topology of this network. Here, we consider a 6-ary fat-tree topology with three layers of edge, aggregation, and core. This topology is built by 6-port commodity switches/routers. It has 6 pods, each contains two layers (edge/aggregation) of 3 switches/routers. Every 6-port edge switch/router is connected to 3 servers, and 3 aggregation switches/routers. We also have 9 core switches/routers. Each 6-port aggregation switch/router is connected to 3 edge switches/routers, and 3 core switches/routers. Every core switch/router is connected to each of 6 pods. The ith port of a core switch/router is connected to the ith pod. Thus, this topology has 54 servers, and 45 switches/routers. Similar to a small random simulation setup, servers are blades with a CPU capacity of 2 GHz. It is assumed a switch/router has 6 $\times$ 10Gbps switching capacity, and a physical link has 10Gbps bandwidth capacity. In this case, the base and maximum power consumption of a physical link are assumed to be 17 W and 20 W, respectively [38].

In a large random simulation setup, a virtual link's bandwidth demand is generated randomly between 300Mbps and 500Mbps, following the uniform distribution. Besides, $\hat{D}$ is assumed to be 50 ms. The rest of the configurations for large random simulation setups are the same as the small random simulation setups, unless otherwise instructed.

7.1. Small random simulation setups

First, we solved the MIDCP and the heuristic for GreenMap, and also the state-of-the-art energy-efficient VN embedding algorithm in [15], for different numbers of virtual nodes per VN, on a small random simulation setup. We measured the total power consumption by servers in all the cases. There is no existing approach that could split virtual nodes and embed heterogeneous MapReduce-based VNs onto a heterogeneous data center network. So, we compared our solution with the well-cited energy-efficient VN embedding algorithm in [15], which does not allow node splitting. The number of virtual nodes per VN is ranged from 5 to 8, while each VN has 3 shuffler virtual nodes. It also has at least 1 mapper and 1 reducer virtual node, but the exact numbers are chosen randomly following the uniform distribution. We tested 10 randomly generated virtual networks for each number of virtual nodes per VN, and plotted the average results including confidence intervals with the confidence level of 90%, in Fig. 5a.

In this simulation setup, the CPU demand of a computation-based virtual node is a random value between 500 MHz and 1500 MHz, following the uniform distribution. Note that this range is chosen, so the state-of-the-art algorithm is able to map a computation-based virtual node onto a single server substrate node.

The results in Fig. 5a first confirm that the MIDCP and the heuristic for GreenMap effectively reduce total power consumption by servers, in comparison to the state-of-the-art energy-efficient mapping algorithm. This is because the state-of-the-art algorithm does not allow either node splitting, or node collocation. Second, the results show incrementing the number of virtual nodes per VN, increases the total consumed power by servers in the data center network. This is because we have fixed number of shuffler virtual nodes and increasing number of computation-based virtual nodes, while a computation-based virtual node needs some power to operate. Besides, the power consumption is increasing linearly in the case of state-of-the-art algorithm, since it maps each virtual node on a single substrate node. However, this is not the case for the MIDCP and the heuristic for GreenMap, as they splits the processing demands of computation-based virtual nodes and might map them onto multiple virtual nodes. They also may collocate multiple virtual nodes of a VN in a single substrate node. Moreover, Fig. 5a verifies that the heuristic could achieve reasonably close to the optimum results of the MIDCP. This is the case while the heuristic is considerably faster than the MIDCP. For example, in the same setup and for a single run, when we have 6 virtual nodes in a VN, the MIDCP's run time is 84,128 s. However, this amount for the heuristic is only 0.0414 s.

In addition, we solved the formulated MIDCP for GreenMap for different CPU demands of a computation-based virtual node, on a small random simulation setup. This is also solved for when the incast constraints in Eq. (15) are relaxed. We measured the mean incast queueing delay in the allocated substrate paths to virtual links that terminate at a reducer virtual node, for both cases. The CPU demands are ranged from 2000 MHz to 3400 MHz. We tested 10 randomly generated virtual networks for each CPU rate, and plotted the average results including confidence intervals with the confidence level of 90%, in Fig. 5b.

