Partitioning Stateful Data Stream Applications in Dynamic Edge Cloud Environments

Shaoshuai Ding, Lei Yang, Jiannong Cao, *Fellow, IEEE,* Wei Cai, Mingkui Tan and Zhenyu Wang

Abstract—Computation partitioning is an important technique to improve the application performance by selectively offloading some computations from the mobile devices to the nearby edge cloud. In a dynamic environment in which the network bandwidth to the edge cloud may change frequently, the partitioning of the computation needs to be updated accordingly. The frequent updating of partitioning leads to high state migration cost between the mobile side and edge cloud. However, existing works don't take the state migration overhead into consideration. Consequently, the partitioning decisions may cause significant network congestion and increase overall completion time tremendously. In this paper, with considering the state migration overhead, we propose a set of novel algorithms to update the partitioning based on the changing network bandwidth. To the best of our knowledge, this is the first work on computation partitioning for stateful data stream applications in dynamic environments. The algorithms aim to alleviate the network congestion and minimize the make-span through selectively migrating state in dynamic edge cloud environments. Extensive simulations show our solution not only could selectively migrate state but also outperforms other classical benchmark algorithms in terms of make-span. The proposed model and algorithms will enrich the scheduling theory for *stateful* tasks, which has not been explored before.

Index Terms—edge cloud; computation partitioning; stateful data stream applications

1 Introduction

Edge computing is to enable cloud computing technologies from the traditional Internet data centers to the network edge for low latency data accessing and real-time data processing [7]. As an abstraction of computing resources at the edge, edge clouds are usually distributed geographically at the locations which are closer to the end users such as on the cellular base stations and the wireless local area networks. The general forms of edge computing include Cloudlets [9] and Foglets [10] and even a small cluster of limited devices [8]. Compared with traditional cloud data centers, edge cloud is more lightweight and resource constrained [11]. With an increasing deployment of edge clouds in today's network infrastructures, computation partitioning is considered as an efficient technique to improve the mobile application performance by selectively offloading some computation from the mobile devices to the nearby edge cloud [4] [12] [18].

There exist many related works on computation partitioning for achieving different purposes such as reducing the execution time [3], saving the energy consumption on the end device [1] and the data transmission overhead to the cloud [2] [4]. These works have different modeling approach for the applications. Typical application models include the method call tree for the procedure-oriented programs, service invocation graph for the service-oriented applications and the data flow graph for the data stream applications. Among these types of applications, data stream applications have been increasingly concerned such as the augmented reality and object tracking [5]. The application is composed of a set of functional modules with the data flowing through them. The partitioning of the data stream applications aims to decide for each incoming data frame which functions are executed locally and which others are executed on the edge cloud [4] [12]. However, existing works do not consider the partitioning of *stateful* data stream applications. We define the data stream application as being stateful if it includes stateful function modules. By stateful function module, if one data frame flows through it, a 'footprint' will be left on the device at which the module is processed. This 'footprint', also named by state, is needed by the processing of the next data frame. Many applications such as object tracking pertain to the stateful applications.

Partitioning stateful data stream applications is challenging particularly in dynamic edge cloud environments where the network connection to the edge cloud changes frequently and even disconnection could occur. Because of dynamics of network connection, the partitioning of the application needs to be updated accordingly, which would cause state migration between the mobile device and the edge cloud. Therefore, we need to partition computations through selectively migrating state to alleviate network congestion. Existing works on computation partitioning consider the application with stateless function modules. When they are applied in partitioning of stateful applications, high state migration overhead happens in the network, which may lead to network congestion and result in a long completion time of applications. That is why we need

- *Lei Yang is the corresponding author. He is with School of Software Engineering, South China University of Technology, Guangzhou, China. E-mail: sely@scut.edu.cn*
- *S. Ding, M. Tan and Z. Wang are with School of Software Engineering, South China University of Technology, Guangzhou, China. E-mail:201721045787@mail.scut.edu.cn, {mingkuitan, wangzy}@scut.edu.cn*
- *J. Cao is with Department of Computing, The Hong Kong Polytechnic University, Hong Kong. E-mail: csjcao@comp.polyu.edu.hk*
- *W. Cai is with the School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen, China. Email: caiwei@cuhk.edu.cn*

to specially design new approach for partitioning stateful application, aiming to balance a good partitioning with low completion time and the additional state migration time over the network.

In this paper, we develop a set of efficient algorithms to solve the problem of partitioning stateful data stream applications, with the aim of alleviating network congestion and minimizing the make-span through selectively migrating state. In particular, we design a novel algorithm, namely Score Matrix-based Heuristic (SM-H), to solve the one-shot problem, which updates the partitioning of the current arriving data frame when the edge network environment changes. SM-H uses a *matrix* to record the benefit score of adjusting the execution position of each module, and then always select the module with the greatest score to adjust. The adjustment is done iteratively until none of the modules has a positive score, which means that adjusting anyone of modules will cause the increase of the completion time. On basis of the one-shot SM-H algorithm, we further extend to solve the partitioning problem with multiple steps look ahead.

We evaluate the proposed algorithms via extensive simulations and compare them with several benchmark methods including the sequential adjustment that is a naive greedy heuristic, list scheduling that is a classical scheduling method in parallel and distributed computing, genetic algorithm and so on. The results show that the proposed algorithms outperform the benchmark algorithms in terms of the make-span. We summarize the contributions of this paper as follows.

- To the best of our knowledge, we are the first to study the partitioning problem for the *stateful* data stream applications. The problem models can be extended into the scheduling of *stateful* tasks on distributed processors which so far has not been studied in the area of task scheduling.
- We develop a new algorithm for partitioning the *stateful* data stream applications. The algorithm enriches the scheduling theory and methodology for *stateful* application tasks.
- We evaluate the proposed algorithm through extensive simulations, and the results show the proposed SM-H outperforms the benchmark algorithms in terms of the make-span.

2 System Model and Problem Formulation

In this section, we will introduce the system model and the definition of computation partitioning for stateful data stream applications in dynamic edge cloud environments. Furthermore, the functionalities in object tracking is presented as a realistic example of stateful data stream applications.

2.1 System Model

In our paper, we focus on the partitioning for the *stateful* data stream applications. These applications take the streaming data frames as input, perform a series of operations onto each incoming data frame, and then output the results continuously. The input data frames are sampled periodically from the sensors on the mobile device. In addition, some operation modules of the application, named *stateful* modules, generate state during execution. Every *stateful* module is left with a 'footprint' (state) after processing a data frame. The footprint (state) is needed by the processing of the next data frame. These *stateful* modules need to rely on the state during subsequent execution when the next data frame comes. In addition, for different stateful applications, the proportion of *stateful* modules and the size of the state are diverse.

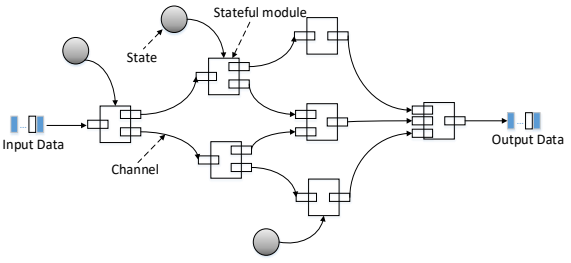


Fig. 1. The Model for *Stateful* Data Stream Applications

We model *stateful* data stream application as a data flow graph. It consists of a set of modules and a set of edges as shown in Fig.1. The set of modules include *stateful* modules and stateless modules. Each of them takes the output data from all its precedent edges, and performs particular operations and then output the data into its successive edges. The state is associated with a specific module and is generated at the device where the module is executed. We use $(\mathbb{V}, \mathbb{E})$ to denote the set of modules and edges of the *stateful* application graph, where $\mathbb{V}_{state} \in \mathbb{V}$ represents the set of *stateful* modules. We define the streaming data transferring along the edge as *data flow*

In our model, the network bandwidth between a mobile device and the edge cloud is dynamically changing. We abstract the total network bandwidth as a set of virtual network channels to simplify the model referring to the Frequency Division Multiplexing (FDM) in network transmission protocols. Each channel is assumed to have the same network bandwidth. Our model and method can also be extended to heterogeneous network channels. The data transmission in the network include the data flows and the state migration. The scheduling of data flows in the network use the First-Come-First-Served policy. If multiple modules are allocated to the mobile device or edge cloud server simultaneously, we execute these modules in a Shortest-Job-First manner.

The metric to measure the computation partitioning performance of *stateful* data stream applications is $make\text{-}span$.

- **Make-span** is defined as the completion time for processing one data frame. In the stateful data flow graph, the weights of the modules are represented as the execution time on the mobile device and on the edge cloud server for processing one data frame respectively. The weight of the edge is represented as the transmission time, and the weight of the state is expressed by its migration time. The make-span is defined as the total completion time for processing

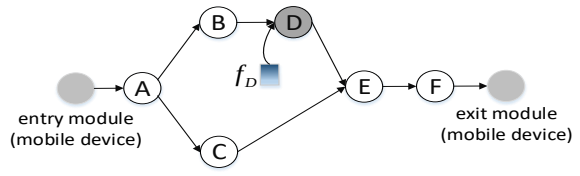


Fig. 2. STC algorithm: a realistic object tracking example

one data frame.

$Make\text{-}span$ is usually the primary metric for the performance of *stateful* data stream applications and directly affects the user experience. Since the state migration overhead could bring network congestion potentially, selectively migrating state is important for minimizing the $make\text{-}span$ in dynamic edge cloud environments.

