DeepWeave: Accelerating Job Completion Time with Deep Reinforcement Learning-based Coflow Scheduling

Penghao Sun[1], **Zehua Guo**[2*], **Junchao Wang**[1], **Junfei Li**[1], **Julong Lan**[1] and **Yuxiang Hu**[1]
[1]National Digital Switching System Engineering & Technological R&D Center
[2]Beijing Institute of Technology
sphshine@126.com, guolizihao@hotmail.com, wangjunchao11@126.com, lijunfei90@qq.com, {ndscljl, huyx}@126.com

Abstract

To improve the processing efficiency of jobs in distributed computing, the concept of coflow is proposed. A coflow is a collection of flows that are semantically correlated in a multi-stage computation task. A job consists of multiple coflows and can be usually formulated as a Directed-Acyclic Graph (DAG). A proper scheduling of coflows can significantly reduce the completion time of jobs in distributed computing. However, this scheduling problem is proved to be NP-hard. Different from existing schemes that use hand-crafted heuristic algorithms to solve this problem, in this paper, we propose a Deep Reinforcement Learning (DRL) framework named DeepWeave to generate coflow scheduling policies. To improve the inter-coflow scheduling ability in the job DAG, DeepWeave employs a Graph Neural Network (GNN) to process the DAG information. DeepWeave learns from the history workload trace to train the neural networks of the DRL agent and encodes the scheduling policy in the neural networks, which make coflow scheduling decisions without expert knowledge or a pre-assumed model. The proposed scheme is evaluated with a simulator using real-life traces. Simulation results show that DeepWeave completes jobs at least 1.7× faster than the state-of-the-art solutions.

1 Introduction

As the demand of data processing increases, cluster computing-based computing (e.g., big data and cloud computing [Xu and Tang, 2015]) has been widely adopted in recent years since it can achieve a high parallel computation performance under a relatively low Capital Expenditure (CAPEX). Many cluster computing applications (e.g., Spark and MapReduce) are deployed in data centers and abstracted with the job model, which is composed of several successive computation stages and communication stages among the computation stages. The execution of each computation stage can only start when the intermediate data from the previous stage are completely transmitted. This working procedure can be modeled as a job Directed-Acyclic Graph (DAG), where computation stages and communication stages are modeled as nodes and edges, respectively. In a job DAG, one communication stage involves a set of flows, which interweave with each other from two groups of machines and share a common job goal. A set of such flows is called a *coflow* [Chowdhury and Stoica, 2012]. Existing works [Chowdhury *et al.*, 2011] point out that in Data Center Networks (DCNs), the coflow transmission (i.e., the sequence-dependent transmission of intermediate data from one computation element to another) consumes at least 50% processing time of a whole job. Thus, a good coflow scheduling could significantly reduce the Job Completion Time (JCT) [Wang *et al.*, 2018b].

However, designing a good coflow scheduling scheme is non-trivial. Efficiently scheduling coflows in DCNs is challenging because various coflow characteristics should be considered, including properties of a coflow (e.g., flow size and the number of parallel flows) and among coflows (e.g., the relationship of different coflows in a job DAG). An optimal coflow scheduling has been proved to be NP-hard [Chowdhury *et al.*, 2014; Wang *et al.*, 2019; Qiu *et al.*, 2015]. Existing works simplify the problem and use heuristic solutions to minimize the transmission time of coflows [Chowdhury *et al.*, 2014; Zhao *et al.*, 2015; Chen *et al.*, 2016; Li *et al.*, 2016; Chowdhury and Stoica, 2015; Dogar *et al.*, 2014; Huang *et al.*, 2015; Zhang *et al.*, 2016; Gao *et al.*, 2016; Wang *et al.*, 2019; 2018a]. However, these works have the following two limitations:

1. Most existing works only concentrate on minimizing the transmission time of coflows in a single communication stage and do not provide a deep dive into the job-specific communication requirement [Chowdhury *et al.*, 2014; Chen *et al.*, 2016; Li *et al.*, 2016; Zhang *et al.*, 2016; Gao *et al.*, 2016; Wang *et al.*, 2018a; 2019; Tan *et al.*, 2019]. Thus, the execution sequence of coflows in different communication stages of the job DAG may not be considered, and these job-agnostic scheduling methods cannot bring an optimal JCT.