In this simulation setup, we have 5 virtual nodes per VN, with the minimum of 1 mapper, 1 reducer, and 1 shuffler virtual node. The exact number is chosen randomly according to the uniform distribution. Moreover, we assume that for a mapper virtual node $\dot{v}_n^k$, $\frac{\hat{C}_c^{\dot{m}}(\dot{v}_n^k)}{\hat{C}_c(\dot{v}_n^k)} = 0.5$ and $\frac{\hat{C}_c^{\dot{m}}(\dot{v}_n^k)}{\hat{C}_c(\dot{v}_n^k)} = 1$. Besides, for a reducer virtual node $\ddot{v}_n^k$, $\frac{\hat{C}_c^{\ddot{m}}(\ddot{v}_n^k)}{\hat{C}_c(\ddot{v}_n^k)} = 0$ and $\frac{\hat{C}_c^{\ddot{m}}(\ddot{v}_n^k)}{\hat{C}_c(\ddot{v}_n^k)} = 1$.

According to the results in Fig. 5b, the mean incast queueing delay is always less than the determined maximum tolerable delay $\hat{D}$ of 25 ms. This confirms that the MIDCP for GreenMap could control the incast queueing delay, effectively. Nevertheless, simulation results show the mean incast queueing delay of *infinity*, for any CPU demand in the range, when we relax the incast constraints in Eq. (15). This means for each case, at least one queue over the allocated substrate path to a virtual link that terminates at an allocated reducer virtual node, is unstable. So, the queue's service mean rate is less than its arrival rate. Hence, in the case we do not control the introduced incast queueing delay, providers might not catch their latency targets for the individual MapReduce tasks. This also may result in violation of their Service Level Agreements (SLAs).

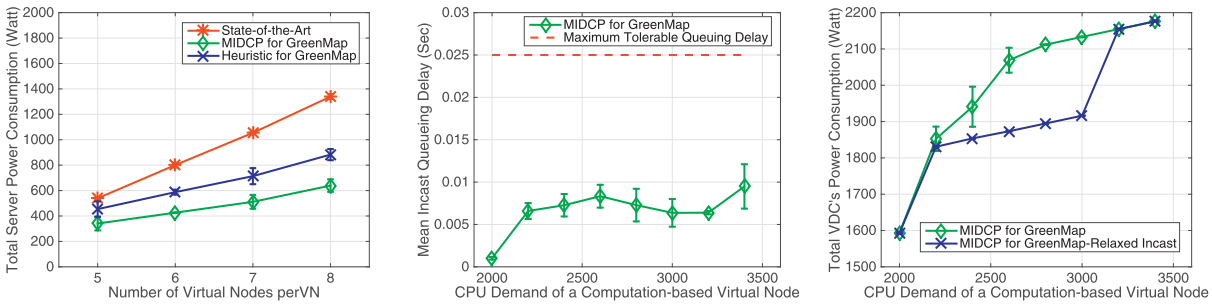

Fig. 5. (a) The total power consumption based on different numbers of virtual nodes per VN for the state-of-the-art algorithm, the MIDCP for GreenMap, and the heuristic for GreenMap. (b) The mean incast queueing delay based on different CPU demands of a computation-based virtual node for the MIDCP, as well as the chosen maximum tolerable queueing delay $\hat{D}$ for a virtual link. (c) The total VDC's power consumption based on different CPU demands of a computation-based virtual node for the MIDCP, and the MIDCP when the incast constraints are relaxed. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Note that the incast queueing delay is fluctuating by changing the CPU demand. In order to satisfy the objective and the constraints, different CPU demands of a computation-based virtual node might be splitted and allocated in server substrate nodes, differently. Since the incast queueing delay is related to the assigned CPU capacity to reducer virtual nodes, it fluctuates by changing the CPU demand.

In the same simulation setup, we also measured the total consumed power by servers, switches/routes, and physical links. The results are shown in Fig. 5c. The figure shows that increasing the demanded CPU rate of computation-based virtual nodes, increases the total power consumption. This is because we need to allocate more processing/bandwidth capacities in the data center network in order to handle the higher demands.

But, the interesting point about the results in Fig. 5c is that for some demanded CPU rates, the total power consumption is higher when we control the incast queueing delay. In other words, sometimes we could save more power if we relax the incast constraints. According to Section 5, the way we split and map the CPU demands of reducer virtual nodes onto server substrate nodes directly impacts the incast queueing delay. Hence, when we limit the incast queueing delay by setting the maximum tolerable queueing delay for a virtual link, we are not able to use every available processing/bandwidth capacity in the network to save energy. For example, we might have to turn an inactive server on to handle a CPU demand and keep the incast queueing delay in the requested range. However, if we do not concern about the incast queueing delay, we may be able to split the CPU demand into smaller blocksand allocate them in active servers to save the energy.