2.2 A Realistic Example in Object Tracking

Object tracking can be applied to the fields of traffic and reconnaissance, which collects images through cameras and makes real-time processing. Considering the limited processing power of camera terminals, when the data processing rate cannot meet the requirements, edge cloud servers are needed to assist the acceleration. Then the models and methods in our work can be applied in this scenario.

In order to clearly describe the practical significance of the problem proposed in this paper, we discuss *stateful* data stream applications with a realistic object tracking example. In this example, object tracking is implemented by STC (Spatio-Temporal Context) algorithm mentioned in the paper [5]. The STC algorithm is based on a Bayesian computation framework to learn the spatio-temporal relationship between the object of interest and its local context region. Furthermore, the function modules and execution flows of the STC algorithm is shown in Fig.2, and the data flow between adjacent modules is used to transfer the calculation result of the previous module.

As shown in Fig.2, module A captures the t-th data frame, and module B calculates the context prior probability $P(c(z)|o)$. Module C calculates the spatial context model h_t^{sc}. Module D is a *stateful* function module. The state f_D is a spatio-temporal context model H_{t-1}^{stc} at the (t-1)-th frame for updating the spatio-temporal context model H_t^{stc} of the t-th frame in module D, which is represented as $H_t^{stc} = (1-\rho)H_{t-1}^{stc} + \rho h_t^{sc}$. Module E calculates the confidence map of the t-th frame based on Bayesian framework and FFT, which is represented as $m_t(x) = H_t^{stc}(x) \otimes P(c(z)|o) = F^{-1}(F(H_t^{stc}(x)) \cdot F(P(c(z)|o)))$. The module F estimates the best object location of the t-th frame by $x_t^* = \arg\max_{x \in \Omega_c(x_{t-1}^*)} m_t(x)$. The tracking result returns to mobile device finally.

The above is the stateful data flow process of the STC algorithm in object tracking, which can be abstracted as a *stateful* data stream application. In this paper, we study how to partition the *stateful* data stream applications to alleviate network congestion and minimize the make-span in dynamic edge cloud environments.

TABLE 1
Mathematical notations in this paper

V	the set of modules in the application graph;
E	the set of edges in the application graph;
V_{state}	the set of stateful modules in the application graph;
N^η	the number of network channels in the time η;
B	the network bandwidth of each channel;
M	the mobile device, supposing there is only one mobile device;
C	the edge cloud server, supposing there is only one server at the edge cloud;
i, j	index of the module in the application graph;
f_i	the amount of state generated by task i;
(i, j)	the edge in the application graph;
k	index of the network channel;
m_i	the execution time of the module i on the mobile device in processing a unit of data;
c_i	the execution time of the module i on the cloud processor in processing a unit of data;
$\overrightarrow{x}^\eta$	the computation partitioning of the stateful data stream application at time point η;
x_i^η	a binary variable indicating whether the module i is allocated to the cloud side in the time η;
t_r^i	the release time of module i;
σ_i	the time when the module i starts to execute;
λ_i	the end execution time of module i;
$t_s^{f_i}$	the start transmission time of state f_i;
$t_f^{f_i}$	the end transmission time of state f_i;
$D_{(i,j)}$	the amount of data that needs to be transmitted from module i to module j in processing one unit of data;
$t_r^{(i,j)}$	the release time of data flow that transferring in the edge (i, j);
$t_s^{(i,j)}$	the time when the edge (i, j) starts to transmit;
$t_f^{(i,j)}$	the end transmission time of edge (i, j);
t_e^k	the earliest available time of channel k;
t_e^M	the earliest available time of mobile device;
t_e^C	the earliest available time of edge server;
U^η	the set of modules which update the execution position at time η;
G^η	the set of cross edges at time η;

2.3 Problem Formulation

The problem is modeled as follows. Suppose there exits only one mobile user and only one closest available edge cloud. The application launched from mobile device processes the incoming data frames one by one. The execution time of module i on mobile device and edge cloud is m_i and c_i respectively. The execution mode of modules is assumed to be non-preemptive. The edge cloud may come from Cloudlets [9] and Foglets [10] and even a small cluster of limited devices [8]. We use f_i to represent the amount of state generated by stateful module i. Let B represent the bandwidth of each channel. The decision variable x_i^η indicates whether the module i is offloaded to the edge cloud at time point η. If $x_i^\eta = 1$, it means that module i is offloaded to the edge cloud; otherwise it means the module i is executed locally. We use the $cross\ edge$ to indicate the

data flow whose two connective modules are executed at the different sides.

Decision variables. Given the application graph and its relevant parameters $(\mathbb{V}, \mathbb{E})$, the computation partitioning $\overrightarrow{x}^{\eta-1}$ at time point $\eta-1$, and the number of network channels that has changed at time point η. The problem is to update the computation partitioning according to current network bandwidth, i.e., to decide whether to update the execution position of each module at new time point η.

Let σ_i denote the start execution time of module i. The completion time λ_i of module i can be formulated as $\sigma_i + x_i^\eta \cdot c_i + (1 - x_i^\eta) \cdot m_i$. Let (i,j) denote the data flow from the module i to the module j. The release time of data flow (i,j) is defined as the completion time of its precedent module. We assume that non-preemptive transmission strategy is utilized for scheduling the $cross\,edge$ data flow and state migration onto the network channels.

We use piecewise function $y_{f_i}(t)$ to represent the network bandwidth allocated to state migration, which is defined as Equation(1)

$$y_{f_i}(t) = \begin{cases} B & if\ t \in [t_s^{f_i}, t_f^{f_i}] \\ 0 & otherwise \end{cases}, \tag{1}$$

where $t_s^{f_i}$ and $t_f^{f_i}$ represent the start migration time and end migration time of state f_i respectively. Similarly, we use piecewise function $y_{(i,j)}(t)$ to represent the network bandwidth allocated to transmission of $cross\,edge\ (i,j)$, which is defined as Equation(2)

$$y_{(i,j)}(t) = \begin{cases} B & if\ t \in [t_s^{(i,j)}, t_f^{(i,j)}] \\ 0 & otherwise \end{cases}, \tag{2}$$

where $t_s^{(i,j)}$ and $t_f^{(i,j)}$ represent the start transmission time and end transmission time of $cross\,edge\ (i,j)$ respectively.

Objective. The objective is to minimize the $make\text{-}span$. In order to formulate the objective conveniently, we add two virtual modules into the application graph including the $entry\,module$ denoted by $i=0$, and the $exit\,module$ denoted by $i=n+1$. We use U^η to represent the set of modules which update the execution position at time η. Since the application normally gets input data frames from the mobile device and is required to output the result to the mobile device as well, we set $x_0^\eta = 0$ and $x_{n+1}^\eta = 0$. With the two virtual modules added, σ_0 indicates the start execution time of the application, which is usually equal to the release time of the user's application. σ_{n+1} indicates the end time of the application. For each module i in U^η, if it is a stateful module, the migration of state f_i occurs. Therefore, the objective can be formulated by

$$\min_{\overrightarrow{x}^\eta} (\sigma_{n+1} - \sigma_0). \tag{3}$$

Constraint on the dependency of modules. The execution time of the modules should satisfy the dependency constraint in the application graph. The module can not be executed until all its precedent modules are finished. We assume that the module's execution is non-preemptive. We can represent the execution interval of the i-th module by $[\sigma_i, \lambda_i]$. The dependency constraint among the modules is formulated by

$$\lambda_i \le \sigma_j, \quad \forall (i,j) \in V. \tag{4}$$

Constraint on the state migration. The migration of state should satisfy the dependency with its relevant stateful module. Since the stateful module needs to rely on the state when new data frame comes, the end time of the state migration should be earlier than the start execution of its relevant module. Moreover, when the module i updates its execution position at time point η, the start migration time of the state f_i should be later than the time η. We use $t_s^{f_i}$ to indicate the start migration time of state f_i, and the end migration time $t_f^{f_i}$ is represented as $t_s^{f_i} + f_i/B + \omega$, where ω is the propagation delay. The constraints on the state migration are formulated by

$$t_s^{f_i} \ge \eta, \quad \forall i \in U^\eta,\ and\ i \in V_{state}, \tag{5}$$

$$t_f^{f_i} \le \sigma_i, \quad \forall i \in U^\eta,\ and\ i \in V_{state}. \tag{6}$$

Constraint on the network bandwidth. We use N^η to represent the number of network channels at time point η. Let G^η denote the set of $cross\,edges$ at time point η. At each time point $t \in [\sigma_0, \sigma_{n+1}]$, the sum of allocated bandwidth for each $cross\,edge$ transmission and state migration should be less than $N^\eta \cdot B$. The constraint is formulated by

$$\sum_{(i,j)\in G^\eta} y_{(i,j)}(t) + \sum_{i\in U^\eta,\ i\in V_{state}} y_{f_i}(t) \le N^\eta \cdot B, \quad \forall t \in [\sigma_0, \sigma_{n+1}]. \tag{7}$$

Definition 1 *Stateful data stream application Computation Partitioning Problem (SCPP).* Given the application graph and its associated parameters $(\mathbb{V}, \mathbb{E})$. The number of network channels N^η that have changed at time point η, the network bandwidth B of each network channel, and the computation partitioning $x^{\eta-1}$ at time point $\eta-1$, the problem is to update the computation partitioning at time η, and migrate state for alleviating network congestion, so as to minimize the make-span. The problem is formulated as follows.

$$\min_{\overrightarrow{x}^\eta} (\sigma_{n+1} - \sigma_0), \tag{8}$$

s.t. (4), (5), (6), (7).