2. Hand-crafted heuristic solutions simplify the problem with some relaxations and only ensure an rough approximation to the optimal solution of the NP-hard problem [Chowdhury *et al.*, 2014; Zhao *et al.*, 2015; Chen *et al.*, 2016; Li *et al.*, 2016; Chowdhury and

*Corresponding author

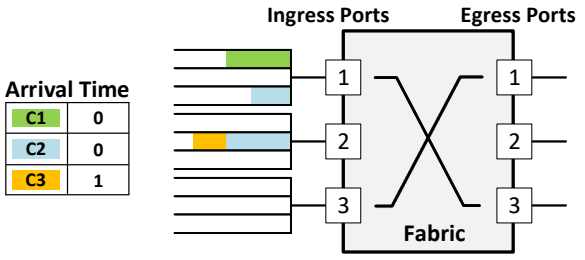

Figure 1: An example of coflow.

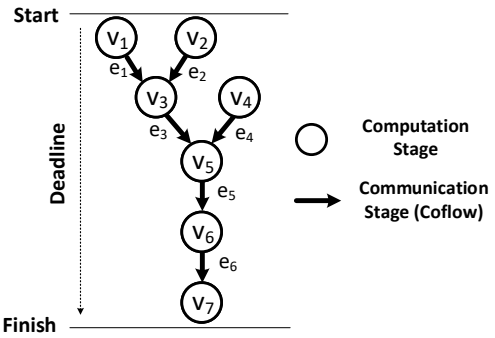

Figure 2: An example of job DAG.

Stoica, 2015; Dogar *et al.*, 2014; Huang *et al.*, 2015; Zhang *et al.*, 2016; Gao *et al.*, 2016; Wang *et al.*, 2019; 2018a] . For example, the non-preemptive scheduling algorithm [Wang *et al.*, 2019] only ensures a 2-approximate optimal solution and leaves a large gap to the optimal solution.

In this paper, we propose DeepWeave, a learning-based method to efficiently schedule coflows. DeepWeave uses the framework of Deep Reinforcement Learning (DRL) to learn from the history trace and train the neural networks. DeepWeave consists of two cascaded sets of neural networks and a policy converter module, and it encodes the scheduling policy in the neural networks, which can make coflow scheduling decisions without expert knowledge and a pre-assumed model. DeepWeave works at both the intra-coflow scheduling and inter-coflow scheduling. Specifically, to better process the job DAG information and ensure the generalization ability of the neural networks under unpredictable input DAGs, we propose to use Graph Neural Network (GNN) for processing input data. DRL agent uses the policy in the trained neural networks to generate a priority scheduling list for different coflows. Based on the list, flows in each coflow is further scheduled in a fine-grained fashion. Simulation results show that DeepWeave completes jobs at least 1.7× faster than the state-of-the-art solutions.

The contributions in this paper can be summarized as follows:

1. Our paper is the first work that proposes a machine learning-based coflow scheduling architecture to accelerate JCT.
2. We use a DRL as the learning-based framework to train the neural networks and explore the coflow scheduling policy without human experience.
3. We design a GNN-based neural network structure to process input job DAGs. The structure generalizes well to different DAG sizes and shapes after training.
4. We evaluate DeepWeave's performance using real-life traces and generated traces. Simulation results show DeepWeave outperforms state-of-the-art solutions in terms of JCT.

The rest of this paper is organized as follows: Section 2 formulates the problem of this paper. Section 3 details the proposed DeepWeave. Section 4 evaluates the performance of DeepWeave. Section 5 concludes this paper.

2 Problem Formulation

Coflows usually exist in the multi-stage computation task in DCNs. In coflow scheduling, a DCN can be logically viewed as a giant non-blocking switch fabric [Chowdhury *et al.*, 2014; Chowdhury and Stoica, 2015] that connects different computation machines. Each port of the fabric has the same capacity, and congestion only occurs at ports since flows in the same port compete for bandwidth. Following existing works, we use this DCN model for our study. Figure 1 is a typical example of coflow [Chowdhury and Stoica, 2015]. The DCN is modeled as a non-blocking switch fabric and interconnects different groups of machines. In this figure, the fabric has three input ports, and each input port has three queues. There are three coflows: C1, C2, and C3. At time slot 0, C1 arrives at port 1, and C2 arrives at ports 1 and 2. C1 and C2 contend for port 1. At time slot 1, C3 arrives at port 2.