This issue does not happen when we have the enough processing capacity in active servers, so splitting the CPU demands does not violate the incast constraint. Besides, the total power consumption in the case of relaxed incast constraints might change linearly for a range of CPU demands, as we could split and allocate them in active servers without turing on inactive servers.

7.2. Large random simulation setups

The simulation results in Fig. 5a confirmed that the heuristic reduces the total server power consumption of the small scale VDCs, effectively. In order to verify the effectiveness of the heuristic regarding saving the energy in large scale data centers, we tested the heuristic as well as the state-of-the-art algorithm on a large random simulation setup, for different numbers of virtual nodes per VN. The number of virtual nodes per VN is ranged from 12 to 18. Here, a VN has 10 shufflers, and at least 1 mapper and 1 reducer virtual node. The exact number of mapper and reducer virtual nodes are chosen randomly based on the uniform distribution. The CPU demand of a computation-based virtual node is a random value between 500 MHz and 1500 MHz, following the uniform distribution. Note that this range is chosen, so the state-of-the-art algorithm is able to map a computation-based virtual node onto a single server substrate node. We examined 10 randomly generated virtual networks for each number of virtual nodes per VN, and plotted the average results including confidence intervals with the confidence level of 90%, in Fig. 6a.

The results in Fig. 6a prove that the heuristic for GreenMap significantly reduces the large scale VDC's total power consumption in comparison to the state-of-the-art algorithm. Besides, the results show that the heuristic for GreenMap saves much larger amounts of power in the large random simulation setup in comparison to the small simulation setup in Fig. 5a. This means GreenMap saves the energy in large network sizes more effectively than small network sizes. This is because a larger number of computation-based virtual nodes could be splitted and mapped and/or collocated in servers. Besides, a larger number of shuffler virtual nodes could be collocated in switches/routers. This decreases the number of active substrate elements remarkably, in comparison to the state-of-the-art algorithm that maps each virtual node onto a single substrate node.

Furthermore, we examined the heuristic on a large random simulation setup, and measured the mean incast queueing delay in the allocated substrate paths to virtual links that terminate at a reducer virtual node. This measurement is performed for different numbers of virtual nodes per VN. The number of virtual nodes per VN is ranged from 12 to 28. A VN has 10 shufflers, and at least 1 mapper and 1 reducer virtual node. The exact number of mapper and reducer virtual nodes are chosen randomly based on the uniform distribution. The CPU demand of a computation-based virtual node is chosen randomly between 2000 Mhz and 3400 MHz, according to the uniform distribution. We tested 10 randomly generated virtual networks for each number of virtual nodes per VN, and plotted the average results including confidence intervals with the confidence level of 90%, in Fig. 6b.

Fig. 6b confirms that heuristic effectively controls the mean incast queueing delay, and it is always less than the defined maximum tolerable queueing delay of 50 ms. Note that the mean incast queueing delay is fluctuating by changing the number of virtual nodes per VN. This is because the algorithm might split and map the computation-based virtual nodes differently, in order to satisfy the constraints. Because the incast queueing delay is related to the assigned CPU capacity to reducer virtual nodes, it fluctuates by changing the number of virtual nodes per VN.

Moreover, in another large random simulation setup, we tested the admittance ratio of the network for different values of $\hat{D}$. The admittance ratio is the number of accepted and mapped VNs, di-