An Example. Fig.3 illustrates an example of the computation partitioning problem for stateful data stream applications when network environment deteriorates. In this example, we assume that the ratio of the module's execution time in the edge cloud to the mobile device is 1:2. The execution time of the module i on the mobile device and on the edge cloud server is described as $[m_i, c_i]$. The weight on edge indicates the transmission time. The transmission time is equal to zero if the two connecting modules are executed on the same resource. All the weights are measured under the assumption that the module or edge occupies the device or network channel exclusively.

In the first graph, assuming the previous partitioning at time $\eta-1$ is that modules A and B are executed on the edge cloud server, and modules C and D are executed

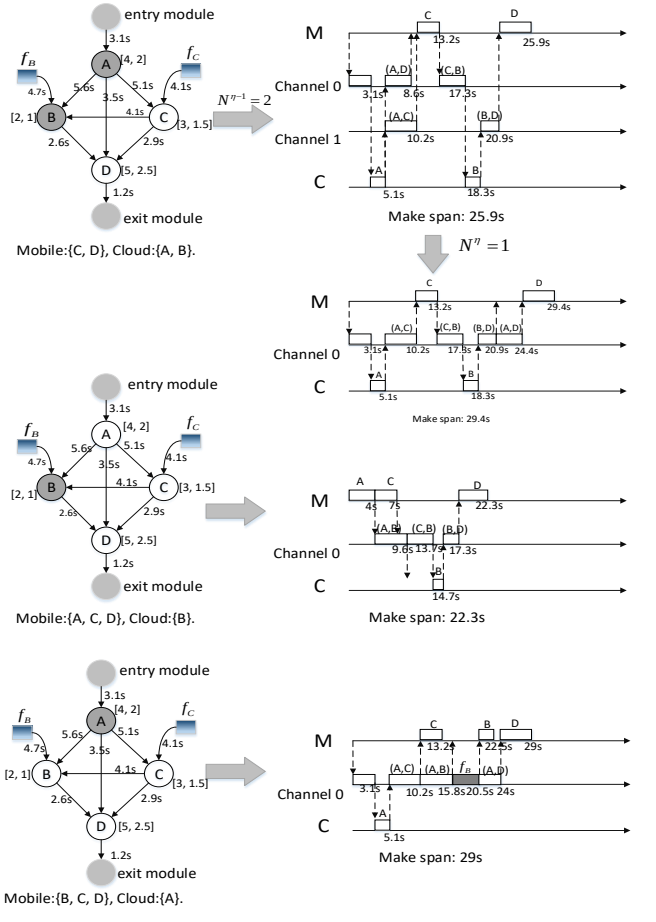


Fig. 3. An example to illustrate the updating process of computation partitioning for stateful data stream applications

in the mobile device. We can get through First-Come-First-Served (FCFS) policy that the make-span at time point $\eta - 1$ is $25.9s$. In the second graph, the network environment deteriorates and the number of network channels changes from 2 to 1 at time point η. The make-span becomes $29.4s$ as a result. We need to update the computation partitioning of this stateful data stream application based on the current network environment, so as to minimize the make-span. In the third graph, we assume that updating the execution position of module A and the make-span becomes $22.3s$. In the fourth graph, we assume that updating the execution position of module B and the make-span becomes $29s$. As we can see, the migration overhead of state f_B increases the make-span. So updating module A first could bring more benefits than module B and reduce state migration meanwhile. Considering the priority of updating the modules can have a large impact on the results. In this paper, We discuss the strategy of adjusting the module first which can reduce the make-span greatly.

2.4 Problem Uniqueness and Challenges

In general cloud offloading, the cloud that accommodates the offloaded tasks has abundant computing resources, while the edge cloud has limited computing resources. Due to this property, the offloading decision needs to be jointly considered with the resource scheduling at the server side such as to optimize the performance. Moreover, the network model for cloud offloading includes the wireless access and the Internet connection. The bandwidth of Internet connection is a dominant factor to determine the data transmission cost, because it is relatively constrained to the access network. However, the network model for edge cloud offloading mainly considers the wireless access network. The bandwidth of wireless access often changes frequently due to the users mobility. Thus, the offloading decisions in edge cloud scenario needs to be updated frequently. The offloading problem in edge cloud pertains to be a dynamic decision problem which aims to make adaptive decisions according to the changing bandwidth, while in most of existing cloud offloading problem like MAUI [19], the offloading is modeled as a static optimization problem. The offloading decision can be obtained by an 0-1 programming solver. Due to the two differences above, the offloading problem in edge cloud is much more challenging than previous cloud offloading.

The stateful migration occurs in MapReduce [29]. However, the state migration in MapReduce occurs due to the data dependence among the tasks within a job. For example, the intermediate data from Map task is needed by the Reduce task. However, MapReduce does not model the correlation of different jobs. The tasks in MapReduce are stateless. A map/reduce task has certain life cycle only for a particular job. The tasks are terminated when a job is finished. As a new job is released, new tasks are launched for the job. In our work, we consider the application model of stateful streaming application. It is different from the general task model in MapReduce. Although an extended version of MapReduce, namely MapReduce Online, is proposed for processing streaming data. The workload in MapReduce Online fits into the task model in our work. However, the work does not propose the scheduling method particularly for stateful streaming application. Moreover, the resources in a MapReduce cluster are relatively static compared to the resources in mobile edge cloud. With static resources, the scheduling of tasks does not need to be changed frequently. That is why existing scheduling works for big data cluster rarely studied the dynamic scheduling problem.

MAUI [19] makes an offloading decision based on the estimation of the network bandwidth and the profiling data of the program. It is designed for executing a work-flow program rather than the data stream application. In MAUI, once the program is started, it generates an offloading decision by solving an 0-1 integer programming problem. So the offloading problem in MAUI pertains to a static one-shot optimization problem. In data stream application, the program would be repeatedly executed for processing the periodically arriving data. The data have temporal relationship. The processing of a data unit at one time would leave some information (state) which is required to process the data arriving at the next time. As a result, the offloading decisions at different time slots are correlated in data stream application. Due to the correlation, the offloading problem in our paper naturally pertains to a dynamic decision problem. The dynamic offloading problem is more challenging than previous one-shot offloading problem.

In addition, our problem is different from the classical network flow problem. Since the weight of each data flow

reflects the sum of the transmission delay and propagation delay in the current network environment, our problem does not need to meet the Capacity Limit Conditions of the network flow problem. Correspondingly, the constraint on the network bandwidth in Section 2.3 is also different from the Capacity Limit Conditions. Because the sum of data amount received and the sum of data amount transmitted by each module are not necessarily the same, it is also different from the Equilibrium Conditions of the network flow problem. Additionally, the optimization goal of network flow problems is usually to solve the maximum flow problem, which can usually be solved by linear programming, while our problem is a scheduling problem. The optimization objective with make-span is usually affected by multiple constraints, such as the complexity of the DAG itself, and the dynamics of the edge cloud network, which is an NP-hard problem.

3 Solutions to SCPP

In this section, we introduce the SM-H algorithm and the RSM-H algorithm to solve the one shot scheduling problem and the Δt-steps look ahead problem respectively. Moreover, we represent the execution process and the time complexity of both algorithms.

3.1 Solving One Shot Optimization

We propose a new heuristic algorithm, named Score Matrix-based Heuristic (SM-H), to solve the Problem. Given the computation partitioning $x^{\eta-1}$ at time $\eta-1$ and the predicted number of channels N^{η} at time η, the SM-H algorithm greedily adjusts the module with the greatest score until the performance can not be improved. Specifically, the network trace prediction could resort to the Network Status Prediction method [12]. SM-H consists of two phases: adjustment and update. During the adjustment, we define the adjustment score of each module as the reduction of the make-span if the execution position of the module is changed. The *score matrix* records the adjustment score of each module. We always select the module with the greatest score and adjust this module's execution position. Then in the update phase, the score matrix will be recomputed according to the latest computation partitioning. The above two phases are repeated alternatively until there is no positive score in the score matrix. Then we obtain the new computation partitioning x^{η} at time η. The pseudo-code of the Score Matrix-based Heuristic is shown in Algorithm 1.

Algorithm 1: Score Matrix-based Heuristic Algorithm

Input : $\Omega=\{V,E\}$; the set of stateful modules V_{state}; one mobile device M; one edge cloud server C; $x^{\eta-1}$; N^{η} and each channel's bandwidth B.
Output: x^{η}

1 Process each module $i \in V$ by order of Breadth-First Traversal of the data flow graph Ω.
2 Initialize a $1 \times n$ scoring matrix SM;
3 **for** *each module* $i \in V$ **do**
4 Set the original make-span $\alpha_1 \leftarrow Max\{\lambda_i, \forall i \in V\}$;
5 **if** *updating the module i's execution location* **then**
6 Get the predecessor cross edge set G_i of module i;
7 Use the First-Come-First-Served policy to transmit the $cross\ edge\ (j,i) \in G_i$ and update its *transmission time list*;
8 **if** $i \in V_{state}$ **then**
9 Migrate the state f_i in advance and update its *migration time list*;
10 Get the predecessor module set P_i of module i;
11 Set the release time of module i, $t_r^i \leftarrow Max\{t_f^{f_i}, Max\{t_f^{(j,i)}, \forall (j,i) \in G_i\}, Max\{\lambda_j, \forall j \in P_i\}\}$;
12 **if** $X_i^{\eta} == 0$ **then**
13 Execute the module i using FCFS policy and set
14 $\sigma_i \leftarrow Max\{t_e^M, t_r^{(i)}\}$;
15 $\lambda_i \leftarrow \sigma_i + x_i^{\eta} \cdot c_i + (1-x_i^{\eta}) \cdot m_i$;
16 Update $t_e^M \leftarrow \lambda_i$;
17 **else**
18 Use the same scheduling method to execute module i on C as lines 13-16;
19 Execute the remaining modules and get the make-span $\alpha_2 \leftarrow Max\{\lambda_s, \forall s \in V\}$;
20 Set $SM[i] \leftarrow \alpha_1 - \alpha_2$;
21 Get the greatest score β and the relevant module i;
22 **while** $\beta > 0$ **do**
23 Adjusting the module i's execution position as $x_i^{\eta} = 1 - x_i^{\eta-1}$;
24 Recompute the scoring matrix SM as lines 3-20;
25 Get the greatest score β and the related module i;
26 return x^{η};

The module's adjustment score plays an important role in the algorithm SM-H. And then we first define the data structures and relevant terminologies in calculating the make-span as follows.