We model a job in DCN as a job DAG $G = (V, E, T)$, where $V = \{v_1, v_2, ...v_{|V|}\}$ is the set of nodes, which denotes a computation stage of the job, $E = \{e_1, e_2, ...e_{|E|}\}$ is the set of edges, which denotes a communication stage[1] of the job, and T denotes a global deadline of the job. Each communication stage is a coflow between two computation stages. A job DAG is processed in pipeline following the dependencies among its different stages, and its processing has to be completed within deadline time T. Figure 2 shows an example of job DAG including seven nodes and six edges. There are two types of coflow dependency relationships in a job DAG [Chowdhury and Stoica, 2015]: Starts-After and Finishes-Before. In a Starts-After relationship, e_a cannot start until e_b finishes (denoted as $e_a \mapsto e_b$); in a Finishes-Before relationship, e_a cannot finish until e_b finishes (denoted as $e_a \rightarrow e_b$).

Within a job DAG, computation stages are usually executed in parallel, and thus coflows are generated in parallel from different computation stages and compete with each other at the ports of the fabric. The computation time of the computation stages can be regarded as constant, and the scheduling of coflows is the main factor of the overall JCT.

Consider that in a job DAG, there are N coflows $\{e_1, e_2, ...e_{|N|}\}$. The switch fabric has M ingress ports and

[1]In this paper, we use communication stages and coflows interchanganbly.

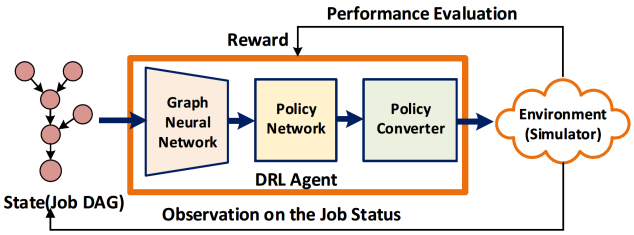

Figure 3: The DRL framework used in DeepWeave.

M egress ports, and a coflow can be denoted as a group of flows: $e_i = \{f_i^{m,n} | 1 \leq m \leq M, 1 \leq n \leq M\}$, where $f_i^{m,n}$ denotes a flow from port m to port n in coflow e_i. $fs_i^{m,n}$ denotes the size of flow $f_i^{m,n}$ normalized to the capacity of the port. If there are no flows in coflow e_i from port m to port n, $fs_i^{m,n} = 0$. Since each port has the same capacity, the transmission time of a flow is proportional to the flow size at each port. Without loss of generality, we use the size of each flow to denote the transmission time in the following. We also use C_i to denote the transmission time of coflow e_i in the job DAG, and $C_i = End(e_i) - Start(e_i)$, where $End(e_i)$ and $Start(e_i)$ denote the end time and start time of e_i, respectively. Under these definitions, the aim of coflow scheduling is to minimize the average transmission time of the whole job (i.e., $\sum_i^N C_i$) under dependency constraints (i.e., $Start(e_i) \geq End(e_x | e_x \mapsto e_i)$ and $End(e_i) \leq End(e_x | e_x \rightarrow e_i)$).

3 Design of DeepWeave

In this section, we introduce the design of DeepWeave. First, we overview the architecture of DeepWeave. Second, we explain DeepWeave's DRL framework in the training. Third, we introduce the processing mechanism of the neural networks in DeepWeave. Finally, we introduce Policy Converter that translates the output values of the DRL agent into concrete coflow scheduling policies.

3.1 Overview of DeepWeave

DeepWeave is designed based on the framework of DRL to generate coflow scheduling policies for the network. Figure 3 shows a coflow scheduling agent, which is composed of a group of neural networks (detailed in Sections 3.2 and 3.3) and a policy converter (detailed in Section 3.4). During the working process, DeepWeave collects the job DAG information and converts the information to a feature map as the input of GNN, which calculates the features of the job DAG and passes the calculated information to Policy Network. Then, Policy Network generates a priority list as the output, which is used as an input in Policy Converter. Finally, Policy Converter translates the priority list into coflow scheduling actions, which are used to schedule coflows and flows in the coflows.

3.2 DRL Framework for Training

DeepWeave trains its neural networks through the framework of DRL and uses a tuple that contains three elements $\langle S, A, R \rangle$ to describe the interaction. In the tuple, S is the state space, which stands for the status observation of the environment (i.e., job DAG in DeepWeave); A is the action space, which stands for the space of the output actions of the RL agents; R is the reward space, which evaluates the quality of the action. The DRL agent of DeepWeave gets trained through the interaction with the network environment step by step. At each step, the agent collects an observation $s \in S$ from the environment and generates an action $a \in A$ to schedule coflows in the environment through a certain calculation process, which can be abstracted as a policy. The quality of a certain schedule action $a \in A$ is evaluated by an evaluation function with a reward $r \in R$ to describe the quality. Based on the reward $r \in R$, the policy is polished towards a better performance. Specifically, the interfaces in DeepWeave are defined as follows:

- **State**: flows in each coflow and the shape of a job DAG.
- **Action**: priority list that denotes the scheduling priority of edges in a job DAG.
- **Reward**: the completion time of a job DAG.