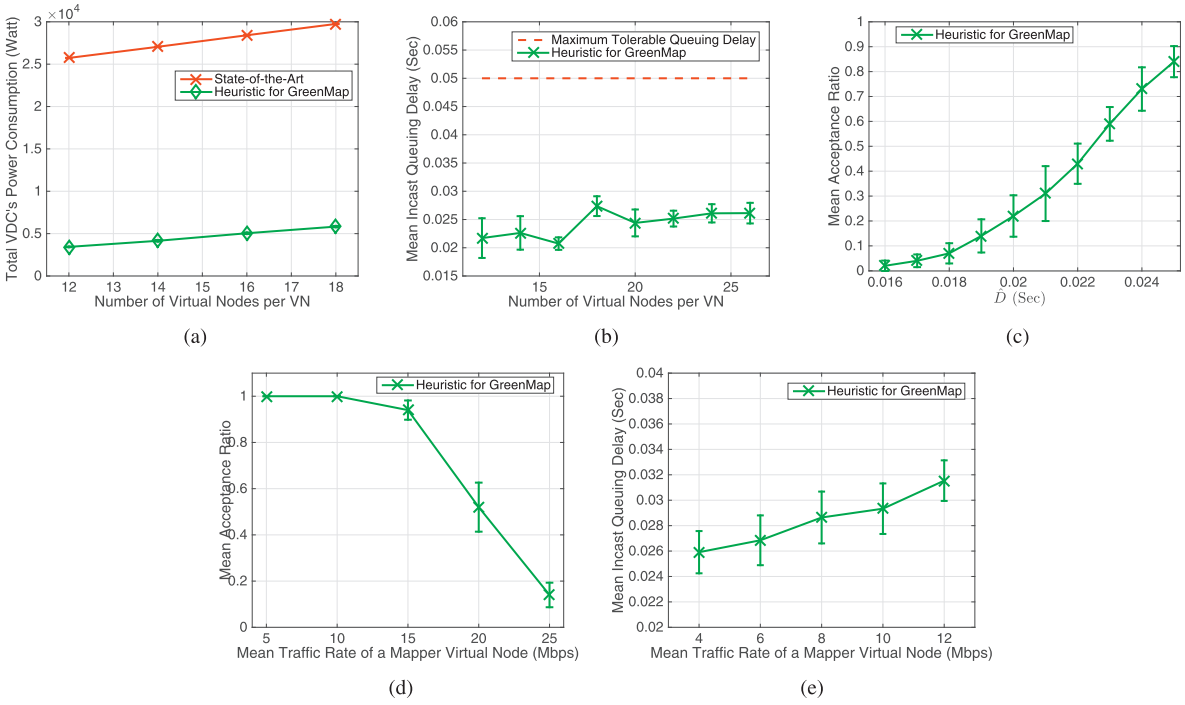

Fig. 6. (a) The total power consumption of the VDC based on different numbers of virtual nodes per VN for the state-of-the-art algorithm, and the heuristic for GreenMap. (b) The mean incast queueing delay based on different numbers of virtual nodes per VN for the heuristic for GreenMap, as well as the maximum tolerable queueing delay $\hat{D}$ for a virtual link. (c) The mean acceptance ratio based on different values of $\hat{D}$ for the heuristic for GreenMap. (d) The mean acceptance ratio based on different mean traffic rates of a mapper virtual node for the heuristic for GreenMap. (e) The mean incast queueing delay based on different mean traffic rates of a mapper virtual node for the heuristic for GreenMap. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

vided by the total number of received VN requests. We considered the range of 15 ms to 25 ms for $\hat{D}$. We examined 10 randomly generated scenarios. In each scenario, the network receives 10 randomly generated virtual networks for every defined $\hat{D}$. Here, a virtual network has 10 shuffler virtual nodes, 5 mapper virtual nodes, and 5 reducer virtual nodes. The CPU demand of a computation-based virtual node is chosen randomly between 2000 MHz and 3400 MHz, according to the uniform distribution. The average results including confidence intervals with the confidence level of 90%, is plotted in Fig. 6c.

Fig. 6c demonstrates that the mean acceptance ratio is increasing by increasing the value of $\hat{D}$. The smaller values of $\hat{D}$ enforce the heuristic to split the processing demands of reducer virtual nodes in relevantly larger processing blocks, and map them onto servers. Allocating larger processing capacities in servers leaves smaller available processing capacities in them for new VN requests. Therefore, it is more probable that the heuristic could not map a new VN onto the data center network successfully, for smaller values of $\hat{D}$. This results in a lower network's acceptance ratio for smaller values of $\hat{D}$.

In the same simulation setup, for when $\hat{D}$ is 50 ms, we probed the acceptance ratio for different mean traffic rates of a mapper virtual node. The results are shown in Fig. 6d. We assume the same mean traffic rate for all mapper virtual nodes. The mean traffic rate of a mapper virtual node is ranged from 5Mbps to 25Mbps. According to Eq. (18), incrementing the mean traffic rate of mapper virtual nodes and therefore increasing λ_n^M, increases the minimum amount of processing capacity the heuristic must allocate to a reducer virtual node to control the incast queueing delay. This causes smaller available processing capacities in servers for new VNs, and accordingly reduces the acceptance ratio.