- **Execution time list**. Execution time list is used to represent the execution process of a module. It contains the module's release time, the start execution time, the end execution time and its execution place. We use the list $[t_r^i, \sigma_i, \lambda_i, x_i^{\eta}]$ to represent the execution time list of module i at time point η.
- **Transmission time list**. Transmission time list is used to represent the transmission process of a cross edge. It consists of the cross edge's release time, the start transmission time, the end transmission time and the channel index for transmission. Let $k_{(i,j)}$ indicate the index of channel for transmitting cross edge (i,j). We use the list $[t_r^{(i,j)}, t_s^{(i,j)}, t_f^{(i,j)}, k_{(i,j)}]$ to represent the transmission time list of cross edge (i,j).
- **Migration time list**. Migration time list is used to represent the migration process of a state. It contains the state's release time, the start migration time, the end migration time and the index of its occupied channel. Let k_{f_i} indicate the channel index for transmitting the state f_i. We use the list $[t_r^{f_i}, t_s^{f_i}, t_f^{f_i}, k_{f_i}]$ to represent the migration time list of state f_i.

The calculation process of make-span is shown in lines 4-19. The related scheduling schemes are shown as follows.

- **Modules execution scheduling**. We describe the scheduling of the module's execution by using the module's execution list $[t_r^i, \sigma_i, \lambda_i, x_i^{\eta}]$. The update process of t_r^i is shown in lines 6-11. The release time of the module i is

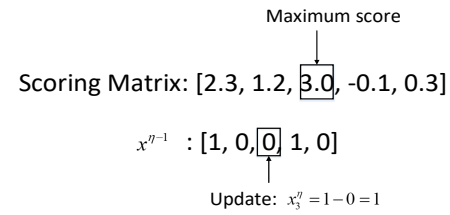


Fig. 4. The illustration of one shot problem

equal to the maximum value of the end transmission time $Max\{t_f^{(j,i)}, \forall(j,i) \in G_i\}$ of its predecessor data flows, the end execution time $Max\{\lambda_j, \forall j \in P_i\}$ of its predecessor modules and the end migration time t_f^i of its state f_i. If the module is not a stateful module, then its release time is equal to the maximum of the first two values. The update process of σ_i and λ_i are shown in lines 12-18.

• **Cross edge transmission scheduling**. We describe the scheduling of the transmission of cross edges by updating its data transmission list $[t_r^{(i,j)}, t_s^{(i,j)}, t_f^{(i,j)}, k_{(i,j)}]$. As shown in line 7, the $t_r^{(i,j)}$ is equal to the end execution time of module i, as shown in Equation(6)

$$t_r^{(i,j)} = \lambda_i. \tag{9}$$

Additionally, the $t_s^{(i,j)}$ and $t_f^{(i,j)}$ can be expressed as Equation(7) and Equation(8)

$$t_s^{(i,j)} = Max\{t_e^{k_{(i,j)}}, t_r^{(i,j)}\}, \tag{10}$$

$$t_f^{(i,j)} = t_s^{(i,j)} + D_{(i,j)}/B + \omega. \tag{11}$$

• **State migration scheduling process**. We describe the scheduling of state migration based on the state migration list $[t_r^{f_i}, t_s^{f_i}, t_f^{f_i}, k_{f_i}]$. As shown in line 9, the $t_r^{f_i}$ can be defined as Equation(9)

$$t_r^{f_i} = \eta. \tag{12}$$

If there exists an idle time interval in a channel between the state's release time and its relevant module's start execution time which is capable of migrating the state f_i, then the state will be migrated in this idle time interval in advance. Otherwise, the state will be migrated by use of FCFS policy and the $t_s^{f_i}$ and $t_f^{f_i}$ can be expressed as Equation(10) and Equation(11)

$$t_s^{f_i} = Max\{t_e^{k_{f_i}}, t_r^{f_i}\}, \tag{13}$$

$$t_f^{f_i} = t_s^{f_i} + f_i/B + \omega. \tag{14}$$

A simple example is proposed to describe the key steps of Algorithm 1. As shown in Fig. 4, in the first adjustment phase, we will adjust the execution position of module 3 with the biggest score of 3.0 in the scoring matrix, as shown in line 23. Then in the update phase, we will recompute the scoring matrix, as shown in line 24. The two phases are executed iteratively until there is no score greater than zero in the score matrix.

The time complexity of SM-H algorithm is $O(\lambda \times n^2)$, where n denotes the number of modules. Specifically, the time complexity required to compute the Score Matrix once is $O(n^2)$. Since the number of repeated adjustments of each module is set to no more than 3, then the number of updating rounds λ is within $O(n)$.

3.2 Benchmark Algorithms

In this section, we introduced classical baseline algorithms such as List Scheduling, Re-partitioning Algorithm, Genetic Algorithm, and Sequential Adjustment.

3.2.1 List Scheduling

List scheduling is a classical method for scheduling data stream applications on the constrained resources. It contains two steps: task selection and resource assignment. In the scheduling, we abstract the modules, data flows, and state as tasks. The mobile device, edge cloud server and N^{η} network channels are abstracted as $N^{\eta}+2$ machines. The tasks are selected with priorities based on their release time. The tasks are selected according to first-come-first-served policy. The algorithm selects the task with the earliest release time, and assign it to the machine which can complete the task at the earliest time. It is a shortsighted heuristic. For fairness in the comparison, we also use First-Come-First-served policy for $cross\,edge$ transmission and state migration as we do in SM-H algorithm.

3.2.2 Re-partitioning Algorithm

Re-partitioning algorithm does the computation partitioning under the current network bandwidth without considering the state migration overhead. The algorithm updates the module's execution position using the algorithm for partitioning stateless applications. Since the state migration overhead is ignored, it is more likely to cause network congestion and increase the make-span.

3.2.3 Genetic Algorithm

The genetic algorithm is a well-known evolutionary approach to solve the optimization problem. The key functions in the genetic algorithm implementation are to determine the chromosome representation and the corresponding fitness function to evaluate the performance of generated chromosomes. We encode the chromosome using the 0-1 module execution position variable $\overrightarrow{x}^t$ at time point t. Thus, the chromosome contains $1 \times n$ bits. Given the module execution position variable $\overrightarrow{x}^t$, the fitness value is represented as the reciprocal of make-span. The algorithm first initializes a population of a certain size. After several rounds of selection, crossover and mutation, the population converges to a minimum make-span as the final solution.

3.2.4 Sequential Adjustment

Sequential adjustment approach uses the same method as SM-H to calculate the adjustment score of each module. However, the difference is that the sequential adjustment approach update each module in a sequential order. The algorithm terminates until all the modules have been traversed.

3.3 Online Algorithm with Δt-Steps Look Ahead

The one shot SCPP updates the computation partitioning in the next time slot, aiming to alleviate the network congestion and minimize the make-span through selectively migrating state. Now we can predict the network bandwidth in the following Δt time slots by use of Network Status Prediction method [12] and aim to compute the computation partitioning over the Δt time slots. In this section, we design an online algorithm, namely Repeated Score Matrix-based Heuristic (RSM-H), for Δt steps look ahead.

Our goal is to alleviate the network congestion and minimize the average make-span of the following Δt time slots. We define the objective of this Δt-steps look ahead as the Equation(12)

$$Min \frac{1}{\Delta t} \sum_{t=\eta+1}^{\eta+\Delta t} (\sigma_{n+1}^{t} - \sigma_{0}^{t}). \tag{15}$$

The idea of the online algorithm is shown as follows. First, we determine the computation partitioning at each one of the future Δt time slots by using the SM-H algorithm. Then, we adjust the computation partitioning towards the Δt-steps look ahead. Specifically, we use the Execution Matrix (EM) to describe the execution position of each module from $\eta + 1$ to $\eta + \Delta t$. Given the computation partitioning x^{η} at time point η and the network bandwidth from $\eta + 1$ to $\eta + \Delta t$, we use the SM-H algorithm to orderly calculate the computation partitioning from $\eta + 1$ to $\eta + \Delta t$. Then we obtain the initial execution matrix. We still use the adjustment score to indicate the module's priority for adjustment. The adjustment score of module i at time t is expressed as follows. After adjusting the execution position of module i, we use the SM-H to orderly recompute the computation partitioning from time $t + 1$ to $\eta + \Delta t$. And then the adjustment score of module i is represented as the reduction of the average make-span from time η+1 to η+Δt. The Scoring Matrix represents the adjustment score for each module from $\eta + 1$ to $\eta + \Delta t$.

