Dependent Function Embedding for Distributed Serverless Edge Computing

Shuiguang Deng, *Senior Member, IEEE,* Hailiang Zhao, Zhengzhe Xiang, *Member, IEEE,* Cheng Zhang, Rong Jiang, Ying Li, Jianwei Yin, Schahram Dustdar, *Fellow, IEEE*, and Albert Y. Zomaya, *Fellow, IEEE*

Abstract—Edge computing is booming as a promising paradigm to extend service provisioning from the centralized cloud to the network edge. Benefit from the development of serverless computing, an edge server can be configured as a carrier of limited serverless functions, in the way of deploying Docker runtime and Kubernetes engine. Meanwhile, an application generally takes the form of directed acyclic graphs (DAGs), where vertices represent dependent functions and edges represent data traffic. The status quo of minimizing the completion time (a.k.a. makespan) of the application motivates the study on optimal function placement. However, current approaches lose sight of proactively splitting and mapping the traffic to the logical data paths between the heterogeneous edge servers, which could affect the makespan significantly. To remedy that, we propose an algorithm, termed as Dependent Function Embedding (DPE), to get the optimal edge server for each function to execute and the moment it starts executing. DPE finds the best segmentation of each data traffic by exquisitely solving several infinity norm minimization problems. DPE is theoretically verified to achieve the global optimality. Extensive experiments on Alibaba cluster trace show that DPE significantly outperforms two baseline algorithms in makespan by 43.19% and 40.71%, respectively.

Index Terms—Edge computing, dependent function embedding, directed acyclic graph, function placement, task scheduling.

1 Introduction

In recent years, the micro-services application architecture has achieved rapid advances. By changing applications from monolithic to small pieces of code as functions, Function-as-a-Service (FaaS) is leading its way to the future service pattern of cloud computing [1] [2]. Combing FaaS with lightweight containerization and service orchestration tools, such as the Docker runtime [3] and the Kubernetes engine [4], the concept of serverless computing becomes increasingly popular [5]. Serverless computing offers a platform that allows the execution of software without providing any notion of the underlying computing clusters, operating systems, VMs or containers [6]. It is a one step forward in the abstraction staircase from Infrastructure-as-a-Service (IaaS) to Platform-as-a-Service (PaaS) [2].

Meanwhile, with more and more applications offloaded to remote cloud data-centers, it is hard to meet the QoS requirements of latency-sensitive applications [7]. To mitigate latency, near-data processing within the network edge is a more applicable way to gain insights, which leads to the birth of edge computing. Generally, edge computing refers to leveraging the computation and communication enabled servers, located at the network edge, to make quick response to mobile and IoT applications [8] [9]. Edge servers can be co-located with telecommunication equipment in multiple places across radio access networks (RAN) to the core network (5GC), in the way of small-scale data-centers or machine rooms [10].

Demonstrating common features with the requirements of Internet of Things (IoT) applications, the adaptation of serverless in edge computing has attracted special attention from both industrial community and academia, leading to the birth of serverless edge computing [11]. The paradigm of serverless edge computing allows users to execute their differentiated applications without managing the underlying servers and clusters. Serverless has proved to be more cost-efficient and user-friendly compared with a traditional IaaS architecture in many pilot projects for edge computing [2]. Nevertheless, serverless edge computing faces a series of problems need to be solved urgently. One of the problems restricting its development concerns the scheduling of the functions with complex inter-task dependency to the resource-constrained edge [11] [12] [13]. To address this issue, works studying *optimal function placement* across the heterogeneous edge servers are conducted [14] [15] [16]. In these works, applications are structured as a service function chain (SFC) or a directed acyclic graph (DAG) composed of dependent functions, and the placement of each function is obtained by minimizing the makespan of the application, under the trade-off between function processing time and cross-server data transferring overhead.

However, when minimizing the makespan of the application, state-of-the-art approaches only optimize the placement of functions, but how the request to call the function's successors being routed and how the corresponding data stream being mapped onto the virtual links between edge servers are ignored [14] [15]. Actually, routing and manage-

- *S. Deng, H. Zhao, C. Zhang, Y. Li, and J. Yin are with the College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China. e-mail: {dengsg, hliangzhao, coolzc, cnliying, zjuyjw}@zju.edu.cn*
- *Z. Xiang is with Zhejiang University City College, Hangzhou 310015, China. e-mail: xiangzz@zucc.edu.cn*
- *S. Deng and R. Jiang are with Institute of Intelligence Applications, Yunnan University of Finance and Economics, Kunming 650221, China. e-mail: jiang_rong@aliyun.com*
- *S. Dustdar is with the Distributed Systems Group, Technische Universität Wien, 1040 Vienna, Austria. e-mail: dustdar@dsg.tuwien.ac.at*
- *A. Y. Zomaya is with the School of Computer Science, University of Sydney, Sydney, NSW 2006, Australia. e-mail: albert.zomaya@sydney.edu.au*
- *Y. Li is the corresponding author.*

ment of flow traffic are of great importance for cloud-native applications. In Kubernetes-native systems, Istio is popular for traffic management. It relies on the Envoy proxies co-configured along with the functions [17]. Istio enables the manager of the edge-cloud cluster to configure how each function's request calls its successors, along with its internal output data, routes within an Istio service mesh. Based on flexible and smart configurations, we can find that better utilization of traffic routing and stream mapping can result in less makespan even though the corresponding function placement is not optimal. This phenomenon is captured in Fig. 1. The top half of this figure is an undirected connected graph of six edge servers, abstracted from the physical infrastructure of the heterogeneous edge (In Section 2 and Section 3 we will explain how this connected graph is modeled). The numbers tagged in each vertex and beside each edge of the graph are the processing power (measured in *gflop/s*) and guaranteed bandwidth (measured in *GB/s*), respectively. The bottom half is an SFC with three functions. The number tagged inside each function is the required processing power (measured in *gflops*). The number tagged beside each data stream is the size of it (measured in *GB*). Fig. 1 demonstrates two solutions of function placement. The numbers tagged beside vertices and edges of each solution are the time consumed. Just in terms of function placement, solution 1 enjoys lower function processing time ($2.5 < 4$). However, the makespan of solution 2 is 1.5 lesser than solution 1 because it has a better traffic routing policy.

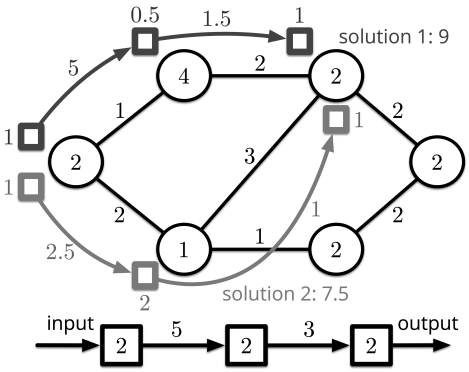

Fig. 1. Two function placement solutions for an SFC with different traffic routing policies.

The above example implies that different traffic routing policies could affect the makespan of an application significantly. It leads us to take traffic routing into consideration *proactively*. In this paper, we name the combination of function placement and stream mapping as *function embedding*. Moreover, if stream splitting is allowed, i.e., the internal output of a function can be split and routed on multiple paths, the makespan decreases further. This phenomenon is captured in Fig. 2. The structure of this figure is the same as Fig. 1. It demonstrates two function embedding solutions with stream splitting allowed or not, respectively. In solution 2, the output stream of the first function is divided into two parts, each with 2 or 3 units. Correspondingly, the times consumed on routing are 3 and 2.5, respectively. Although the two solutions have the same function placement, the makespan of solution 2, which is calculated as $1+\max\{3, 2.5\}+1+1.5+1 = 7.5$, is 4.5 lesser than solution 1.

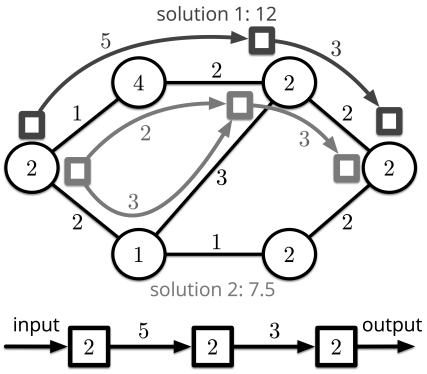

Fig. 2. Two function placement solutions with stream splitting allowed or not, respectively.

To capture the importance of proactive traffic routing, in this paper we study the *dependent function embedding* problem with *stream splitting* at the serverless edge. For a DAG with complicated structure, the problem is combinatorial and difficult to solve when the DAG scales up. In this paper, we firstly present the optimal substructure of the problem. Then, we solve each substructure optimally with dynamic programming. Specifically, for each substructure, we separate several infinity norm minimization sub-problems and solve them optimally by following the analytical solutions. Our paper makes the following contributions:

- *Model contribution:* We study the dependent function embedding problem at the serverless edge. Other than existing works where only function placement is studied, our novel contribution takes proactive traffic routing and data splitting into consideration and leverages dynamic programming as the approach to embed DAGs onto the heterogeneous edge.
- *Algorithm contribution:* We present an algorithm that solves the dependent function embedding problem optimally. We firstly find the optimal substructure of the problem. In each substructure, we derive the optimal data splitting for the internal data.
- *Experiment contribution:* We conduct extensive experiments on a cluster trace with 2119 unique DAGs from Alibaba [18]. Experimental results show that our algorithm significantly outperforms two algorithms, FixDoc [14] and HEFT [19], on the average completion time by 43.19% and 40.71%, respectively.

The remainder of the paper is organized as follows. In Section 2, we present a working example of the serverless functions. In Section 3, we present system model and formulate the problem. In Section 4, we present the proposed algorithm DPE and several auxiliary algorithms. Performance guarantee and complexity analysis are provided in the same place. The experiment results are demonstrated in Section 5. In Section 6, we review related works on functions placement on the heterogeneous edge. Section 7 concludes this paper.