Moreover, in the previous simulation setup, we also measured the mean incast queueing delay in the allocated substrate paths to the virtual links that terminate at a reducer virtual node. The results are demonstrated in Fig. 6e. Different from the previous simulation setup, here we considered the range of 4Mbps–12Mbps for the mean traffic rate of a mapper virtual node. This range is chosen, so no VN is rejected. As the results confirm, increasing the mean traffic rate of mapper virtual nodes, increases the mean incast queueing delay. This is because the difference between the mean service rate and the mean traffic rate in an allocated traffic capacity to the virtual links in substrate links is decreased. Therefore, according to M/M/1 queue, the mean incast queueing delay is increased.

Note that every simulation setup is quite large to cover a substantial number of random virtual networks in order to verify the effectiveness of the proposed solutions. Besides, the calculated confidence intervals confirm that the results are precise enough to reveal significances of GreenMap.

8. Conclusion

Saving energy in today's data centers is a key challenge. On the other hand, data centers are moving toward virtualized data centers. In this paper, we proposed GreenMap, a novel energy-efficient embedding method that maps heterogeneous MapReduce-based virtual networks onto a heterogeneous data center network. Moreover, for the first time, we introduced a new incast problem that specially may happen in VDCs. GreenMap also controls the incast queueing delay. A MIDCP is formulated and a novel heuristic is suggested for GreenMap. Simulation results prove that both of the solutions for GreenMap could map heterogeneous MapReduce-based VNs onto a heterogeneous data center network, and reduce a VDC's total consumed energy, effectively. It is also confirmed both of the MIDCP and the heuristic for GreenMap control the introduced incast queueing delay. As a future work, it would be helpful to show the comparison of our proposed solutions with the available mapping methods that allow virtual node collocation.

Appendix A

Considering the continuous function $f(g,h)=\frac{g^2}{h}$, $h>0$, the hessian matrix H of f is shown in Eq. (19). Eigenvalues of H are 0 and $\frac{2g^2+2h^2}{h^3}$. Because all eigenvalues of H are non-negative, H is positive semi-definite. Therefore, f is jointly convex on both g and h.

$$H=\begin{bmatrix} \frac{2}{h} & \frac{-2g}{h^2} \\ \frac{-2g}{h^2} & \frac{2g^2}{h^3} \end{bmatrix} \tag{19}$$

Appendix B

The following linear constraints force $\dot{d}_n^{x,y}(m)$ to take the value of $\check{\phi}(v_n^{a^m},v_s^x)\check{\phi}(v_n^{b^m},v_s^y)$.

$$\begin{aligned}
&\dot{d}_n^{x,y}(m)\le\check{\phi}(v_n^{a^m},v_s^x)\,,\\
&\dot{d}_n^{x,y}(m)\le\check{\phi}(v_n^{b^m},v_s^y)\,,\\
&\dot{d}_n^{x,y}(m)\ge\check{\phi}(v_n^{a^m},v_s^x)+\check{\phi}(v_n^{b_m},v_s^y)-1\,,\\
&\dot{d}_n^{x,y}(m)\ge 0\,,\\
&\forall x\in V_s,\forall y\in V_s,\forall n\in\{n|G_n\in\Phi\},m=1,\ldots,L_n
\end{aligned} \tag{20}$$

Appendix C

The following linear constraints force $\ddot{d}^{i,j}(l_n^{x,y}(m))$ to take the value of $z^{i,j}(l_n^{x,y}(m))\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})$. Note that M is the largest virtual link bandwidth demand in Φ.

$$\begin{aligned}
&\ddot{d}^{i,j}(l_n^{x,y}(m))\le Mz^{i,j}(l_n^{x,y}(m))\,,\\
&\ddot{d}^{i,j}(l_n^{x,y}(m))\le\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})\,,\\
&\ddot{d}^{i,j}(l_n^{x,y}(m))\ge\dot{d}_n^{x,y}(m)\hat{C}_b(l_n^{a^m,b^m})-M(1-z^{i,j}(l_n^{x,y}(m)))\,,\\
&\\
&\ddot{d}^{i,j}(l_n^{x,y}(m))\ge 0\,,\\
&\forall(i,j)\in E_s,\forall x\in V_s,\forall y\in V_s,\forall n\in\{n|G_n\in\Phi\}\,,\\
&m=1,\ldots,L_n
\end{aligned} \tag{21}$$