The RSM-H algorithm also consists of two phases: adjustment and update. In the first adjustment phase, we adjust the module with the greatest score in SM. When the adjusted module is at time $t \in [\eta + 1, \eta + \Delta t]$, we will use SM-H to adjust the computation partitioning from $t + 1$ to $\eta + \Delta t$ and update the execution matrix accordingly. In the update phase, we recompute the adjustment score of each module from $\eta + 1$ to $\eta + \Delta t$ and get the updated SM. The two phases are performed iteratively until there is no positive score in the SM. Then we get the final solution.

Algorithm 2 shows the pseudo-code of RSM-H. We first initialize a $n \times (\Delta t + 1)$ execution matrix EM as lines 1-4. Then we get the initial scoring matrix as shown in lines 5-11. The two key steps of the algorithm are performed next. Lines 14 and 15 indicate the adjustment phase, and line 16 indicates the update phase. These two phases are iteratively performed until there is no positive score in SM. Line 18 indicates that this algorithm ends and returns the final EM.

The time complexity of the algorithm is $O(|\Delta t|^2 \times \lambda_1 \times \lambda_2 \times n^3)$, where λ_1 and λ_2 denote the number of updating rounds of SM-H and RSM-H respectively, and n denotes the number of modules. Specifically, the time complexity of SM-H is $O(\lambda_1 \times n^2)$. And the time complexity for computing the Scoring Matrix once is $O(|\Delta t|^2 \times n)$. Through setting the number of repeated adjustment of each module to no more than 3, the number of updating rounds λ_2 is within $O(n)$.

Algorithm 2: Δt-Steps Look Ahead Algorithm RSM-H

Input : The computation partitioning x^{η} at time η; the network bandwidth during Δt time slots: $N^{\eta+1}, N^{\eta+2}, ..., N^{\eta+\Delta t}$
Output: the execution matrix EM

1 Initialize a $n \times (\Delta t + 1)$ execution matrix EM and set the 0-th column of EM as $EM[0] \leftarrow x^{\eta}$;
2 **for** *each time point* $t \in [\eta + 1, \eta + \Delta t]$ **do**
3 Calculate the computation partitioning x^t by use of SM-H;
4 Set the (t-η)-th column of EM as $EM[t - \eta] \leftarrow x^t$;
5 Initialize a $n \times \Delta t$ Scoring Matrix SM;
6 **for** *each time point* $t \in [\eta + 1, \eta + \Delta t]$ **do**
7 **for** *each module* $i \in V$ **do**
8 **if** *assume that updating the module i's execution position* **then**
9 Update the computation partitioning from $t + 1$ to $\eta + \Delta t$ with SM-H;
10 Compute the decrease on average make-span γ for Δt time slots;
11 Set $SM[i][t - \eta - 1] \leftarrow \gamma$;
12 Retrieve the greatest score β, the time point t, and the related index i of module;
13 **while** $\beta > 0$ **do**
14 Adjust the module i's execution location at time point t in EM, $EM[i][t - \eta] \leftarrow 1 - EM[i][t - \eta]$;
15 Adjust the computation partitioning from $t + 1$ to $\eta + \Delta t$ with SM-H and updating the EM accordingly;
16 Recompute the Scoring Matrix SM as lines 6-11;
17 Retrieve the greatest score β, the time point t, the index i of related module;
18 return the Execution Matrix EM;

4 Performance Evaluation

In this section, we will evaluate the performance of the proposed SM-H for one shot problem and the online solution for Δt-steps look ahead respectively. The performance metric we consider in the evaluation is the $make\text{-}span$.

4.1 Evaluations of the SM-H for the one shot problem

4.1.1 Architecture Design of the Simulator

The simulation was developed in one single machine configured as Intel(R) Core(TM) i5-2520M CPU. The simulation language was Java.

Fig.5 shows the architecture of the simulator, which includes a stateful data stream applications generator, an edge cloud simulator, a network resource monitor, a network status predictor, and a resources scheduler. The stateful data stream application is generated by the stateful data stream application generator firstly. Then the application is scheduled to execute in the edge cloud environment, which includes a mobile device, an edge cloud server and network bandwidth resources. Specifically, we abstract the total network bandwidth as a set of virtual network channels referring to the FDM protocol. We use the network

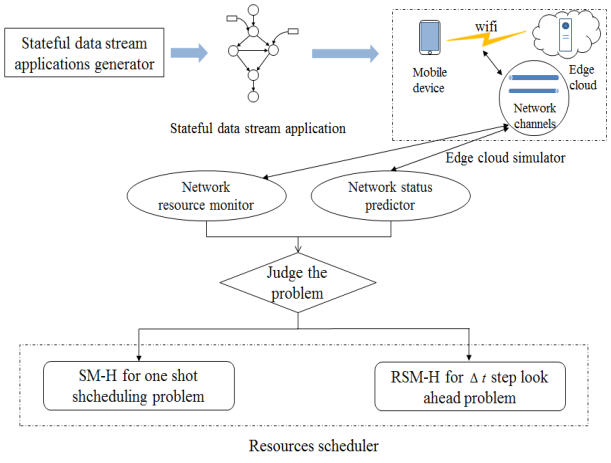


Fig. 5. Architecture of the simulator

TABLE 2
Parameters in each simulation

Parameters	Values
The number of modules n	40
The average indegree	3
The number of network channels N^t	3
The network bandwidth B	2MBps
The proportion of stateful modules	50%
The average execution time of modules on mobile device	2.4s
The average execution time of modules on edge cloud server	1.6s
The average data transmission size	2.66MB
The average size of state	5.32MB
CCR	1

resource monitor to get the usage of network resources, the degree of network congestion, and the amount of state migration. We use the network status predictor to predict the dynamically changing edge cloud network resources. If the future network bandwidth change significantly, or if the network resource monitor observes that there is a large amount of state migration at the current time, and the makespan is significantly increased caused by severe network congestion, the resources scheduler will be triggered.

Finally according to the different optimization goals of the problem, the SM-H algorithm and the RSM-H algorithm are utilized to solve the one shot scheduling problem and Δt-step look ahead problem. In addition, the resources scheduler is responsible for scheduling network bandwidth resources, mobile device and edge cloud resources at the same time. Although the stateful data stream applications we generate are not real data, they are similar to TPC-H queries and the stateful data flow processes of the STC algorithm.

Additionally, to the best of our knowledge, there is no approach/simulator/emulator for partitioning stateful data stream applications. We can not directly use it for simulating our approach. If we want to add new components for stateful tasks simulation, we need to modify source-code of the existing simulator which is not always supported. That is why we develop our own simulator. In our future work, we aim to abstract it into a general simulation framework for stateful task scheduling and delivery it as an open-source simulator.

4.1.2 Simulation Settings

Considering the number and size of applications in experiments are both possibly large, we implement a stateful data flow graph generator to simulate workloads of the stateful streaming applications. The generator can simulate applications similar to TPC-H queries [28], which are used in simulation experiments in the literature [13], and stateful data flow applications similar to the stateful data flow processes of the STC algorithm [5] in object tracking. These generated DAG-type stateful applications can cover all the configuration parameters described next. We use the level-by-level method to create the graph which was proposed by Tobita and Kasahara [6]. We could control the application graph through the following parameters: 1) the number of modules; 2) the average indegree of each module; 3) the proportion of stateful modules; and 4) communication-to-computation ratio (CCR), which is defined as the ratio of the average data state transmission time to the average computation time as shown in Equation (13). If an application graph's CCR is high, it can be considered as a communication-intensive application; otherwise, it is a computation-intensive application. If the proportion of stateful modules is bigger than zero, the application is considered as a stateful data stream application; otherwise, it is a stateless data stream application.

$$CCR = \frac{\frac{D_{(0,1)}+D_{(n,n+1)}+\sum_{(i,j)\in E} D_{(i,j)}}{(|E|+2)\times B} + \frac{\sum_{i\in V_{state}} f_i}{|V_{state}|\times B}}{2\times[\sum_{i\in V}(\frac{m_i+c_i}{2})/|V|]} \quad (16)$$

In order to make a trade-off between the convergence time and the experimental performance in the GA experiments. We set the population size as 50. The evolution generations are set to 150. The crossover probability and mutation probability are set to 0.8 and 0.15 respectively.

We have done a group of simulations to evaluate the effect of input parameters to the performance. In the simulation, we generate a stateful data stream application with 40 modules. We consider a mobile device and an edge cloud server. The average execution time of modules on the mobile device and on the edge cloud server is 2.4s and 1.6s respectively. The number of network channels is 3 and each channel has the bandwidth 2MBps. The proportion of the stateful modules is 50%. The CCR is set to 1. The amount of migration state in the application graph are randomly generated with the mean of 5.32MB. The amount of data transmitting on the edges are generated randomly with the mean of 2.66MB. Table 2 shows the default values of the environment parameters.

4.1.3 Evaluation Results

The primary performance metric we consider is the $makespan$ of the application, which is the optimization objective in our problem. Furthermore, we also concern on the $migration\text{-}state$, which indicates how much state migration arises from the update of partitioning. We compare SM-H

with four benchmark algorithms described in Section 3.2. The simulation results are shown as follows.

Fig.6(a) compares the make-span of SM-H with the benchmark algorithms under various application graph sizes n. As n increases, the other parameters are set as the default values in Table 2. The number of channels changes from 3 at t-1 to 2 at t. We define a terminology, named by make-span at time t-1, which indicates the make-span at time t if the computation partitioning is the same with that at time t-1. Under all the application graph sizes, our proposed SM-H has obviously better performance. As a benchmark naive approach, Re-partitioning has the worst performance. LS has slightly better performance than Re-partitioning due to it schedules the state migration. Compared to LS, Sequential Adjustment takes into account the impact of the module adjustment on the make-span, and thus has much better performance than LS. However, compared to the Sequential Adjustment, our proposed SM-H considers the adjustment priority of modules, therefore, SM-H outperforms Sequential Adjustment. GA can theoretically approximate optimum by adjusting parameters and increasing evolution generations. However, the computation complexity is very high. In order to make a trade-off between the computation complexity and the result, we set the population size as 50 and the evolution generations as 150.