2 A Working Example

In this section, we demonstrate a working example on dependent functions at the serverless edge.

The edge network is organized as a *weighted directed graph* [20] [21], where the vertices are edge servers with heterogeneous processing power and the edges are virtual links with certain propagation speed.

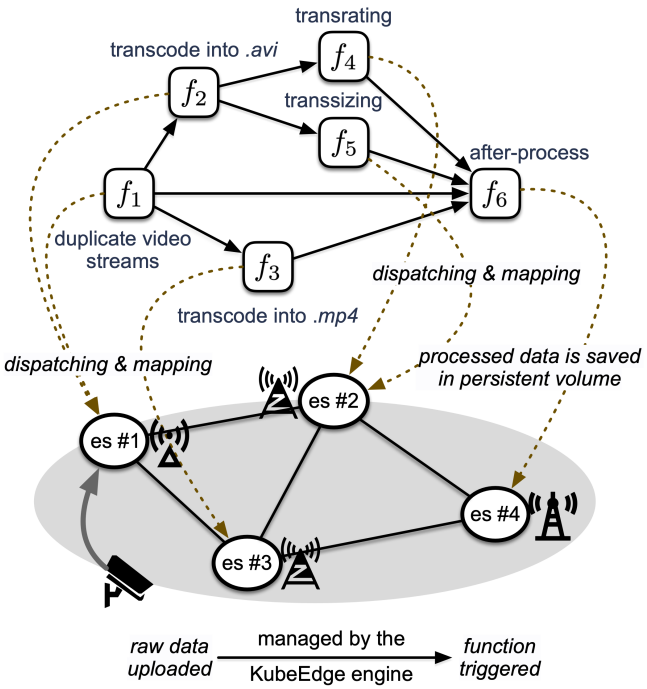

Fig. 3. The architecture of surveillance video processing by leveraging the elastic edge.

The edge network is managed with a lightweight Kubernetes platform, for example, the KubeEdge [22]. Let us deploy an application of surveillance video processing. The procedure is captured in Fig. 3. An edge device, i.e., a surveillance camera, uploads the raw video and pre-prepared configurations[1] to a nearby edge server periodically. With raw video input, the functions are pulled from remote Docker registries and triggered immediately. For video processing, `ffmpeg`[2] could be used to produce the corresponding Docker images. When it is done, the processed results are saved into a PersistentVolume (PV) configured in the PVC. In the above procedure, the camera only needs to upload the raw video data. The functions will be triggered automatically. The intermediate data produced by each function is saved into a local volume located at some host path.

To make the most of the quick response of the serverless edge, we need to study where each function is processed and how the flow traffic is mapped, to minimize the completion time as much as possible.

1. A configuration file is used to define Kubernetes object. It should describes the object's name, functions (Docker container instances) to call, PersistentVolumeClaims (PVC), and input & output relations of functions, etc.

2. https://www.ffmpeg.org/

3 System Model

Let us formulate the heterogeneous edge network as *an undirected connected graph* $\mathcal{G} \triangleq (\mathcal{N}, \mathcal{L})$, where $\mathcal{N} \triangleq \{n_1, ..., n_N\}$ is the set of edge servers and $\mathcal{L} \triangleq \{l_{ij}\}_{n_i,n_j \in \mathcal{N}}$ is the set of *virtual* links. A virtual link $l_{ij} = (n_i, n_j)$ is an augmented link between edge servers n_i and n_j. In $\mathcal{G}$, each edge server $n \in \mathcal{N}$ has a processing power ψ_n, measured in *tflop/s* while each virtual link $l_{ij} \in \mathcal{L}$ has a maximum bandwidth b_{ij}^{max}, measured in *GB/s*. We assume $b_{ij}^{max} = b_{ji}^{max}$. When $n_i = n_j$, we simply set the data transferring time as 0 since the intra-server processing is usually negligible. The computation of functions, which is non-preemptive, can be overlapped with communication.

Let us use $\mathcal{P}(n_i, n_j)$ to denote the set of *simple paths*[3] from source n_i to target n_j. For arbitrary given edge network, all the simple paths $\{\mathcal{P}(n_i, n_j)\}_{\forall n_i,n_j \in \mathcal{N}}$ can be obtained through depth-first search (DFS). The algorithm designed in this paper needs to know the simple paths between any two edge servers as a priori. In Section 4.4, we give a simple algorithm to calculate them based on DFS.

TABLE 1
Summary of key notations.

Notation	Description
$\mathcal{G} \triangleq (\mathcal{N}, \mathcal{L})$	The graph abstracted from the edge network
$\mathcal{N}$	The set of edge servers
$n_i \in \mathcal{N}$	The i-th edge server in $\mathcal{G}$
$\{\psi_n\}_{\forall n \in \mathcal{N}}$	Processing power of each edge server n
$\mathcal{L}$	The set of virtual links
$l_{ij} \in \mathcal{L}$	A virtual link from source n_i to target n_j in $\mathcal{G}$
$\{b_{ij}^{max}\}_{\forall l \in \mathcal{L}}$	Maximum bandwidth of the virtual link l_{ij}
$\mathcal{P}(n_i, n_j)$	The set of simple paths from server n_i to n_j
$(\mathcal{F}, \mathcal{E})$	The DAG abstracted from an IoT application
$\{f\}_{\forall f \in \mathcal{F}}$	The set of dependent functions
$c_i, \forall f_i \in \mathcal{F}$	The number of floating point operations of f_i
$e_{ij}, \forall f_i, f_j \in \mathcal{F}$	The data stream from function f_i to f_j
$s_{ij}, \forall f_i, f_j \in \mathcal{F}$	The data stream size from function f_i to f_j
$p(f) \in \mathcal{N}$	The placement of function $f \in \mathcal{F}$
$\varrho \in \mathcal{P}(n_i, n_j)$	A simple path in the set $\mathcal{P}(n_i, n_j)$
z_ϱ	The data stream size route through ϱ
$T(p(f_i))$	The finish time of f_i when scheduled onto $p(f_i)$
$t(e_{ij})$	The time consumed for transferring s_{ij}
$t(p(f_i))$	The time consumed for processing f_i on $p(f_i)$
ι_n	The earliest idle time of edge server n
$\sigma(\cdot, \cdot)$	The communication start-up cost
b_{mn}^{ϱ}	The bandwidth allocated to z_ϱ on l_{mn}

3.1 Application as a DAG

Each IoT application with dependent functions is modeled as a DAG. Let us use $(\mathcal{F}, \mathcal{E})$ to represent the DAG, where $\mathcal{F} \triangleq \{f_1, ..., f_F\}$ is the set of F dependent functions listed in *topological order*[4]. $\forall f_i, f_j \in \mathcal{F}, i \neq j$, if the output stream of f_i is the input of its downstream function f_j, a directed

3. A simple path from n_i to n_j is a path from source n_i to target n_j which contains no loop.

4. A topological order of a DAG is a linear ordering of all the vertices such that for every directed edge (f_i, f_j) in this DAG, f_i comes before f_j in this ordering.

edge e_{ij} exists. $\mathcal{E} \triangleq \{e_{ij} | \forall f_i, f_j \in \mathcal{F}\}$ is the set of all directed edges. For each function $f_i \in \mathcal{F}$, we write c_i for the required number of floating point operations of it. For each directed link $e_{ij} \in \mathcal{E}$, the data stream size is denoted as s_{ij} (measured in *GB*).

An entry function is a function which does not have predecessors. An exit function is a function which does not have successors. We write $\mathcal{F}_{entry}$ for the set of entry functions and $\mathcal{F}_{exit} \subset \mathcal{F}$ for the set of exit functions of the DAG. In this paper, we make no restrictions on the shape of DAGs. They could be single-entry single-exit, or multi-entry multi-exit.

3.2 Dependent Function Embedding

The dependent function embedding problem is decomposed into two sub-problems, where each function to be dispatched to and how each data stream is mapped onto virtual links.

We write $p(f) \in \mathcal{N}$ for the chosen edge server which $f \in \mathcal{F}$ to be dispatched to. For any function pair (f_i, f_j) and the associated edge $e_{ij} \in \mathcal{E}$, the data stream of size s_{ij} can be splitted and route through different paths in $\mathcal{P}\big(p(f_i), p(f_j)\big)$. $\forall \varrho \in \mathcal{P}\big(p(f_i), p(f_j)\big)$, let us use z_ϱ to represent the allocated non-negative data stream size for path ϱ. Then, $\forall e_{ij} \in \mathcal{E}$, we have the following constraint:

$$\sum_{\varrho \in \mathcal{P}\left(p(f_i), p(f_j)\right)} z_\varrho = s_{ij}.$$

Notice that if $p(f_i) = p(f_j)$, i.e., f_i and f_j are dispatched to the same edge server, then $\mathcal{P}(e_{ij}) = \varnothing$ and the data transferring time is zero.

Fig. 4 gives an example for data splitting. The connected graph has four edge servers and five virtual links. There are four simple paths between n_1 and n_4. The two squares represent the source function f_i and the destination function f_j. From the edge server $p(f_i)$ to the edge server $p(f_j)$, e_{ij} routes through three out of four simple paths with data size of 3 GB, 2 GB, and 1 GB, respectively. In this example, $s_{ij} = 6$. On closer observation, we can find that two data streams route through l_1. Each of them is from path ϱ_1 and ϱ_2 with 3 GB and 2 GB, respectively.

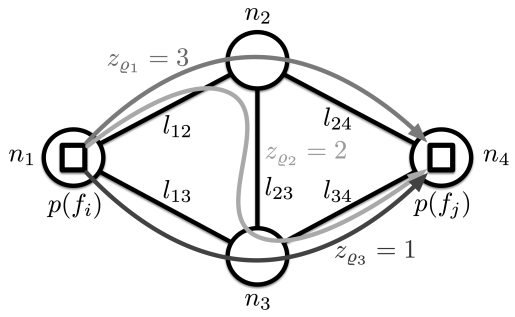

Fig. 4. An example of data stream splitting.