References

[1] J. Baliga, R.W. Ayre, K. Hinton, R. Tucker, Green cloud computing: balancing energy in processing, storage, and transport, Proc. IEEE IEEE 99 (1) (2011) 149–167.
[2] L.A. Barroso, U. Hlzle, The case for energy-proportional computing, Comput. IEEE 40 (12) (2007) 33–37.
[3] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: 6th Conference on Symposium on Operating Systems Design Implementation, ACM, 2004, p. 10.
[4] W. Li, H. Yang, Z. Luan, D. Qian, Energy prediction for mapreduce workloads, in: Ninth International Conference on Dependable, Autonomic and Secure Computing (DASC), IEEE, 2011, pp. 443–448.
[5] M.F. Zhani, Q. Zhang, G. Simon, R. Boutaba, Vdc planner: dynamic migration-aware virtual data center embedding for clouds, in: International Symposium on Integrated Network Management (IM), IFIP/IEEE, 2013, pp. 18–25.
[6] D. Nagle, D. Serenyi, A. Matthews, The panasas activescale storage cluster: delivering scalable high bandwidth storage, in: Conference on Supercomputing, ACM/IEEE, 2004, p. 53.
[7] J. Hwang, J. Yoo, N. Choi, Deadline and incast aware tcp for cloud data center networks, Comput. Netw. Elsevier 68 (2014) 20–34.
[8] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, M. Sridharan, Data center tcp (dctcp), Comput. Commun. Rev. ACM SIGCOMM 41 (4) (2011) 63–74.
[9] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: measurements and analysis, in: 9th Conference on Internet Measurement Conference, ACM SIGCOMM, 2009, pp. 202–208.
[10] H. Wu, Z. Feng, C. Guo, Y. Zhang, Ictcp: incast congestion control for tcp in data-center networks, Transac. Netw. (TON) IEEE/ACM 21 (2) (2013) 345–358.
[11] Y. Ren, Y. Zhao, P. Liu, K. Dou, J. Li, A survey on tcp incast in data center networks, Int. J. Commun. Syst. 27 (8) (2014) 1160–1172. Wiley Online Library
[12] A. Fischer, M.T. Beck, H.D. Meer, An approach to energy-efficient virtual network embeddings, in: International Symposium on Integrated Network Management (IM), IFIP/IEEE, 2013, pp. 1142–1147.
[13] S. Su, Z. Zhang, X. Cheng, Y. Wang, Y. Luo, J. Wang, Energy-aware virtual network embedding through consolidation, in: Computer Communications Workshops (INFOCOM WKSHPS), IEEE, 2012, pp. 127–132.
[14] B. Wang, X. Chang, J. Liu, J.K. Muppala, Reducing power consumption in embedding virtual infrastructures, in: Globecom Workshops (GC Wkshps), IEEE, 2012, pp. 714–718.
[15] J.F. Botero, X. Hesselbach, M. Duelli, D. Schlosser, A. Fischer, H.D. Meer, Energy efficient virtual network embedding, Commun. Lett. IEEE 16 (5) (2012) 756–759.
[16] J.F. Botero, X. Hesselbach, Greener networking in a network virtualization environment, Comput. Netw. Elsevier 57 (9) (2013) 2021–2039.
[17] C. Guo, G. Lu, H.J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, Y. Zhang, Secondnet: a data center network virtualization architecture with bandwidth guarantees, in: 6th International Conference on Emerging Networking Experiments and Technologies (CoNEXT), ACM, 2010, p. 15.
[18] M. Chowdhury, M.R. Rahman, R. Boutaba, Vineyard: virtual network embedding algorithms with coordinated node and link mapping, Transac. Netw. (TON) IEEE/ACM 20 (1) (2012) 206–219.
[19] Y. Zhu, M.H. Ammar, Algorithms for assigning substrate network resources to virtual network components., in: INFOCOM, IEEE, 2006, pp. 1–12.
[20] M. Yu, Y. Yi, J. Rexford, M. Chiang, Rethinking virtual network embedding: substrate support for path splitting and migration, Comput. Commun. Rev. ACM 38 (2) (2008) 17–29.
[21] N.M.K. Chowdhury, M.R. Rahman, R. Boutaba, Virtual network embedding with coordinated node and link mapping, in: INFOCOM, IEEE, 2009, pp. 783–791.