Fig.6(b) shows how the migration state changes as the application size n increases. It is shown that Re-partitioning, LS, and GA bring more migration state than SM-H. This is because the Re-partitioning algorithm does not take into account the overhead of state migration, so it will bring more migration state when it is applied to the stateful applications. Reducing state migration can reduce the possibility of network congestion and thus reduce the make-span. Moreover, it can be seen from Fig.6(a) and Fig.6(b) that although the migration state affects the make-span, there is no strict correlation between them. In other words, the SM-H could minimize the make-span by selectively migrating state.

Next, we study how the performance changes as the ratio of the processing power of the mobile device to the edge cloud decreases. We set the average execution time of the modules on the mobile device as 2.4s, 3.2s, 4.8s, 6.4s, 9.6s, 12.8s while maintaining the average execution time of the modules on the edge cloud server as 1.6s. Fig.6(c) shows that SM-H always has better performance. When the ratio decreases, the differences among these algorithms get smaller. This is because when the difference between the processing power of the edge cloud and the mobile device is too large, these algorithms offload more modules to the edge cloud server similarly.

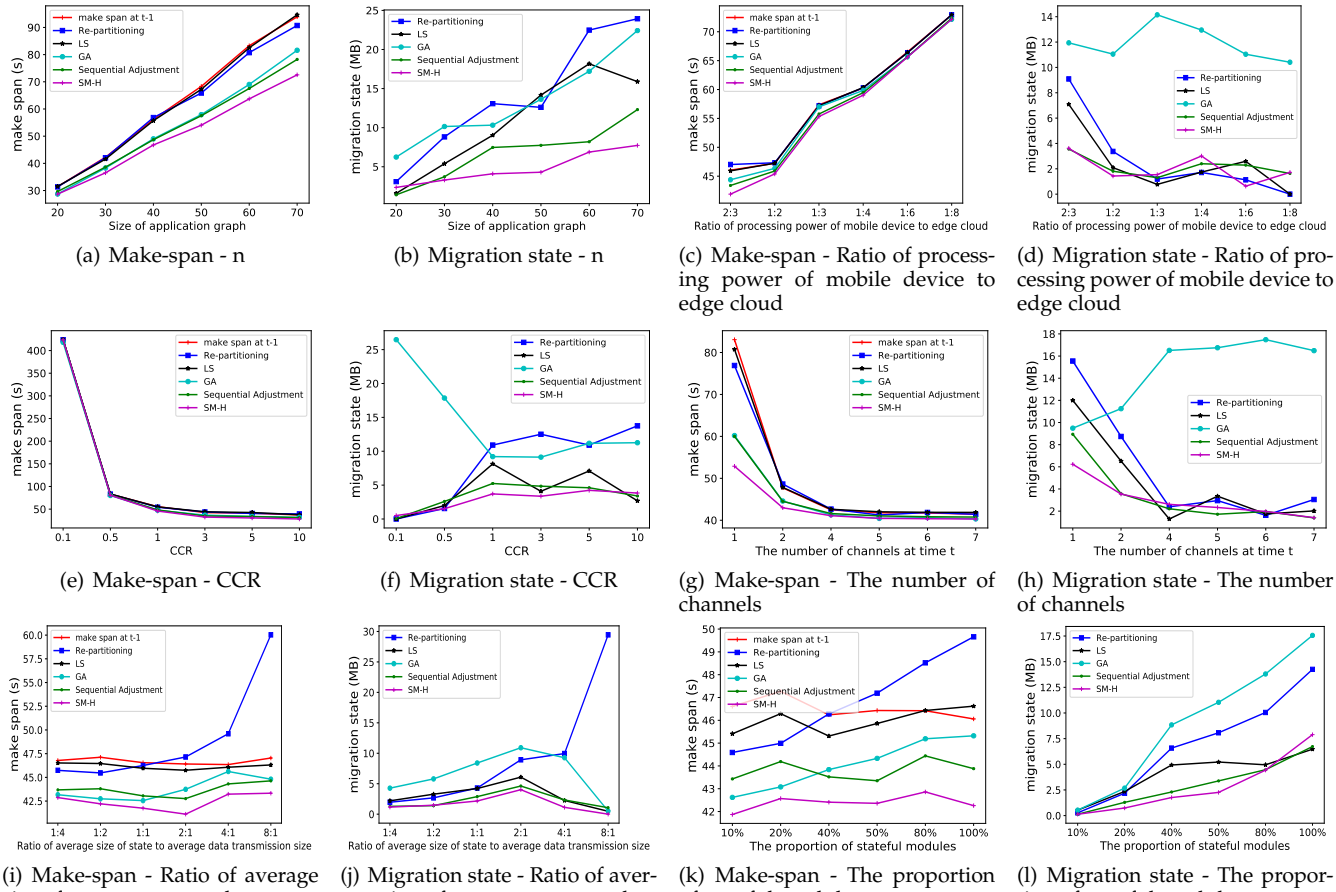
Fig.6(d) shows how the ratio affects the migration state. When the ratio is 2:3 or 1:2, SM-H migrates the least state. However, when the ratio is less than 1:2, SM-H migrates more state. This is because when the difference in the processing power between the mobile device and the edge cloud is small, the partitioning should be updated greatly as the network bandwidth changes. Meanwhile, SM-H can reduce the migration state compared to other approaches. However, when the difference of processing power is large, for example, when the ratio is 1:8, the migration state of the Re-partitioning and LS is 0. This is because they do not update the computation partitioning, while SM-H adjusts more stateful modules in order to minimize the make-span.

Fig.6(e) describes the effect of the CCR to the make-span. As shown in Equation (16), we set the average execution time of the modules as 20s, 4s, 2s, 0.67s, 0.4s, 0.2s, and the average transmission time is set to 2s. It is shown that the make-span decreases as the CCR becomes greater. When the CCR is small, SH-M has slightly better performance than the others, because the application is computation intensive, and most of the modules are executed on the edge cloud. When the CCR is large, the application is transmission intensive, and SM-H can get better results. Fig.6(f) shows the migration state changes as CCR increases. When the CCR is 0.1 or 0.5, the migration state of these methods is very small, since these partitioning methods execute most of the modules in the edge cloud. When the CCR is 1, 3, 5, the computation partitioning needs to be adjusted greatly, and the migration state of SM-H is the least among all the benchmark algorithms in most cases.

Now we study the effect of the number of network channels at time t to the performance. We set the number of channels at time t-1 to 3. The number of channels at time t is set to 1, 2, 4, 5, 6, 7 respectively. Fig.6(g) shows that the SM-H has much better performance when the number of channels is 1 or 2. However, when the number of channels gets large, most modules will be executed on the edge cloud and thus SM-H almost has the same make-span with the other algorithms. It is also shown in Fig.6(h) that the migration state of SM-H is not always the least, especially when the number of channels is greater than 2. This is because when the network has much more bandwidth, SM-H will minimize the make-span by adjusting a small number more of stateful modules.

Fig.6(i) shows the relationship between the make-span and the ratio of the average size of the state to the average data transmission size. We set the average data transmission size as 2.66MB. The average size of the state is set as 0.67MB, 1.33MB, 2.66MB, 5.32MB, 10.64MB, 21.28MB, respectively. We can see that the make-span of Re-partitioning continues to increase as the ratio gets greater because it does not consider the overhead of state migration. Nevertheless, the make-span of the other algorithms is not affected much by the ratio. Fig.6(j) shows how this ratio affects the migration state. When the ratio increases from 1:4 to 2:1, the migration state gradually increases. However, when the ratio increases from 2:1 to 8:1, the migration state decreases. It is because when the state is relatively small, we can reduce the make-span by adjusting some more stateful modules. On the other hand, stateful modules will not be adjusted when the state is too large.

Fig.6(k) and Fig.6(l) presents how the performance changes as the proportion of stateful modules increases. We set the proportion of stateful modules as 10%, 20%, 40%, 50%, 80%, and 100%, respectively. We can see in Fig. 6(k) that SM-H achieves the best performance in make-span. In fact, the changes in make-span of all algorithms are negligible except the Re-partitioning approach. In contrast, the volume of migration state increases significantly when the proportion increases as shown in Fig. 6(l).