Theoretically, overhead exists in the splitting and merging operations of the data stream e_{ij} at the source $p(f_i)$ and the target $p(f_j)$, respectively. However, in our video transcoding scenarios, this overhead is negligible compared with extensive computation of functions or transferring of data streams in gigabytes.

3.3 Involution Function of Finish Time

Let us use $T\big(p(f_i)\big)$ to denote the finish time of f_i if it is scheduled to the edge server $p(f_i)$. Considering that the functions of the DAG have dependent relations, for each function $f_j \in \mathcal{F} \backslash \mathcal{F}_{entry}$, $T\big(p(f_j)\big)$ should involve according to

$$\begin{aligned} T\big(p(f_j)\big) &= \max\left\{\iota_{p(f_j)}, \max_{\forall i: e_{ij} \in \mathcal{E}} \Big(T\big(p(f_i)\big) + t(e_{ij})\Big)\right\} \\ &+ t\big(p(f_j)\big). \end{aligned} \tag{1}$$

In (1), $\iota_{p(f_j)}$ is the earliest idle time of the edge server $p(f_j)$. In our model, the processing of functions is non-preemptive, thus an edge server is idle *iff* the functions assigned to it complete. $t(e_{ij})$ is the transferring time of data stream e_{ij} and $t\big(p(f_j)\big)$ is the processing time of f_j on edge server $p(f_j)$. Corresponding to (1), for each entry function $f_i \in \mathcal{F}_{entry}$,

$$T\big(p(f_i)\big) = \iota_{p(f_j)} + t\big(p(f_i)\big) \tag{2}$$

since f_i has no predecessors. In the following, we demonstrate the calculation of $t(e_{ij})$ and $t\big(p(f_j)\big)$ in turn.

$t(e_{ij})$ is constitutive of two parts, where the first is the communication start-up cost between the two functions f_i and f_j. This cost is mainly decided by the configurations in `kube-proxy` in this edge-cloud Kubernetes cluster [23]. For example, if we use Envoy to implement the network proxy and communication bus for the cluster, data transfer is mainly handled by Envoy route filters. Before data transfer, Envoy proxy needs to do some preparatory works. For example, looking for the route tables to get the actual routing paths. We simply use $\sigma(f_i, f_j)$ to represent the communication start-up cost. The second is the actual communication cost between $p(f_i)$ and $p(f_j)$, which is actually decided by the slowest data transferring time of z_ϱ among all $\varrho \in \mathcal{P}\big(p(f_i), p(f_j)\big)$. Considering that the edge network serves for thousands of services and applications, we assume that for each data stream z_ϱ, during its transferring, the bandwidth allocated to it on each virtual link $l_{mn} \in \varrho$ is *fixed* as b^ϱ_{mn}, and $b^\varrho_{mn} \ll b^{max}_{mn}$ holds. To sum up, $t(e_{ij})$ is defined as

$$t(e_{ij}) \triangleq \sigma(f_i, f_j) + \max_{\varrho \in \mathcal{P}\left(p(f_i), p(f_j)\right)} \sum_{l_{mn} \in \varrho} \frac{z_\varrho}{b^\varrho_{mn}}, \tag{3}$$

Note that the above formulation ignores the data splitting and merging costs as we have mentioned earlier. In real-world scenarios, the real-time bandwidth for transferring some data stream at some point is unknowable because the network status is always in dynamic changes. Even so, our assumption on the fixed bandwidth allocation is reasonable since it can be guaranteed by the QoS level defined in the generic network slice template (GST) [24]. Besides, our model does not require that each function pair of the DAG enjoys the same bandwidth on each virtual link.

$\forall f_j \in \mathcal{F}$, $t\big(p(f_j)\big)$ is decided by the processing power of the chosen edge server $p(f_j)$. Since the processing of functions is non-preemptive, each function is executed with full power. Hence, $t\big(p(f_j)\big)$ is defined as

$$t\big(p(f_j)\big) \triangleq \frac{c_j}{\psi_{p(f_j)}}. \tag{4}$$

3.4 Problem Formulation

After all functions are scheduled, the makespan will be the finish time of the slowest exit function (i.e., the function without successors). Our target is to minimize the makespan of the DAG by finding the optimal $\boldsymbol{p} \triangleq \{p(f)\}_{\forall f \in \mathcal{F}}$ and the optimal $\boldsymbol{z} \triangleq \{z_\varrho | \forall \varrho \in \mathcal{P}\big(p(f_i), p(f_j)\big)\}_{e_{ij} \in \mathcal{E}}$. Thus, the dependent function embedding problem is formulated as:

$$\begin{aligned} \mathbf{P}: \quad & \min\nolimits_{\boldsymbol{p},\boldsymbol{z}} \max\nolimits_{f \in \mathcal{F}_{exit}} T\big(p(f)\big) \\ s.t. \quad & \textstyle\sum_{\varrho \in \mathcal{P}\left(p(f_i),p(f_j)\right)} z_\varrho = s_{ij}, \forall e_{ij} \in \mathcal{E}, \quad (5) \\ & \boldsymbol{z} \geq \mathbf{0}. \quad (6) \end{aligned}$$

4 Algorithm Design

In this section, we firstly give the optimal substructure hidden in $\mathbf{P}$. Then, we propose the DPE algorithm and provide theoretical analysis on its optimality and complexity. In the end, we give a method to obtain simple paths for the given edge network based on DFS. To simplify the notations, in the following, we replace $\mathcal{P}\big(p(f_i), p(f_j)\big)$ and $\sigma\big(p(f_i), p(f_j)\big)$ by $\mathcal{P}_{ij}$ and σ_{ij}, respectively.

4.1 Finding Optimal Substructure

In consideration of the dependency relationship between the fore-and-aft functions, the optimal placement of functions and optimal mapping of data streams cannot be obtained at the same time. Nevertheless, we can solve it optimally step-by-step based on its optimal substructure.

Let us use $T^\star\big(p(f)\big)$ to denote the earliest finish time of function f when it is placed on edge server $p(f)$. Based on (1), $\forall f_j \in \mathcal{F} \backslash \mathcal{F}_{entry}$, we have

$$\begin{aligned} T^\star\big(p(f_j)\big) = \max \Bigg\{ & \max\nolimits_{\forall i: e_{ij} \in \mathcal{E}} \Bigg[\min\nolimits_{p(f_i), \{z_\varrho\}_{\forall \varrho \in \mathcal{P}_{ij}}} \\ & \Big(T^\star\big(p(f_i)\big) + t(e_{ij}) \Big) \Bigg], \iota_{p(f_j)} \Bigg\} + t\big(p(f_j) \quad (7) \end{aligned}$$

Besides, for all the entry functions $f_i \in \mathcal{F}_{entry}$, $T^\star\big(p(f_i)\big)$ is calculated by (2) immediately.

With (7), for each function pair (f_i, f_j) where e_{ij} exists, we can define the sub-problem $\mathbf{P}_{sub}$:

$$\begin{aligned} \mathbf{P}_{sub}: \quad & \min\nolimits_{p(f_i), \{z_\varrho\}_{\forall \varrho \in \mathcal{P}_{ij}}} \boldsymbol{\Phi}_{ij} \triangleq \Big(T^\star\big(p(f_i)\big) + t(e_{ij}) \Big) \\ s.t. \quad & \textstyle\sum_{\varrho \in \mathcal{P}_{ij}} z_\varrho = s_{ij}, \quad (8) \\ & z_\varrho \geq 0, \forall \varrho \in \mathcal{P}_{ij}. \quad (9) \end{aligned}$$

Note that (8) and (9) are from a subset of constraints (5) and (6), respectively. In $\mathbf{P}_{sub}$, we need to decide where f_i is placed and how e_{ij} is mapped. By solving $\mathbf{P}_{sub}$, we can obtain the earliest finish time $T^\star\big(p(f_j)\big)$ by (7). In this way, $\mathbf{P}$ is solved *optimally* by calculating the earliest finish time of each function in topological order.

4.2 Optimal Data Splitting

To solve $\mathbf{P}_{sub}$ optimally, we firstly fix the position of f_i, i.e. $p(f_i)$, then concentrate on the optimal mapping of e_{ij}.

To minimize $t(e_{ij})$, let us define a diagonal matrix $\mathbf{A}$ as follows.

$$\mathbf{A} \triangleq \operatorname{diag}\Bigg(\sum_{l_{mn} \in \varrho_1} \frac{1}{b_{mn}^{\varrho_1}}, \sum_{l_{mn} \in \varrho_2} \frac{1}{b_{mn}^{\varrho_2}}, ..., \sum_{l_{mn} \in \varrho_{|\mathcal{P}_{ij}|}} \frac{1}{b_{mn}^{\varrho_{|\mathcal{P}_{ij}|}}} \Bigg).$$

Obviously, all the diagonal elements of $\mathbf{A}$ are positive real numbers. The variables that need to be determined can be written as $\boldsymbol{z}_{ij} \triangleq [z_{\varrho_1}, z_{\varrho_2}, ..., z_{\varrho_{|\mathcal{P}_{ij}|}}]^\top \in \mathbb{R}^{|\mathcal{P}_{ij}|}$. Thus, $\mathbf{P}_{sub}$ is converted into

$$\begin{aligned} \mathbf{P}_{norm}: \quad & \min\nolimits_{\boldsymbol{z}_{ij}} \|\mathbf{A}\boldsymbol{z}_{ij}\|_\infty \\ s.t. \quad & \begin{cases} \mathbf{1}^\top \boldsymbol{z}_{ij} = s_{ij}, \\ \boldsymbol{z}_{ij} \geq \mathbf{0}. \end{cases} \quad (10) \end{aligned}$$

We drop $T^\star\big(p(f_i)\big)$ and σ_{ij} readily since constant does not change the optimal solution $\boldsymbol{z}_{ij}^\star$. $\mathbf{P}_{norm}$ is an infinity norm minimization problem. By introducing slack variables $\tau \in \mathbb{R}$ and $\boldsymbol{y} \in \mathbb{R}^{|\mathcal{P}_{ij}|}$, $\mathbf{P}_{norm}$ can be transformed into the following *slack form*:

$$\begin{aligned} \mathbf{P}'_{slack}: & \min_{\boldsymbol{z}'_{ij} \triangleq [\boldsymbol{z}_{ij}^\top, \boldsymbol{y}^\top]^\top} \tau \\ s.t. \quad & \begin{cases} \sum_{\varrho \in \mathcal{P}_{ij}} z_\varrho = s_{ij}, \\ \mathbf{A}\boldsymbol{z}_{ij} + \boldsymbol{y} = \tau \cdot \mathbf{1}, \\ \boldsymbol{z}'_{ij} \geq \mathbf{0}. \end{cases} \end{aligned}$$

$\mathbf{P}'_{slack}$ is feasible and its optimal objective value is finite. As a result, simplex method and interior point method can be applied to obtain the optimal solution efficiently.