[22] A. Phanishayee, E. Krevat, V. Vasudevan, D.G. Andersen, G.R. Ganger, G.A. Gibson, S. Seshan, Measurement and analysis of tcp throughput collapse in cluster-based storage systems., in: Conference on File and Storage Technologies, ACM, 8, 2008, pp. 1–14.
[23] E. Krevat, V. Vasudevan, A. Phanishayee, D.G. Andersen, G.R. Ganger, G.A. Gibson, S. Seshan, On application-level approaches to avoiding tcp throughput collapse in cluster-based storage systems, in: 2nd International Workshop on Petascale Data Storage: Held in Conjunction with Supercomputing'07, ACM, 2007, pp. 1–4.
[24] Y. Chen, R. Griffith, J. Liu, R.H. Katz, A.D. Joseph, Understanding tcp incast throughput collapse in datacenter networks, in: The 1st Workshop on Research on Enterprise Networking, ACM, 2009, pp. 73–82.
[25] K. Qi, Z. Zhao, J. Fang, Y. Han, Mapreduce-based data stream processing over large history data, in: International Conference on Service-Oriented Computing, Springer, 2012, pp. 718–732.
[26] X. Fan, W.-D. Weber, L.A. Barroso, Power provisioning for a warehouse-sized computer, in: ACM SIGARCH Computer Architecture News, ACM, 35, 2007, pp. 13–23.
[27] D. Economou, S. Rivoire, C. Kozyrakis, P. Ranganathan, Full-system power analysis and modeling for server environments, in: Workshop on Modeling Benchmarking and Simulation (MOBS), 2006, pp. 13–23.
[28] A.P. Bianzino, C. Chaudet, F. Larroca, D. Rossi, J. Rougier, Energy-aware routing: a reality check, in: GLOBECOM Workshops (GC Wkshps), IEEE, 2010, pp. 1422–1427.
[29] M. Grant, S. Boyd, Y. Ye, Disciplined Convex Programming, Springer, 2006.
[30] N. Chowdhury, R. Boutaba, A survey of network virtualization, Comput Netw. Elsevier 54 (5) (2010) 862–876.
[31] J.Y. Yen, Finding the k shortest loopless paths in a network, Manage. Sci. 17 (11) (1971) 712–716.
[32] H. Aljazzar, S. Leue, K?: a heuristic search algorithm for finding the k shortest paths, Artif. Intell. Elsevier 175 (18) (2011) 2129–2154.
[33] J. Hershberger, M. Maxel, S. Suri, Finding the k shortest simple paths: a new algorithm and its implementation, Transac. Algorithms (TALG) ACM 3 (4) (2007) 45.
[34] E. Ghazisaeedi, N. Wang, R. Tafazolli, Link sleeping optimization for green virtual network infrastructures, in: Globecom Workshops (GC Wkshps), IEEE, 2012, pp. 842–846.
[35] E. Ghazisaeedi, C. Huang, Off-peak energy optimization for links in virtualized network enviornment, Transac. Cloud Comput. (TCC) IEEE PP (99) (2015).
[36] E.D. Andersen, K.D. Andersen, The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm, High Performance Optimization, Springer, pp. 197–232.
[37] B.M. Waxman, Routing of multipoint connections, Sel. Areas Commun. IEEE 6 (9) (1988) 1617–1622.
[38] S. Ricciardi, D. Careglio, U. Fiore, F. Palmieri, G. Santos-Boada, J. Sol-Pareta, Analyzing local strategies for energy-efficient networking, in: NETWORKING 2011 Workshops, Springer, 2011, pp. 291–300.

Ebrahim Ghazisaeedi received his Ph.D. degree in Electrical and Computer Engineering from Carleton University, Canada, in 2015. He also received his M.Sc. degree in Mobile and Satellite Communications from the University of Surrey, England, in 2011. His main research interests are in communication networks, network virtualization, and network optimization.

Changcheng Huang received his Ph.D. degree in Electrical Engineering from Carleton University, Canada, in 1997. Since July 2000, he has been with the Department of Systems and Computer Engineering at Carleton University, where he is currently a professor. His research interests are stochastic control in computer networks, resource optimization in wireless networks, reliability mechanisms for optical networks, network protocol design and implementation issues.