(a) Make-span - n (b) Migration state - n (c) Make-span - Ratio of processing power of mobile device to edge cloud (d) Migration state - Ratio of processing power of mobile device to edge cloud

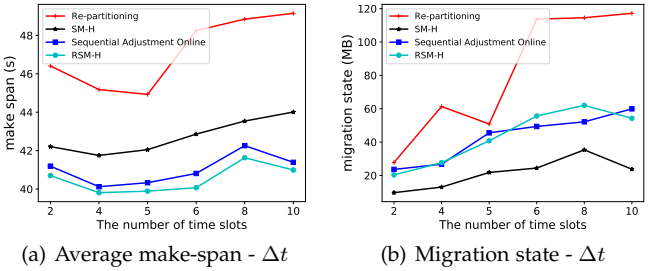
(e) Make-span - CCR (f) Migration state - CCR (g) Make-span - The number of channels (h) Migration state - The number of channels

(i) Make-span - Ratio of average size of state to average data transmission size (j) Migration state - Ratio of average size of state to average data transmission size (k) Make-span - The proportion of stateful modules (l) Migration state - The proportion of stateful modules

Fig. 6. Numerical evaluation results of the one-shot algorithms

TABLE 3
Parameters for Δ t-steps look ahead

Parameters	Values
The number of time slots Δt	10
The range of channels number at each time slot	$[1, 5]$
The number of modules n	40
The average indegree	3
The network bandwidth B	2MBps
The proportion of stateful modules	50%
CCR	1



(a) Average make-span - Δt (b) Migration state - Δt

Fig. 7. Impact of Δt on make-span and migration state

4.2 Evaluations of the online solutions

4.2.1 Environment Setting

We still use the same applications in previous simulations, in which the number of modules is n=40 and the average indegree is set to 3. We set the network bandwidth of each channel to 3MBps for simplicity. We set CCR to 1. The proportion of stateful modules is 50%. The number of time slots is set to 10 and the range of channel number randomly generated at each time slot is set to $[1, 5]$. Table 3 shows the default parameters setting for the Δt-steps look ahead.

We compare the proposed RSM-H with three benchmark methods: Re-partitioning, SM-H, and Sequential Adjustment Online (SAO). The Re-partitioning and SM-H benchmark algorithms are identical to their one shot problem version described in Section 3.2, while SAO is slightly revised from the Sequential Adjustment for the online scenario. Specifically, after calculating the initial solution by using SM-H, we orderly adjust each module at each time slot for just one round. The adjustment criterion is that whether the decrease on average make-span for Δt time slots is bigger than 0 after adjusting the execution position of the module. This method takes the global optimization for Δt time slots into consideration. It does not take into account the priority that the module is adjusted and the condition for ending the adjustment.

4.2.2 Evaluation Results

We simulate the proposed RSM-H and compare with the benchmark algorithms under various Δt , i.e., Δt=2, 4, 5, 6, 8, 10. Our purpose is to minimize the average completion time of future Δt time slots. Fig.7(a) describes the relationship between the average make-span and the number of time slots. Re-partitioning has the worst performance, since it does not take the state migration overhead into consideration. The network congestion leads to a tremendous increase in make-span. Because RSM-H and Sequential Adjustment Online are further optimized based on the initial solution SM-H, the make-span obtained by RSM-H and SAO are better than SM-H. Compared to SAO, RSM-H considers the adjustment priority of modules further, so it has better performance. Fig.7(b) shows how state migration changes when the number of time slots increases. The migration state of RSM-H and Sequential Adjustment is larger than SM-H. This is because RSM-H and SAO will further selectively adjust some stateful modules to minimize the average completion time. Overall, RSM-H outperforms the benchmark algorithms in terms of alleviating the network congestion and minimizing the make-span by selectively migrating state.

4.3 Discussion of the evaluations

To evaluate the proposed approach for a scheduling problem, normally in the experiment we need to emulate the workloads and the compute / network resources that captures the IoT features. These features include the network dynamic, frequent node failures and disconnections. So far, the model in our paper only considers the feature of network dynamic, i.e., the network bandwidth between the model devices and edge cloud can frequently change with time, while the node failures and network disconnection are not modeled in this paper. These features affect the reliability of the partitioning schemes. In our on-going work regarding this topic, we focus on the reliable computation partitioning for stateful streaming application. The solutions and results for solving the issue of node failures and disconnections as suggested by the reviewer will be reported in future.

The make-span of our proposed SM-H algorithm is reduced by 9.09% on average over Sequential Adjustment, and 9.10% on average over GA. The performance increase is measured by averaging the results in more than ten experiments, where the parameters in the environment settings such as the size of application graph, the state size, CCR and etc differ in each experiment. As shown in Fig.6, under particular settings, i.e., the application size is great and the state size is large, SM-H can reduce the make-span by more than 25% compared with SA. Besides the performance increase in terms of make-span, another benefits of SH-M is that it can reduce the state transmission cost than the benchmark methods, which is meaningful and beneficial in some cases where the network resources of the connections between the end device and edge server is very limited.

The make-span of the RSMH is reduced by 1.23% on average compared to Sequential Adjustment Online (SAO), and by 6.54% on average with respect to SM-H. The reason why the performance of RSM-H and SAO is similar is that both of them use the method of score-based module scheduling, which is first proposed in our paper. Compared with SAO, RSM-H is further optimized by considering the adjustment priority of the modules. However, the time complexity increases as the number of iterations increases. In our on-going work, we attempt to leverage the GNN and reinforcement learning to predict the adjustment score of the module, aiming to improve the performance of RSM-H. This results will be reported in our future work. Nevertheless, the value of RSM-H is that it still provides a novel solution framework. Through increasing the algorithm iteration and meanwhile reducing the complexity, solutions based RSM-H framework can further greatly optimize the performance.

5 Related Works

This paper is most related to the computation partitioning and task scheduling in dynamic edge cloud environments. Then we introduce the related works as follows.

Computation partitioning is the widely used technique to solve the resource poverty problem of mobile devices in edge cloud environments. Some earliest works consider the single user computation partitioning problem, which is to decide for a single user which parts of an application should be executed locally and which parts are executed remotely. Rich Wolski [14] proposed a scheduler for making computation offloading decisions in computational grid settings and formulated the scheduling problem as a statistical decision problem that can either be treated "classically" or using a Bayesian approach. Cai Wei et al [16] presented a decomposed cloud gaming platform which supports flexible migrations of gaming components between the cloud server and the players' terminals. Chen Xu et al [17] proposed a comprehensive framework aiming at provisioning flexible on-demand mobile-edge cloud service. These works [20] [21] support the application partitioning in the dynamic network environment. Yang et al [4] proposed a framework for partitioning and execution of data stream applications in mobile cloud computing, which is the first work to propose a framework to provide runtime support for the dynamic computation partitioning and execution of data stream applications. When jointly considering the computation partitioning and the workload scheduling within the cloud, the multi-user computation partitioning model has been studied recently [15] [18]. The problem is to solve the issue arising from the competition among multiple users for the resources in the cloud. Chen et al [22] studied a multi-user computation offloading game in case that the cloud has limited resources.

The related works above consider the partitioning for stateless data stream applications. Our paper is novel in the study of the problem of partitioning the *stateful* data stream applications.

Previous works that involves state migration are also important references to our work. First of all, the state is also mentioned in MapReduce [29]. R. Castro Fernandez [23] discussed the challenges encountered by the SPS system in scaling out on demand and fault tolerance. And the checkpoint method is used to realize state backup and migration. Moreover, the stateful operator is join or aggregate in MapReduce. M. Shah [24] proposed Flux to enable state

partitioning and dataflow routing in MapReduce while state migration overhead is not considered. Y. Zhu [25] studied the dynamic plan migration problem for continuous stateful queries in MapReduce. And two online plan migration strategies are proposed to address the problem of migrating query plans that contain stateful operators, such as joins. However, it is different from the computation partitioning problem in dynamic edge cloud environments. N R. Katsipoulakis [26] presented a new model for stream partitioning which considers tuple imbalance and aggregation cost for *stateful* operations. Nevertheless, that work doesn't take the online re-configuration and state migration overhead into account.

Another related area is the parallel and distributed computing. List-based task scheduling heuristic for data stream applications has been proposed in [30] [31] with high performance and low complexity. Unlike list-based scheduling heuristics, clustering-based scheduling heuristics could be applied in heterogeneous systems [32] [33]. The relevant work to our proposed approach is the Data Ready Time (DRT) based method [34], where DRT means the latest arrival time of all its precedent data flows. However, these classic scheduling heuristics do not consider the partitioning for stateful applications in dynamic edge environments. In contrast, our proposed solutions aim to minimize the make-span with considering the state migration overhead when edge network environments change.

6 Conclusion

In this paper, we study the computation partitioning problem of stateful data stream applications in dynamic edge cloud environments. We developed an efficient heuristic, named by Score Matrix-based Heuristic (SM-H), to solve the problem. The algorithm takes the reduction of make-span after updating the module's execution position as its score. It iteratively adjusts the module with the greatest score until there is no score bigger than zero. On basis of SM-H, we further develop the Repeated Score Matrix-based Heuristic (RSM-H) to solve Δt steps look ahead problem. Through extensive simulations, we conclude that both SM-H and RSM-H have better performance than benchmark methods in terms of make-span through selectively migrating state. Academically the models and solutions proposed in this paper enrich the scheduling theory and methodologies of the stateful application tasks. In practice, they can be applied in edge computing for optimizing the performance of stateful data stream applications such as augmented reality, object tracking and so on.

Considering that our research work is an NP-hard problem, normally it can not be solved with an optimal solution in polynomial time. We present the performance results of our heuristic solutions by extensive evaluation. Since the main purpose of our work is to introduce the computation partitioning problem in dynamic edge cloud environments, we have simplified the edge cloud and network model with only reserving the basic and important properties. We not only simulate the dynamic edge cloud environments and generate stateful data flow applications that cover all required parameters, but also conduct a lot of experiments to verify sufficient test cases. So our simulations are also sufficient and valuable. We did not develop the testbed of edge cloud with the mobile devices, edge servers and varying networks. In our future work, we will test the performance of our proposed method on top of the testbed.

Acknowledgment

The research is supported by the National Natural Science Foundation of China under Grant 61972161, Hong Kong RGC under Grant PolyU 15217919, Key Research and Development Program of Guangdong Province under Grant 2019B010154004, Guangdong Basic and Applied Basic Research Foundation under Grant 2020A1515011496, and the Fundamental Research Funds for the Central Universities under Grant 2018MS53.