However, these standard methods might be unacceptable when the scale of $\mathcal{G}$ increases since simplex method has exponential complexity and interior point method is at least $O(|\boldsymbol{z}'_{ij}|^{3.5})$ in the worst case [25]. Luckily, we can directly obtain the analytical expression of the optimal $\boldsymbol{z}_{ij}^\star$. The result is introduced in the following theorem.

Theorem 1. *The optimal objective value of* $\mathbf{P}_{norm}$ *is*

$$\min_{\boldsymbol{z}_{ij}} \|\mathbf{A}\boldsymbol{z}_{ij}\|_\infty = \frac{s_{ij}}{\sum_{k=1}^{|\mathcal{P}_{ij}|} 1/A_{k,k}}, \quad (11)$$

iff

$$A_{u,u}\boldsymbol{z}_{ij}^{(u)} = A_{v,v}\boldsymbol{z}_{ij}^{(v)}, 1 \leq u \neq v \leq |\mathcal{P}_{ij}|, \quad (12)$$

where $\boldsymbol{z}_{ij}^{(u)}$ *is the* u*-th component of vector* $\boldsymbol{z}_{ij}$ *and* $A_{u,u}$ *is the* u*-th diagonal element of* $\mathbf{A}$.

Proof. For $\mathbf{P}_{norm}$, we have

$$\|\mathbf{A}\boldsymbol{z}_{ij}\|_\infty \triangleq \max_k \Big\{ |A_{k,k}\boldsymbol{z}_{ij}^{(k)}| \Big\} = \lim_{x \to \infty} \sqrt[x]{\sum_{k=1}^{|\mathcal{P}_{ij}|} \Big(A_{k,k}\boldsymbol{z}_{ij}^{(k)} \Big)^x}$$

because $\forall k, A_{k,k} > 0, \boldsymbol{z}_{ij}^{(k)} \geq 0$. According to the *AM–GM inequality*, the following inequality always holds:

$$\frac{\sum_{k=1}^{|\mathcal{P}_{ij}|} \Big(A_{k,k}\boldsymbol{z}_{ij}^{(k)} \Big)^x}{|\mathcal{P}_{ij}|} \geq \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} \Big(A_{k,k}\boldsymbol{z}_{ij}^{(k)} \Big)^x} \quad (13)$$

iff (12) is satisfied. It yields that $\forall x > 0$,

$$\sqrt[x]{\frac{\sum_{k=1}^{|\mathcal{P}_{ij}|} \Big(A_{k,k}\boldsymbol{z}_{ij}^{(k)} \Big)^x}{|\mathcal{P}_{ij}|}} \geq \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} A_{k,k}\boldsymbol{z}_{ij}^{(k)}}. \quad (14)$$

Multiply both sides of (14) by $\sqrt[x]{|\mathcal{P}_{ij}|}$, we have

$$\sqrt[x]{\sum_{k=1}^{|\mathcal{P}_{ij}|} \Big(A_{k,k}\boldsymbol{z}_{ij}^{(k)} \Big)^x} \geq \sqrt[x]{|\mathcal{P}_{ij}|} \cdot \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} A_{k,k}\boldsymbol{z}_{ij}^{(k)}}. \quad (15)$$

By taking the limit of (15), we have

$$\|\mathbf{A}\boldsymbol{z}_{ij}\|_\infty \geq \lim_{x\to\infty} \sqrt[x]{|\mathcal{P}_{ij}|} \cdot \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} A_{k,k}\boldsymbol{z}_{ij}^{(k)}}. \tag{16}$$

Combining with (10) and (12), the right side of (16) is actually a constant. In other words,

$$\begin{aligned}\min_{\boldsymbol{z}_{ij}} \|\mathbf{A}\boldsymbol{z}_{ij}\|_\infty &= \lim_{x\to\infty} \sqrt[x]{|\mathcal{P}_{ij}|} \cdot \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} A_{k,k}\boldsymbol{z}_{ij}^{(k)}} \\ &= \sqrt[|\mathcal{P}_{ij}|]{\prod_{k=1}^{|\mathcal{P}_{ij}|} A_{k,k}\boldsymbol{z}_{ij}^{(k)}} \quad \triangleright \text{ with (10)} \\ &= \frac{s_{ij}}{\sum_{k=1}^{|\mathcal{P}_{ij}|} 1/A_{k,k}}.\end{aligned}$$

The result shows that (11) and (12) are the optimal objective value and corresponding optimal condition of $\mathbf{P}_{norm}$, respectively. □

Base on (12), we can infer that the optimal variable $\boldsymbol{z}_{ij}^\star > \mathbf{0}$ holds, which means that $\forall \varrho \in \mathcal{P}_{ij}$, $z_\varrho^\star \neq 0$. To sum up, the way to obtain the optimal data splitting and mapping is summarized into Algorithm 1 (OSM) as follows.

Algorithm 1: Optimal Stream Mapping (OSM)

Input: $\mathcal{G}$, the function pair (f_i, f_j), and $p(f_j)$
Output: The optimal $\boldsymbol{\Phi}_{ij}^\star$, $p^\star(f_i)$, and $\boldsymbol{z}_{ij}^\star$

1 **for** *each* $m \in \mathcal{N}$ **do in parallel**
2 $p(f_i) \leftarrow m$
3 /* Obtain the m-th optimal $\boldsymbol{\Phi}_{ij}$ by (11) */
4 $\boldsymbol{\Phi}_{ij}^{(m)} \leftarrow \frac{s_{ij}}{\sum_k 1/A_{k,k}^{(m)}} + T^\star\big(p(f_i)\big)$
5 **end for**
6 $p^\star(f_i) \leftarrow \operatorname{argmin}_{m\in\mathcal{N}} \boldsymbol{\Phi}_{ij}^{(m)}$
7 Calculate $\boldsymbol{z}_{ij}^\star$ by (10) and (12) with $\mathbf{A} = \mathbf{A}^{(p^\star(f_i))}$

From line 1 to line 5, we can find that OSM solves $\mathbf{P}_{norm}$ by solving $|\mathcal{N}|$ times of $\mathbf{P}_{sub}$ in parallel, each with a different $p(f_i)$. The procedure can be executed in parallel because intercoupling is nonexistent. In line 4, the objective of $\mathbf{P}_{sub}$ is obtained with the analytical solution (11) directly. The most time-consuming operation lies in line 4 and line 6 since they have at least one traversal over edge servers and simple paths, respectively. OSM is in $O\Big(\max\{|\mathcal{N}|, |\mathcal{P}_{ij}|\}\Big)$-complexity.

4.3 Dynamic Programming-based Embedding

Combing OSM with dynamic programming, we have the algorithm DPE (Dynamic Programming-based Embedding).

In DPE, the loop starts from non-entry functions with a topological order. For each non-entry function $f_j \in \mathcal{N}\backslash\mathcal{N}_{entry}$, DPE firstly fixes its placement $p(f_j)$ as some edge server n in line 3. Then, from line 4 to line 12, DPE solves the sub-problem $\mathbf{P}_{sub}^{(i)}$ by calling OSM for the function pairs $(f_i, f_j), e_{ij} \in \mathcal{E}$ in turn. If $p^\star(f_i)$ has been decided beforehand (under the case where a function is a predecessor of multiple functions), DPE will skip f_i and go to process the next predecessor $f_{i'}$ (line 5 $\sim$ line 7). At the end, DPE updates the finish time of f_j based on the solution of $\{\mathbf{P}_{sub}^{(i)}\}_{\forall e_{ij}\in\mathcal{E}}$ in line 13. Note that the finish time of each entry function f_i should be calculated with (2) before solving $\mathbf{P}_{sub}^{(i)}$. When all the finish time of functions have been calculated, the global minimal makespan of the DAG can be obtained by

$$\max_{f\in\mathcal{F}_{exit}} \operatorname*{argmin}_{p^\star} T^\star\big(p^*(f)\big).$$

The optimal embedding of each function can be retrieved from $\boldsymbol{z}^\star$ and $\boldsymbol{p}^\star$.