The training process of the DRL agent is carried out in episodes, and each episode consists of multiple interaction steps. In each episode, the target of the training is to maximize the cumulated reward $R_{epi} = \sum_{t=0}^{T} r(t)$, where T denotes the total number of interaction steps in this episode. We use μ to represent the policy of the DRL agent. At each step t, the agent gets an observation s_t from the environment and generates action a_t based on $a_t = \mu(s_t)$. As mentioned above, in DRL, the policy is generated by neural networks. Therefore, $a_t = \mu(s_t)$ can be more accurately represented as $a_t = \mu_\theta(s_t)$, where θ denotes the parameters (e.g., the weights and biases among different neurons) of the neural networks. When an episode completes, the cumulated reward R_{epi} can be used to evaluate the policy in this episode.

In DeepWeave, we use a policy gradient method to update the neural networks based on R_{epi} towards a better policy. In the policy gradient method, a gradient descent is calculated to update the neural network parameters as follows [Sutton *et al.*, 2000]:

$$\theta' = \theta + \alpha \sum_{t=1}^{T} \nabla_\theta \log \mu_\theta(s_t, a_t) Q_t,$$

where α denotes the learning rate that controls the update speed of θ during each episode, and Q_t denotes a quality evaluation of the policy in the current episode. Specifically, we make $Q_t = \sum_{t'=t}^{T} r_{t'} - b_t$ as a biased form, where b_t is used as a baseline value to limit the variance of the policy gradient [Greensmith *et al.*, 2004]. In this way, the parameters of the neural networks get updated after each episode, so the expected performance of this DRL agent in the next episode should be better. Since in several initial episodes, the neural network parameters are close to random values, and the policy is not good, we can use a method similar to the one used in Decima [Mao *et al.*, 2019] to accelerate the training by shortening the episode length.

Algorithm 1 Training process of DeepWeave

Input: job DAG, job completion time;
Output: coflow scheduling priority;
1: Initialize all neural networks;
2: **for** iteration = 1 to MAX_NUM **do**
3: Episode length $l = l_{init}$;
4: Run agent $i = 1$ to N and get: $(s_1^i, a_1^i, r_1^i, ..., s_l^i, a_l^i, r_l^i) \sim \mu_\theta$;
5: $\Delta\theta = 0$;
6: **for** $t = 1$ to l **do**
7: **for** agent $i = 1$ to N **do**
8: $R_t^i = \sum_{t'=t}^{l} r_{t'}^i$;
9: **end for**
10: Compute baseline value $b_t = \frac{1}{N}\sum_{j=1}^{N} R_t^j$;
11: **for** agent $i = 1$ to N **do**
12: $\Delta\theta = \Delta\theta + \nabla_\theta \log \mu_\theta(s_t^i, a_t^i)(R_t^i - b_t)$;
13: **end for**
14: **end for**
15: $l = l + \epsilon$;
16: $\theta = \theta + \alpha\Delta\theta$;
17: **end for**

The logic of the whole training process is shown in Algorithm 1. To accelerate the training process, we implement N parallel agents. The training objective of Algorithm 1 is to let the parameters of the neural networks evolve so as to generate a good coflow scheduling policy. Lines 3-4 run a series of experiment episodes for the DRL agent, and line 5 resets the updated value of the neural networks to zero in each iteration. Lines 6-14 calculate the updated value for the neural networks based on the cumulated reward, among which lines 7-10 calculate the baseline value b_k to reduce the bias of the training, and lines 11-12 set the changes of the neural network parameters for each agent. Line 15 regulates the episode length. In our training, the learning rate α is set to 1×10^{-3}, and the gradient descent is used for neural network parameter update with Adam optimizer [Kingma and Ba, 2014].

3.3 Neural Network Implementation

The neural networks in DeepWeave consist of two stages: GNN processing and Policy Network processing, as illustrated in Figure 4. In this subsection, we detail the two stages.