References

[1] Y. Wang, M. Sheng, et.al. Mobile-Edge Computing: Partial Computation Offloading Using Dynamic Voltage Scaling. *IEEE Transactions on Communications*, vol.64, no.10, pp.4268-4281, 2016.

[2] S. Kosta, A. Aucinas, et.al. Thinkair: Dynamic Resource Allocation and Parallel Execution in Cloud for Mobile Code Offloading. In *Proc. of INFOCOM*, pp.945-953, 2012.

[3] R. Balan, M. Satyanarayanan, et.al. Tactics Based Remote Execution for Mobile Computing. In *Proc. of MobiSys*, pp.945-953, 2003.

[4] L. Yang, J. Cao, et.al. A Framework for Partitioning and Execution of Data Stream Applications in Mobile Cloud Computing. In *ACM SigMetrics Performance Evaluation Review*, vol. 40, no. 4, pp.23-32, 2013.

[5] Z. Li, Y. Su, W. Ma. An Improved Spario-temporal Context Tracking Algorithm Combining LK Optical Flow. In *2017 10th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, 2017.

[6] T. Tobita, H. Kasahara. A Standard Task Graph Set for Fair Evaluation of Multiprocessor Scheduling Algorithms. *Journal of Scheduling*, 5(5):379-394, 2002.

[7] R. Balan, J. Flinn, et.al. 2002. The Case for Cyber Foraging. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop (EW 10)*. ACM, New York, NY, USA, 87-92.

[8] K. Habak, M. Ammar, et.al. 2015. Femto Clouds: Leveraging Mobile Devices to Provide Cloud Service at the Edge. In *2015 IEEE 8th International Conference on Cloud Computing*. 9-16.

[9] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. 2009. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing* 8, 4(Oct 2009), 14-23.

[10] E. Saurez, K. Hong, D. Lillethun, et.al. 2016. Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS '16)*. ACM, New York, NY, USA, 258-269.

[11] W. Shi, J. Cao, et.al. 2016. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal* 3, 5 (Oct 2016), 637-646.

[12] L. Yang, J. Cao, et.al. Run Time Application Repartitioning in dynamic mobile cloud environments. *IEEE Transactions on Cloud Computing*, vol. 4, no. 3, pp. 336-348, Jul./Sep. 2016.

[13] H. Mao, M. Schwarzkopf, et.al. Learning Scheduling Algorithms for Data Processing Clusters. *Proceedings of the 2019 ACM conference on SIGCOMM*, August 19-23, 2019, Beijing, China.

[14] R. Wolski, et.al. Using Bandwidth Data to Make Computation Offloading Decisions. In *IPDPS'08*.

[15] S. Guo, B. Xiao, et.al. Energy-Efficient Dynamic Offloading and Resource Scheduling in Mobile Cloud Computing. In *Proc. of INFOCOM*, pp.1-9, 2016.

[16] W. Cai, H. C. B. Chan, X. Wang and V. C. M. Leung. Cognitive Resource Optimization for the Decomposed Cloud Gaming Platform. in *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 12, pp. 2038-2051, Dec. 2015.

[17] X. Chen,W. Li, et.al. Efficient Resource Allocation for On-Demand Mobile-Edge Cloud Computing. *IEEE Transactions on Vehicular Technology*, Vol. 67, No. 9, pp. 8769-8780, Sept. 2018.

[18] L. Yang, J. Cao, et.al. Multi-User Computation Partitioning for Latency Sensitive Applications in Mobile Cloud Computation. *IEEE Transactions on Parallel and Distributed Systems*, vol.28, no. 5, pp. 1440-1452, 2016.

[19] E. Cuervoy, A. Balasubramanianz, D. Cho. MAUI: Making Smartphones Last Longer with Code Offload. In *Proc. MobiSys's 10*. ACM press, 2010.

[20] B. Chun, S. Ihm, P. Maniatis, M. Naik, A. Patti. CloneCloud: Elastic Execution between Mobile Device and Cloud. In *Proc. EuroSys' 10*.

[21] W. Cai, Y. Chi, C. Zhou, C. Zhu and V. C. M. Leung. UBCGaming: Ubiquitous Cloud Gaming System. in *IEEE Systems Journal*, vol. 12, no. 3, pp. 2483-2494, Sept. 2018.

[22] X. Chen, L. Jiao, et.al. Efficient Multi-User Computation Offloading for Mobile-Edge Cloud Computing. *IEEE/ACM Transactions on Networking*, vol. 24, no. 5, pp. 2795-2808, 2016.

[23] R. Castro Fernandez, M. Migliavacca, et.al. Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management. In *SIGMOD*, pages 725-736, 2013.

[24] M. Shah, M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, pages 25-36, 2003.

[25] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries over Data Streams. In *SIGMOD*, pages 431-442, 2004.

[26] N. R. Katsipoulakis, A. Labrinidis, et.al. A Holistic View of Stream Partitioning Costs. In *Proceedings of the VLDB Endowment*, v. 10 n.11, p.1286-1297, August 2017.

[27] M. Elseidy, A. Elguindy, V. A., and C. Koch. Scalable and Adaptive Online Joins. In *PVLDB*, pages 441-452, 2014.

[28] TPC-H 2018. The TPC-H Benchmarks. www.tpc.org/tpch/. (2018)

[29] A. Brito, A. Martin, T. Knauth, S. Creutz, et.al. Scalable and Low-Latency Data Processing with Stream MapReduce. In *Proc. of CloudCom*, 2011.

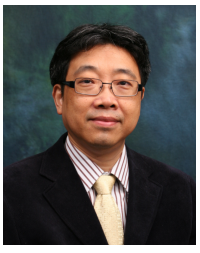
[30] H. Arabnejad and J. G. Barbosa. List Scheduling Algorithm for Heterogeneous Systems by An Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no.3, pp.682-694, Mar.2014.

[31] H. Topcuoglu, et.al. Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no.3, pp. 260-274, Mar. 2014.

[32] B. Jedari, M. Dehghan. Efficient DAG Scheduling with Resource-Aware Clustering for Heterogeneous Systems. *Comput. Inf. Sci.*, vol. 208, pp. 249-261, 2009.

[33] S. Chingchit, et.al. A Flexible Clustering and Scheduling Scheme for Efficient Parallel Computation. In *Proc. 13th Int. 10th Symp. Parallel Distrib. Process.*, pp. 500-505, 1999.

[34] O. Sinnen. Task Scheduling for Parallel Systems. Hoboken, NJ, USA: Wiley, 2007.



Jiannong Cao received the B.Sc. degree in computer science from Nanjing University, China, in 1982, and the M.Sc. and Ph.D. degrees in computer science from Washington State University, USA, in 1986 and 1990 respectively. He is currently a Chair Professor and the Head of Department of Computing at The Hong Kong Polytechnic University, Hong Kong. His research interests include parallel and distributed computing, wireless networks and mobile computing, big data and cloud computing, pervasive computing, and fault tolerant computing. He has co-authored 5 books in Mobile Computing and Wireless Sensor Networks, co-edited 9 books, and published over 500 papers in major international journals and conference proceedings. He is a fellow of IEEE.

Wei Cai [S'12-M'16] received the B.Eng. degree in Software Engineering from Xiamen University, China in 2008, the M.S. degree in Electrical Engineering and Computer science from Seoul National University, Korea, in 2011, and the Ph.D. degree in Electrical and Computer Engineering from The University of British Columbia (UBC), Vancouver, Canada, in 2016. From 2016 to 2018, he was a Postdoctoral Research Fellow with UBC. He joined the School of Science and Engineering, The Chinese University of Hong Kong, Shenzhen, in August 2018, where he is currently an Assistant Professor. He has completed visiting research at National Institute of Informatics, Japan, The Hong Kong Polytechnic University, and Academia Sinica, Taiwan. His recent research interests include software systems, cloud and edge computing, blockchain systems, and video games.

Mingkui Tan received his Bachelor Degree in Environmental Science and Engineering in 2006 and Master degree in Control Science and Engineering in 2009, both from Hunan University in Changsha, China. He received the PhD degree in Computer Science from Nanyang Technological University, Singapore, in 2014. From 2014-2016, he worked as a Senior Research Associate on computer vision in the School of Computer Science, University of Adelaide, Australia. Since 2016, he has been with the School of Software Engineering, South China University of Technology, China, where he is currently a Professor. His research interests include machine learning, sparse analysis, deep learning and large-scale optimization.

Shaoshuai Ding received the BSc degree from Jiangnan University, in 2017. He is now a second-year student for MSe at the School of Software Engineering, South China University of Technology (SCUT), China. His research interests include edge computing, cloud computing, and Internet of things.

Zhengyu Wang received the BSc degree from Xiamen University, China, in 1987, and the MSc and the PhD degrees from Harbin Institute of Technology, China, in 1990 and 1993, all in computer science. He is currently a professor and the dean of the School of Software Engineering, South China University of Technology, China. His research interests include distributed computing and SOA, operating systems, software engineering, and large-scale application design and development.

Lei Yang received the BSc degree from Wuhan University, in 2007, the MSc degree from the Institute of Computing Technology, Chinese Academy of Sciences, in 2010, and the PhD degree from the Department of Computing, The Hong Kong Polytechnic University, in 2014, where he worked as a postdoctoral fellow for more than a year. Since 2016, he has been working at the School of Software Engineering, South China University of Technology, China, as an associate professor. His research interest includes mobile cloud computing, Internet of things, and big data analytic.