Algorithm 2: DP-based Embedding (DPE)

Input: $\mathcal{G}$ and $(\mathcal{F}, \mathcal{E})$
Output: Optimal value and corresponding solution

1 **for** $j = |\mathcal{F}_{entry}| + 1$ *to* $|\mathcal{F}|$ **do**
2 **for** *each* $n \in \mathcal{N}$ **do**
3 $p(f_j) \leftarrow n$ // Fix the placement of f_j
4 **for** *each* $f_i \in \{f_i | e_{ij}$ *exists*$\}$ **do**
5 **if** $p^\star(f_i)$ *has been decided* **then**
6 **continue**
7 **end if**
8 **if** $f_i \in \mathcal{F}_{entry}$ **then**
9 $\forall p(f_i) \in \mathcal{N}$, update $T^\star\big(p(f_i)\big)$ by (2)
10 **end if**
11 Obtain the optimal $\boldsymbol{\Phi}_{ij}^\star$, $p^\star(f_i)$, and $\boldsymbol{z}_{ij}^\star$ by calling OSM
12 **end for**
13 Update $T^\star\big(p(f_j)\big)$ by (7)
14 **end for**
15 **end for**
16 **return** $\max_{f\in\mathcal{F}_{exit}} \operatorname{argmin}_{p^*} T^\star\big(p^*(f)\big)$, $\boldsymbol{z}^\star$, *and* $\boldsymbol{p}^\star$

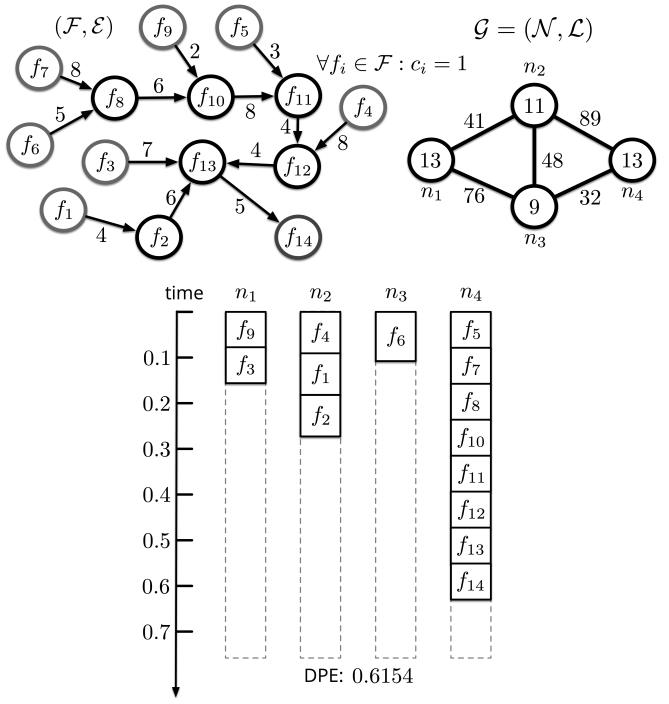

Fig. 5. Embedding of a DAG with the DPE algorithm.

Fig. 5 demonstrates an example on how PDE works. The top left portion of the figure is a DAG randomly sampled from the Alibaba cluster trace, where all the functions are named in the manner of topological order. $\forall f_i \in \mathcal{F}$, c_i is set as 1. The top right is the edge server cluster $\mathcal{G}$. The bottom demonstrates how the functions are placed and scheduled by DPE.

In the following, we demonstrate the complexity of DPE.

Theorem 2. *In the worst case, the complexity of DPE is*

$$O\Big(|\mathcal{N}| \times |\mathcal{E}| \times \max\big\{|\mathcal{N}|, \max_{n_i,n_j} |\mathcal{P}_{ij}|\big\}\Big).$$

Proof. For each $f_j \in \mathcal{F}\backslash\mathcal{F}_{entry}$, the average number of predecessors is $\frac{|\mathcal{E}|}{\mathcal{F}-|\mathcal{F}_{entry}|}$. Thus, for each placement of f_j, OSM is called $\frac{|\mathcal{E}|}{\mathcal{F}-|\mathcal{F}_{entry}|}$ times on average. Thus, OSM is called

$$\Big(|\mathcal{F}| - |\mathcal{F}_{entry}|\Big) \times |\mathcal{N}| \times \frac{|\mathcal{E}|}{|\mathcal{F}| - |\mathcal{F}_{entry}|}$$

times in the worst case (under which a function is not a predecessor of more than one function). Due to the complexity of OSM is $O\Big(\max\{|\mathcal{N}|, |\mathcal{P}_{ij}|\}\Big)$, the result is immediate. □

4.4 Recursion-based Path Finding

In this subsection, we demonstrate how to obtain the simple paths for each vertex pair in the undirected graph $\mathcal{G}$. All the simple paths can be calculated, based on the method proposed in the following, and saved into a shareable table for each edge server. The calculation of simple paths is to be a *once-and-done* operation, and it is calculated immediately after the creation of the Kubernetes cluster. Once the edge network status changes, for example, an edge server is inaccessible or a new edge server is configured into the network, only the related entries in this table need to be re-calculated. The modification has much lower complexity.

Before introducing our algorithm for path finding, we firstly prove that the simple path finding problem is NP-hard.

Proposition 1. *Finding all the simple paths for all the vertex pairs in the undirected connected graph $\mathcal{G}$ is NP-hard*[5].

Proof. Suppose there exists a polynomial algorithm $\mathcal{A}$ which can obtain all the simple paths between any source vertex $p(f_i) \in \mathcal{N}$ and any destination vertex $p(f_j) \in \mathcal{N}$. Because $\mathcal{G}$ is a connected graph, $\mathcal{P}\big(p(f_i), p(f_j)\big)$ is not empty. Since $\mathcal{A}$ lists all the simple paths in $\mathcal{P}\big(p(f_i), p(f_j)\big)$, we can easily design another simple polynomial algorithm $\mathcal{A}'$, based on the results of $\mathcal{A}$, to judge which simple path in $\mathcal{P}\big(p(f_i), p(f_j)\big)$ is the longest path. Since $\mathcal{G}$ is undirected and connected, we can claim that this longest simple path has to traverse all vertices of $\mathcal{G}$. It means that we have found a Hamiltonian Path in polynomial time, which is well known to be NP-hard. As a result, we can conclude that, unless P = NP, $\mathcal{A}$ is very unlikely to exist. Therefore, finding all the simple paths between any two vertices of an undirected connected graph is NP-hard. Based on this, we can easily prove that finding all the simple paths for all the vertex pairs of a undirected connected graph is also NP-hard. □

5. The proof of this proposition is based on the discussions in the web page https://stackoverflow.com/questions/9535819/find-all-paths-between-two-graph-nodes.

Let us introduce a new notation, $\Omega(n_i, n_j, \mathcal{M})$, to represent the set of simple paths from n_i to n_j where no path goes through vertices from the set $\mathcal{M} \subseteq \mathcal{N}$. The set of simple paths from n_i to n_j we want, i.e. $\mathcal{P}_{ij}$, is equal to $\Omega(n_i, n_j, \varnothing)$. $\forall n \in \mathcal{N}$, let us use $\mathcal{A}(n)$ to represent the set of edge servers adjacent to n. Then, $\Omega(n_i, n_j, \mathcal{M})$ can be calculated by the following recursion formula:

$$\Omega(n_i, n_j, \mathcal{M}) = \Big\{J(\varrho, n_i)\Big| \bigcup_{m\in\mathcal{S}} \Omega\big(m, n_j, \mathcal{M} \cup \{n_i\}\big)\Big\},$$

where $\mathcal{S} \triangleq \mathcal{A}(n_i) - \mathcal{M} \cup \{n_i\}$, and $J(\varrho, n_i)$ is a function that joins the node n_i to the path ϱ and returns the new joint path $\varrho \cup \{n_i\}$.

Based on the above recursion formula, we introduce Algorithm 3, RPF (Recursion-based Path Finding). Before calling RPF, we need to initialize the global variables. Specifically, $\mathcal{P}_{ij}$ stores all the simple paths, which is initialized as $\varnothing$. $\mathcal{V}$, as the set of visited vertices, is initialized as $\varnothing$. ϱ, as the path to be calculated currently, which is also initialized as $\varnothing$.

Algorithm 3: Recursion-based Path Finding (RPF)

Input: $n_i, n_j \in \mathcal{N}$
Output: $\mathcal{P}_{ij}$
$\mathcal{V}, \varrho, \mathcal{P}_{ij} \leftarrow \varnothing, \varnothing, \varnothing$
$n \leftarrow n_i$
if $n == n_j$ **then**
  $\mathcal{P}_{ij}.append\big(J(\varrho, n)\big)$ `// Store the path J(ϱ,n)`
else
  $\varrho.push(n)$; $\mathcal{V}.add(n)$
  for *each* $n' \in \mathcal{A}(n) - \mathcal{V}$ **do**
    `RPF`(n_i, n', n_j) `// Recursive call`
  end for
  $\varrho.pop()$; $\mathcal{V}.delete(n)$
end if

Although the calling of RPF is a *once-and-done* operation, we still give the complexity of it as follows. Let us use $\kappa(i, j)$ to denote the flops required to compute all the simple paths between n_i and n_j. If $\mathcal{G}$ is fully connected,

$$\begin{aligned}\kappa(1, |\mathcal{N}|) &= \sum_{i=2}^{|\mathcal{N}|-1} \Big(\kappa\big(i, |\mathcal{N}|\big) + 1\Big) + 1 \\ &= (|\mathcal{N}| - 1) + (|\mathcal{N}| - 2) \cdot \kappa(2, |\mathcal{N}|).\end{aligned}$$

Further, we use κ_i to replace $\kappa(i, |\mathcal{N}|)$ by fixing the target vertices as the $|\mathcal{N}|$-th vertex. We can conclude that $\forall i \in \{1, ..., |\mathcal{N}| - 1\}$,

$$\kappa_i = \big(|\mathcal{N}| - i\big) + \big(|\mathcal{N}| - i - 1\big) \cdot \kappa_{i+1}. \tag{17}$$

Based on the recursion formula (17), we have

$$\begin{aligned}\kappa_1 &= (|\mathcal{N}|-2)!\cdot\sum_{i=1}^{|\mathcal{N}|-1}\frac{|\mathcal{N}|-i}{(|\mathcal{N}|-i-1)!}\\ &= (|\mathcal{N}|-2)!\cdot\sum_{i=1}^{|\mathcal{N}|-1}\frac{(|\mathcal{N}|-i)^2}{(|\mathcal{N}|-i)!}\\ &= (|\mathcal{N}|-2)!\cdot\sum_{i=1}^{|\mathcal{N}|-1}\frac{i^2}{i!},\end{aligned}\tag{18}$$

which is the maximum flops required to compute all the simple paths between any two edge servers. To get the upper bound of κ_1, in the following, we introduce several lemmas.