GNN processes the information of the job DAG. Compared to other popular neural networks, such as convolutional neural network, GNN is more appropriate to process graph-like structured data [Zhou *et al.*, 2018]. Besides, GNN is proved to have a better knowledge transfer ability [Rusek *et al.*, 2019]. For example, when the size of the input data is slightly different from the training history, GNN can still make precise inferences. In coflow scheduling, the training process can hardly cover all job DAG shapes, so the knowledge transfer ability is critical.

In the GNN, both node attributes and job attribute are calculated. First, the information (e.g., the computation cost) of v_i is abstracted as the node's feature x_i in the graph. Then, attribute $attr_i$ of node v_i is calculated based on attributes of v_i's descendant nodes. The attribute information is transmitted

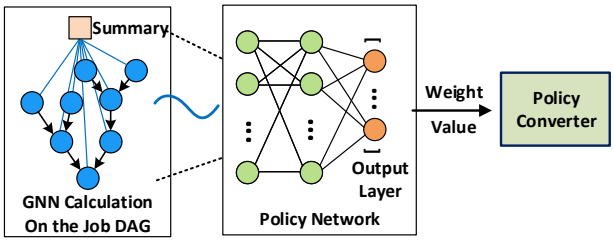

Figure 4: The working mechanism of the neural networks.

from node v_i's descendant nodes (denoted as $des(v_i)$) one by one to v_i through a sequence of message passing steps [Mao *et al.*, 2019]. During these message passing steps, each node calculates its attribute value by $attr_i = F(x_i, \{attr_u | v_u \in des(v_i)\})$. In practice, the function $F(\cdot)$ can be implemented with neural networks. We implement two neural networks in DeepWeave for $F(\cdot)$ as follows:

$$attr_i = f[\sum_{v_u \in des(v_i)} g(attr_u)] + x_i$$

where $f(\cdot)$ and $g(\cdot)$ are two small feedforward neural networks with two hidden layers. This function enables a non-linear transformation of the corresponding input information. In this way, the message passing process can transmit complicated information from a node's descendant nodes to the node. Besides, [Mao *et al.*, 2019] points out that one neural network cannot compute the critical path of a DAG [Kelley Jr, 1961]. Each node attribute is presented as one abstract value. The attribute values are sorted, and the order of each node is recorded based on their values before the values leave the GNN. After the calculation of node attributes, there is also a job-level information summary y. In DeepWeave, another set of neural networks is used for calculating y, where $y = f_{job}(\{attr_i, x_i | v_i \in V\})$.

After the GNN calculation process, the sequence of the job-level information and node attributes are sent to the stage of Policy Network. The Policy Network is implemented using a feedforward neural network with one hidden layer. The feedforward neural network calculates the job-level information and node attributes and generates priority list $P = (p_1, p_2, ..., p_{|V|})$ as the output, where $p_i (1 \le i \le |V|)$ corresponds to the priority value of node v_i in the execution of the job, and a higher p_i stands for a higher priority. Note that the Policy Network cannot change its size as GNN does. In our design, the size of the input layer equals the maximum number of vertices that may appear in the experiment. For small DAGs, the sorted values from node attributes are padded with zeros before they are used as the input of the Policy Network.

3.4 Policy Converter

Policy Converter translates the output value of Policy Network into concrete coflow scheduling policies. As mentioned before, the Policy Network generates priority list $P = (p_1, p_2, ..., p_{|V|})$ for each node in the job DAG. Since a job DAG only has one edge originating from one certain node, we assign the priority of edge e (i.e., a coflow) with the priority value of its source node. Assume that we have priority

Algorithm 2 Scheduling policy of coflows and flows in coflows at ingress ports

Input: priority list P;
Output: flow scheduling priority;
1: **while** $P \neq \emptyset$ **do**
2: Select coflow e_i with the maximum $p_i \in P$;
3: Select the flow $f_i^{m*,n*}$ with the largest size in e_i;
4: Assign $f_i^{m*,n*}$ with the highest priority in the remaining source of ingress port $m*$;
5: Get $End(f_i^{m*,n*})$;
6: **for** ingress port m = 1 to M **do**
7: **if** $f_i^{m,n*} \neq 0$ **then**
8: Add $f_i^{m,n*}$ in ingress port m with the minimum$|End(f_i^{m,n*}) - End(f_i^{m*,n*})|$;
9: **end if**
10: **end for**
11: Remove p_i from P;
12: **end while**

list $P = \{p_1, p_2, , ..., p_{|E|}\}$ that denotes the priority of each coflow. The scheduling problem is to schedule flows at each port based on the priority of the coflow they belong to. The scheduling process includes scheduling at ingress ports and scheduling at egress ports, and the two schedulings s are similar. For simplicity, we only show the scheduling process at ingress ports.