Lemma 1. *$\forall N \geq 7$ and $N \in \mathbb{N}^+$, $N! > N^3(N+1)$.*

Proof. The proof is based on *induction*. When $N = 7$, $N! = 5040 > N^3(N+1) = 2744$. The lemma holds. Assume that the lemma holds for $N = q$, i.e., $q! > q^3(q+1)$ (*induction hypothesis*). Then, for $N = q+1$, we have

$$(q+1)! = (q+1)\cdot q! > (q+1)^2q^3.\tag{19}$$

Notice that the function $g(q) \triangleq (\frac{1}{q}+\frac{2}{q^2}+\frac{4}{q^3})^{-1}$ monotonically increases when $q \in \mathbb{N}^+ - \{1,2\}$. Hence $g(q) \geq g(3) = \frac{27}{25} > 1$, and

$$1 < \frac{q^3}{(q+2)^2} < \frac{q^3}{(q+1)(q+2)},$$

which indicates that

$$q^3 > (q+1)(q+2).\tag{20}$$

Multiply both sides of (20) by $(q+1)^2$, we get

$$(q+1)\cdot q^3\cdot(q+1) > (q+1)^3(q+2),$$

which means the induction hypothesis is not violated, and the lemma holds for $q+1$. □

Lemma 2. *$\forall N \geq 2$ and $N \in \mathbb{N}^+$, $\sum_{i=1}^{N-1}\frac{i^2}{i!} < 6 - \frac{1}{N}$.*

Proof. We can verify that when $N \in [2,7]\cap\mathbb{N}^+$, the lemma holds. In the following we prove the lemma holds for $N > 7$ by *induction*. Assume that the lemma holds for $N = q$, i.e., $\sum_{i=1}^{q-1}\frac{i^2}{i!} < 6 - \frac{1}{q}$ (*induction hypothesis*). Then, for $N = q+1$, we have

$$\sum_{i=1}^{q}\frac{i^2}{i!} < 6 - \frac{1}{q} + \frac{q^2}{q!}.\tag{21}$$

By applying Lemma 1, we get

$$\sum_{i=1}^{q}\frac{i^2}{i!} < 6 - \frac{1}{q+1},$$

which means the lemma holds for $q+1$. □

Based on the above lemmas, we can obtain the complexity of RPF, as illustrated in the following theorem:

Theorem 3. *In the worst case, where $\mathcal{G}$ is a fully connected graph and $|\mathcal{N}| \geq 2$, the complexity of RPF is $O\big((|\mathcal{N}|-2)!\big)$.*

Proof. According to Lemma 2,

$$\lim_{|\mathcal{N}|\to\infty}\sum_{i=1}^{|\mathcal{N}|-1}\frac{i^2}{i!} < 6.$$

Hence $\lim_{|\mathcal{N}|\to\infty}\kappa_1 < 6(|\mathcal{N}|-2)! = O\Big((|\mathcal{N}|-2)!\Big)$. □

In real-world edge computing scenario for IoT stream processing, $\mathcal{G}$ might not be fully connected. Even though, the number of edge servers is small. Thus, the real complexity is much lower. Besides, note that the calling of RPF is actually a *once-and-done* operation. As a priori to DPE, query in this shareable table is of linear complexity. This conclusion is immediate by analyzing line 4 of OSM.

4.5 Extending to Multiple DAGs

DPE can be performed for multiple DAGs. To do this, we only need to concatenate all the DAGs into one augmented DAG. Specifically, if a DAG has more than one entry function, we add a dummy head function f_h and several directed edges $\{e_{hi} \mid \forall f_i \in \mathcal{F}_{entry}\}$ such that the required floating point operations of f_h is zero and all the s_{hi} are zero. In the same way, if a DAG has more than one exit function, we add a dummy tail function and corresponding directed edges. Based on that, we add a directed edge between the (dummy) tail function of the DAG before and the (dummy) head function of the DAG after. Take the augmented DAG as the input of DPE, we get the embedding results.

5 Experimental Validation

In this section, we conduct extensive experiments to evaluate the effectiveness and efficiency of DPE. Based on a real-world dataset, the Alibaba's cluster trace [18], we firstly verify the performance of DPE against several popular algorithms on makespan. Then, we analyze the impact of several system parameters.

We summarize the key findings of our experiments as follows, and details can be found in Section 5.2.

- Compared with a well-known heuristic HEFT [19], and a recent algorithm FixDoc [14], DPE achieves the smallest makespan over all the 2119 DAGs with absolute superiority under arbitrary parameters.
- DPE is robust to the system parameters. The advantage of DPE is magnified when the scale of the edge network increases.

5.1 Experiment Setup

IoT stream processing workloads. The simulation is conducted based on Alibaba's cluster trace of data analysis. This dataset contains more than 3 million jobs (called applications in this work), and 20365 jobs with unique DAG information. Considering that there are too many DAGs with only single-digit functions, we sampled 2119 DAGs with different sizes from the dataset. The distribution of the samples is visualized in Fig. 6. For each $f \in \mathcal{F}$, the processing power required and output data size are extracted from the corresponding job in the dataset and scaled to $[1,10]\times10^2$ *gflops* and $[5,15]\times10^2$ *MB*, respectively.

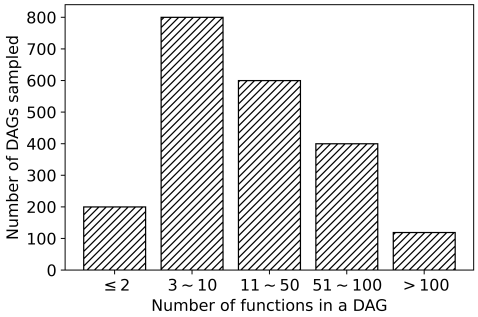

Fig. 6. Data distribution sampled from the cluster trace.

heterogeneous edge servers. In our simulation, the number of edge servers is 10 in default. Considering that the edge servers are required to formulate a *connected* graph, the impact of the sparsity of the graph is also studied. The processing power of edge servers and the allocated of bandwidth for each data flow are uniformly sampled from $[20, 40]$ *gflops* and $[30, 80] \times 10^2$ *MB/s* in default, respectively.

Algorithms compared. We compare DPE with the following algorithms.

- *FixDoc* [14]: FixDoc is a function placement and DAG scheduling algorithm with fixed function configuration for the serverless edge computing platform. FixDoc places each function onto *homogeneous* edge servers optimally to minimize the DAG completion time. Actually, [14] also proposes an improved version, GenDoc, with function configuration optimized, too. However, for IoT applications such as video transcoding, on-demand function configuration is not necessary since the tools such as `ffmpeg` can be installed and configured beforehand easily. Thus, we only compare DPE with FixDoc.
- *Heterogeneous Earliest-Finish-Time (HEFT)* [19]: HEFT is a classic heuristic to schedule a set of dependent tasks onto heterogeneous workers with communication time taken into account. Starting with the highest priority, tasks are assigned to different workers to heuristically minimize the overall completion time. HEFT is an algorithm that stands the test of time.

5.2 Experimental Results

All the experiments are implemented in Python 3.7 on macOS Catalina equipped with 3.1 GHz Quad-Core Intel Core i7 and 16 GB RAM. In the following, the unit of the left y-axis is *100 seconds*.

5.2.1 Theoretical Performance Verification

Fig. 7 illustrates the overall performance of the three algorithms. For different data batches, DPE can reduce 43.19% and 40.71% of the completion time on average over FixDoc and HEFT on 2119 DAGs. The advantage of DPE is more obvious when the scale of DAG is large because the parallelism is fully guaranteed. Fig. 8 shows the accumulative distribution of 2119 DAGs' completion time. DPE is superior to HEFT and FixDoc on 100% of the DAGs. In Fig. 8, the maximum completion time of a DAG achieved by DPE is 124 seconds. By contrast, only less than 94% of DAGs' completion times achieved by HEFT and FixDoc are smaller than this value.

Fig. 7 and Fig. 8 verify the superiority of proactive stream mapping and data splitting. By spreading data streams over multiple virtual links, data transferring time is greatly reduced. Besides, the optimal substructure makes sure DPE can find the optimal placement of each function simultaneously.

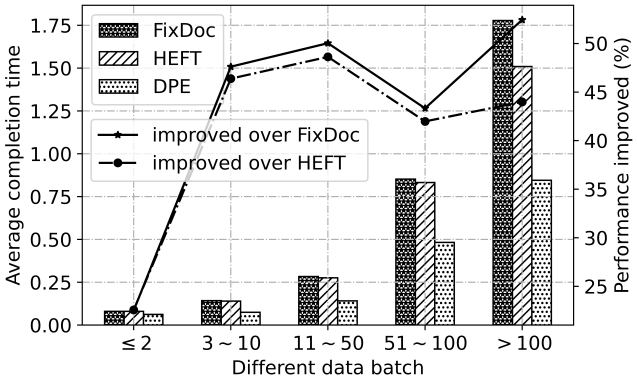

Fig. 7. Average completion time achieved by different algorithms.

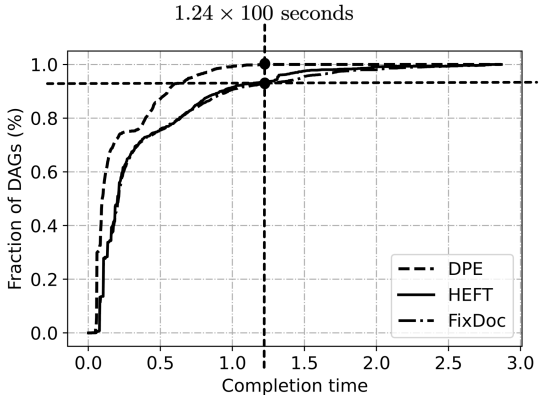

Fig. 8. The accumulative distributions of the completion time achieved by three algorithms, respectively.

5.2.2 Scalability Analysis

Fig. 9 and Fig. 10 show the impact of the scale of the heterogeneous edge $\mathcal{G}$. In Fig. 9, we can find that the average completion time achieved by all algorithms decreases as the edge server increases. The result is obvious because more *idle* servers available, more functions can be executed in parallel without delay. For all data batches, DPE achieves the best result. It is interesting to find that the gap between other algorithms and DPE gets widened when the scale of $\mathcal{G}$ increases. This is because the available simple paths become more and the data transmission time is reduced even further. Fig. 9 also shows the run time of different algorithms at the right y-axis. The results show that DPE has the minimum running time overhead.

Fig. 10 shows the impact of sparsity of $\mathcal{G}$. The horizontal axis is the overall number of simple paths $\mathcal{G}$. As it increases, $\mathcal{G}$ becomes more denser. Because DPE can reduce transmission time with optimal data splitting and mapping, average

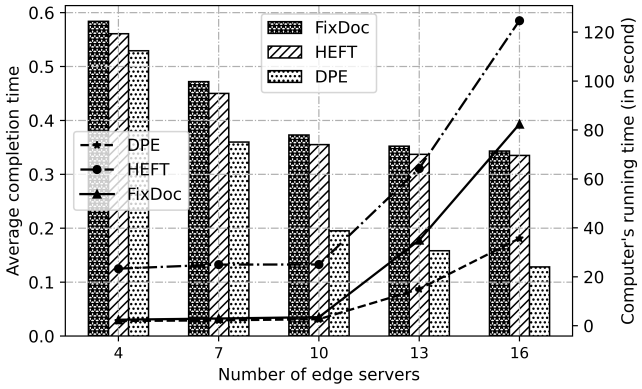

Fig. 9. Average completion time of DAGs and the computer's run time achieved by each algorithm under different numbers of edge servers.

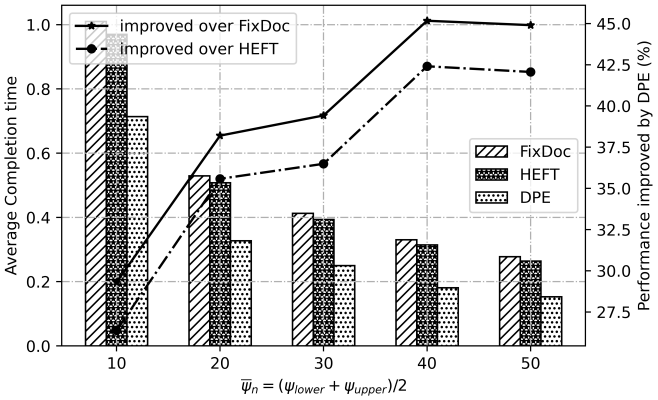

Fig. 11. Average completion time under different processing powers of servers.

completion time achieved by it decreases pretty evident. By contrast, FixDoc and HEFT have no obvious change.

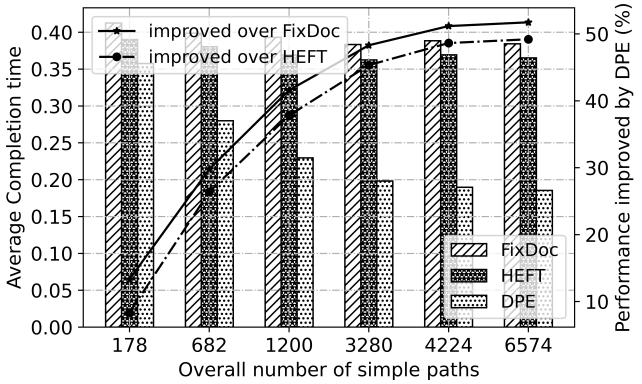

Fig. 10. Average completion time under different sparsities of $\mathcal{G}$.

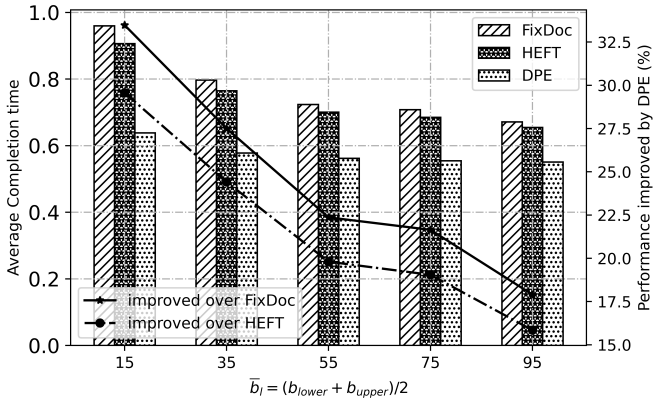

Fig. 12. Average completion time under different throughputs of links.

5.2.3 Sensitivity Analysis

Fig. 11 and Fig. 12 demonstrate the impact of system parameters, ψ_n and b_l. Notice that $\forall n \in \mathcal{N}, l \in \mathcal{L}$, ψ_n and b_l are sampled from the interval $[\psi_{lower}, \psi_{upper}]$ and $[b_{lower}, b_{upper}]$ uniformly, respectively. When the processing power and throughput increase, the computation and transmission time achieved by all algorithms are reduced. The results are immediate because a larger processing power directly reduce the computation time of functions while a larger throughput reduces the data transferring time directly. Even so, DPE achieves the smallest average completion time, which verifies the robustness of DPE adequately.

6 Related Works

In this section, we review related works on function placement and DAG scheduling in serverless edge computation systems.

Studying the optimal function placement is not new. Since cloud computing paradigm became popular, it has been extensively studied in the literature [26] [27] [28]. When bringing function placement into the paradigm of edge computing, especially for the IoT stream processing, different constraints, such as the response time requirement of latency-critical applications, availability of function instances on the heterogeneous edge servers, and the wireless and wired network throughput, etc., should be taken into consideration [29] [30] [31]. In edge computing, the optimal function placement strategy can be used to maximize the network utility [32], minimize the inter-node traffic [33] [34] [35], minimize the makespan of the applications [14] [15] [16], or even minimize the budget of application service providers [36].

In edge computing, the application is either modeled as an individual black-box or a DAG with complicated composite patterns. Considering that the IoT stream processing applications at the edge usually have dependent correlations between the fore-and-aft functions, dependent function placement problem has a strong correlation with DAG dispatching and scheduling. Scheduling algorithms for edge computation tasks have been extensively studied in recent years [19] [37] [38] [39]. In edge computing, the joint optimization of DAG scheduling and function placement is usually NP-hard. As a result, many works can only achieve a near optimal solution based on heuristic or greedy policy. For example, Gedeon et al. proposed a heuristic-based solution for function placement across a three-tier edge-fog-cloud heterogeneous infrastructure [40]. Cat et al. proposed a greedy algorithm for function placement by estimating the response time of paths in a DAG with queue theory [41]. Although FixDoc [14] can achieve the global optimal

function placement, the completion time can be reduced further by optimizing the stream mapping.

7 Conclusion

In this paper, we study the optimal dependent function embedding problem at the serverless edge. We first point out that proactive stream mapping and data splitting could have a strong impact on the makespan of DAGs with several use cases. Based on these observations, we design the DPE algorithm, which is theoretically verified to achieve the global optimality for an arbitrary DAG when the topological order of functions is given. DPE obtains the optimal stream mapping for each function pair with dependent relations. Extensive simulations based on the Alibaba cluster trace dataset verify that our algorithms can reduce the makespan significantly compared with a state-of-the-art function placement and scheduling methods, FixDoc, and a widely accepted algorithm, HEFT. The algorithms proposed in this paper is an offline algorithm. We leave the extension to online scenarios to our future work.

Acknowledgments

This work was partially supported by the Key Research Project of Zhejiang Province (No. 2022C01145) and the National Science Foundation of China (No. U20A20173 and No. 62125206). Schahram Dustdar's work is supported by the Zhejiang University De-qing Institute of Advanced technology and Industrilization (ZDATI).

References

[1] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, p. 44–54, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3368454

[2] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437378.3444367

[3] "Docker: Accelerate how you build, share and run modern applications." https://www.docker.com/.

[4] "Kubernetes: Production-grade container orchestration." https://kubernetes.io/.

[5] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. J. Yadwadkar, J. E. Gonzalez, R. A. Popa, I. Stoica, and D. A. Patterson, "Cloud programming simplified: A berkeley view on serverless computing," *CoRR*, vol. abs/1902.03383, 2019. [Online]. Available: http://arxiv.org/abs/1902.03383

[6] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[7] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.

[8] P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, and T. Taleb, "Survey on multi-access edge computing for internet of things realization," *IEEE Communications Surveys Tutorials*, vol. 20, no. 4, pp. 2961–2991, 2018.

[9] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar, and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, vol. 7, no. 8, pp. 7457–7469, 2020.

[10] 5G PPP Architecture Working Group, "View on 5g architecture: Version 3.0," https://doi.org/10.5281/zenodo.3265031, Feb 2020.

[11] Bahman Javadi, Jingtao Sun, and Rajiv Ranjan, "Serverless architecture for edge computing," pp. 249–264, 2020.

[12] P. Aditya, I. E. Akkus, A. Beck, R. Chen, V. Hilt, I. Rimac, K. Satzke, and M. Stein, "Will serverless computing revolutionize nfv?" *Proceedings of the IEEE*, vol. 107, no. 4, pp. 667–678, 2019.

[13] L. Baresi, D. Filgueira Mendonça, and M. Garriga, "Empowering low-latency applications through a serverless edge computing architecture," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 196–210.

[14] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proceedings of the International Symposium on Quality of Service*, ser. IWQoS '19, New York, NY, USA, 2019.

[15] S. Khare, H. Sun, J. Gascon-Samson, K. Zhang, A. Gokhale, Y. Barve, A. Bhattacharjee, and X. Koutsoukos, "Linearize, predict and place: Minimizing the makespan for edge-based stream processing of directed acyclic graphs," in *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, ser. SEC '19, New York, NY, USA, 2019, p. 1–14.

[16] Z. Zhou, Q. Wu, and X. Chen, "Online orchestration of cross-edge service function chaining for cost-efficient edge computing," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 8, pp. 1866–1880, 2019.

[17] "Istio: Connect, secure, control, and observe services." https://istio.io/latest/.

[18] "Alibaba cluster trace program," https://github.com/alibaba/clusterdata.

[19] H. Topcuoglu, S. Hariri, and Min-You Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[20] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 94–100, 2017.

[21] S. Vassilaras, L. Gkatzikis, N. Liakopoulos, I. N. Stiakogiannakis, M. Qi, L. Shi, L. Liu, M. Debbah, and G. S. Paschos, "The algorithmic aspects of network slicing," *IEEE Communications Magazine*, vol. 55, no. 8, pp. 112–119, 2017.

[22] "Kubeedge: An open platform to enable edge computing." https://kubeedge.io/en/.

[23] "Envoy: An open source edge and service proxy, designed for cloud-native applications." https://www.envoyproxy.io/.

[24] GSM Association, "Official document ng.116 - generic network slice template v4.0," https://www.gsma.com/newsroom/wp-content/uploads//NG.116-v4.0-2.pdf, Nov 2020.

[25] J. Fearnley and R. Savani, "The complexity of the simplex method," in *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, 2015, pp. 201–208.

[26] G. T. Lakshmanan, Y. Li, and R. Strom, "Placement strategies for internet-scale data stream systems," *IEEE Internet Computing*, vol. 12, no. 6, pp. 50–60, 2008.

[27] L. Tom and V. R. Bindu, "Task scheduling algorithms in cloud computing: A survey," in *Inventive Computation Technologies*, S. Smys, R. Bestak, and Á. Rocha, Eds. Cham: Springer International Publishing, 2020, pp. 342–350.

[28] B. Addis, D. Belabed, M. Bouet, and S. Secci, "Virtual network functions placement and routing optimization," in *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, 2015, pp. 171–177.

[29] H. Badri, T. Bahreini, D. Grosu, and K. Yang, "Energy-aware application placement in mobile edge computing: A stochastic optimization approach," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 04, pp. 909–922, apr 2020.

[30] M. Nardelli, V. Cardellini, V. Grassi, and F. Presti, "Efficient operator placement for distributed data stream processing applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 08, pp. 1753–1767, aug 2019.

[31] Z. Ning, P. Dong, X. Wang, S. Wang, X. Hu, S. Guo, T. Qiu, B. Hu, and R. K. Kwok, "Distributed and dynamic service placement in pervasive edge computing networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 06, pp. 1277–1292, jun 2021.

[32] M. Leconte, G. S. Paschos, P. Mertikopoulos, and U. C. Kozat, "A resource allocation framework for network slicing," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 2177–2185.

[33] J. Xu, Z. Chen, J. Tang, and S. Su, "T-storm: Traffic-aware online scheduling in storm," in *2014 IEEE 34th International Conference on Distributed Computing Systems*, 2014, pp. 535–544.
[34] L. Liu, S. Guo, G. Liu, and Y. Yang, "Joint dynamical vnf placement and sfc routing in nfv-enabled sdns," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2021.
[35] S. Yang, F. Li, S. Trajanovski, X. Chen, Y. Wang, and X. Fu, "Delay-aware virtual network function placement and routing in edge clouds," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 445–459, 2021.
[36] L. Chen, J. Xu, S. Ren, and P. Zhou, "Spatio–temporal edge service placement: A bandit learning approach," *IEEE Transactions on Wireless Communications*, vol. 17, no. 12, pp. 8388–8401, 2018.
[37] Y. Kao, B. Krishnamachari, M. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE Transactions on Mobile Computing*, vol. 16, no. 11, pp. 3056–3069, 2017.
[38] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, 2018, pp. 37–45.
[39] J. Meng, H. Tan, C. Xu, W. Cao, L. Liu, and B. Li, "Dedas: Online task dispatching and scheduling with bandwidth constraint in edge computing," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, 2019, pp. 2287–2295.
[40] J. Gedeon, M. Stein, L. Wang, and M. Muehlhaeuser, "On scalable in-network operator placement for edge computing," in *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, 2018, pp. 1–9.
[41] X. Cai, H. Kuang, H. Hu, W. Song, and J. Lü, "Response time aware operator placement for complex event processing in edge computing," in *Service-Oriented Computing*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, pp. 264–278.

Shuiguang Deng is currently a full professor at the College of Computer Science and Technology in Zhejiang University, China, where he received a BS and PhD degree both in Computer Science in 2002 and 2007, respectively. He previously worked at the Massachusetts Institute of Technology in 2014 and Stanford University in 2015 as a visiting scholar. His research interests include Edge Computing, Service Computing, Cloud Computing, and Business Process Management. He serves for the journal IEEE Trans. on Services Computing, Knowledge and Information Systems, Computing, and IET Cyber-Physical Systems: Theory & Applications as an Associate Editor. Up to now, he has published more than 100 papers in journals and refereed conferences. In 2018, he was granted the Rising Star Award by IEEE TCSVC. He is a fellow of IET and a senior member of IEEE.

Hailiang Zhao received the B.S. degree in 2019 from the school of computer science and technology, Wuhan University of Technology, Wuhan, China. He is currently pursuing the Ph.D. degree with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. He has been a recipient of the Best Student Paper Award of IEEE ICWS 2019. His research interests include edge computing, service computing and machine learning.

Zhengzhe Xiang received the B.S. and Ph.D. degree of Computer Science and Technology in Zhejiang University, Hangzhou, China. He was previously a visiting student worked at the Karlstad University, Sweden in 2018. He is currently a Lecturer with Zhejiang University City College, Hangzhou, China. His research interests lie in the fields of Service Computing, Cloud Computing, and Edge Computing.

Cheng Zhang received the MS degree in electrical engineering from Zhejiang University, China, in 2013, Currently, he is working toward the PhD degree in computer science and technology at Zhejiang University. His research interests include edge computing and edge intelligence.

Rong Jiang is currently a deputy dean of Institute of Intelligence Applications, Distinguished Professor and Doctoral Supervisor at Yunnan University of Finance and Economics, Kunming, China. He received his Ph.D. degree in system analysis and integration from the School of Software, Yunnan University, China. He is an Excellent Professional and Technical Talents with Outstanding Contributions in Yunnan Province, an Expert Enjoying Special Government Allowances in Yunnan Province, a Young and Middle-aged Academic and Technical Leaders in Yunnan Province, an Excellent Teacher in Yunnan Province, the Director of Key Laboratory of Service Computing and Security Management of Yunnan Provincial Universities, and the Director of Kunming Key Laboratory of Information Economy and Information Management. His current research interests include cloud computing, big data, block chain, AI application and information management, digital economy and software engineering. Up to now, he has published more than 60 papers. He has received more than 70 prizes in recent years.

Ying Li is an Associate Professor with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China. His research interests include Service Computing, Cloud Computing, and Data Science.

Jianwei Yin received the Ph.D. degree in computer science from Zhejiang University (ZJU) in 2001. He was a Visiting Scholar with the Georgia Institute of Technology. He is currently a Full Professor with the College of Computer Science, ZJU. Up to now, he has published more than 100 papers in top international journals and conferences. His current research interests include service computing and business process management. He is an Associate Editor of the IEEE Transactions on Services Computing.

Schahram Dustdar is a Full Professor of Computer Science (Informatics) with a focus on Internet Technologies heading the Distributed Systems Group at the TU Wien. He is Chairman of the Informatics Section of the Academia Europaea (since December 9, 2016). He is elevated to IEEE Fellow (since January 2016). From 2004-2010 he was Honorary Professor of Information Systems at the Department of Computing Science at the University of Groningen (RuG), The Netherlands.

From December 2016 until January 2017 he was a Visiting Professor at the University of Sevilla, Spain and from January until June 2017 he was a Visiting Professor at UC Berkeley, USA. He is a member of the IEEE Conference Activities Committee (CAC) (since 2016), of the Section Committee of Informatics of the Academia Europaea (since 2015), a member of the Academia Europaea: The Academy of Europe, Informatics Section (since 2013). He is recipient of the ACM Distinguished Scientist award (2009) and the IBM Faculty Award (2012). He is an Associate Editor of IEEE Transactions on Services Computing, ACM Transactions on the Web, and ACM Transactions on Internet Technology and on the editorial board of IEEE Internet Computing. He is the Editor-in-Chief of Computing (an SCI-ranked journal of Springer).

Albert Y. Zomaya is Chair Professor of High-Performance Computing & Networking in the School of Computer Science and Director of the Centre for Distributed and High-Performance Computing at the University of Sydney. To date, he has published > 600 scientific papers and articles and is (co-)author/editor of > 30 books. A sought-after speaker, he has delivered > 190 keynote addresses, invited seminars, and media briefings. His research interests span several areas in parallel and distributed computing and complex systems. He is currently the Editor in Chief of the ACM Computing Surveys and served in the past as Editor in Chief of the IEEE Transactions on Computers (2010-2014) and the IEEE Transactions on Sustainable Computing (2016-2020).

Professor Zomaya is a decorated scholar with numerous accolades including Fellowship of the IEEE, the American Association for the Advancement of Science, and the Institution of Engineering and Technology (UK). Also, he is an Elected Fellow of the Royal Society of New South Wales and an Elected Foreign Member of Academia Europaea. He is the recipient of the 1997 Edgeworth David Medal from the Royal Society of New South Wales for outstanding contributions to Australian Science, the IEEE Technical Committee on Parallel Processing Outstanding Service Award (2011), IEEE Technical Committee on Scalable Computing Medal for Excellence in Scalable Computing (2011), IEEE Computer Society Technical Achievement Award (2014), ACM MSWIM Reginald A. Fessenden Award (2017), and the New South Wales Premier's Prize of Excellence in Engineering and Information and Communications Technology (2019).