Algorithm 2 details the translation from the priority list to the concrete scheduling process at ingress ports. In Algorithm 2, coflows are scheduled according to their priorities in P. Line 2 selects the coflow with the highest priority to schedule. Lines 3-4 minimize the end time of the coflow by first scheduling the flow with the largest size in the selected coflow. Line 5 gets the end time of the flow and uses this flow's end time as the end time for this coflow. Lines 6-10 schedule other flows on port $n*$. Line 11 removes the scheduled coflow from priority list.

4 Experiment and Evaluation

4.1 Simulation Environment

In this section, we evaluate the performance of DeepWeave. For the simulation environment, we implement a coflow scheduling simulator with Python 3.6. During the training, at each step, the simulator tells the DRL agent about the state information, then the action from the DRL agent is translated into flow scheduling operations to schedule the flows in the simulator. Finally, the simulator calculates the job completion time, which is used to calculate the reward value. A group of previous works [Huang *et al.*, 2016; Zhang *et al.*, 2016; Li *et al.*, 2016; Wang *et al.*, 2019; Tan *et al.*, 2019] used the real-life traces of Facebook [Chowdhury *et al.*, 2014] to train and test the coflow scheduling schemes. Specifically, the training and test data are separately chosen, and they are different from each other. However, the Facebook trace is a collection of single-stage jobs, which do not contain the information of correlations among different coflows. In this paper, we use TPC-DS [tpc, 2019] to evaluate the performance of different schemes in job DAGs. We use Shark [Xin *et al.*, 2013] to generate queries and set the single wave to each stage in the job DAG. The DRL algorithm is implemented on TensorFlow based on Python 3.6. The simulation runs on a desktop computer equipped with an Intel i7700 CPU, GTX 1080Ti Graphics card, and 32G DDR4 RAM. The DRL agent is trained for 60000 iterations in our experiment.

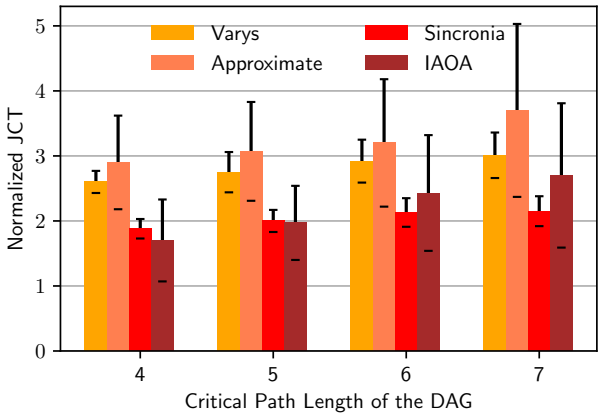

Figure 5: Normalized Average JCT (normalized to DeepWeave).

We evaluate the performance of DeepWeave with the following schemes:

1. **Varys** [Chowdhury *et al.*, 2014] uses a Smallest-Effective-Bottleneck-First (SEBF) heuristic algorithm to greedily schedule a coflow based on its bottleneck flow's completion time.
2. **Approximate** [Shafiee and Ghaderi, 2018] tries to solve a convex optimization problem and provides a polynomial-time deterministic algorithm with an approximation factor of 5 to the optimal solution.
3. **Sincronia** [Agarwal *et al.*, 2018] uses a greedy mechanism to schedule coflows based on a priority list of coflows and can achieve an approximation factor of 4 to the optimal solution.
4. **IAOA** (Information-Agnostic Online Algorithm) [Wang *et al.*, 2019] formulates a weighted coflow completion time minimization problem and proposes a heuristic solution with an approximation factor of 2 to the optimal solution. However, IAOA does not consider the correlations of coflows in job DAGs.

4.2 Simulation Results

Job Completion Time

First, we select a group of TPC-DS queries that consist of DAGs with different shapes. Specifically, the critical path lengths of these DAGs vary from 4 to 7, and each DAG contains 5 queries. Figure 5 shows the average JCTs of different schemes normalized to the JCT of DeepWeave. In this figure, all comparison schemes' JCTs are larger than DeepWeave's in all the tested DAGs. DeepWeave outperforms the best comparison scheme at least 1.7×. When the critical path length in the DAG increases to 7, all comparison

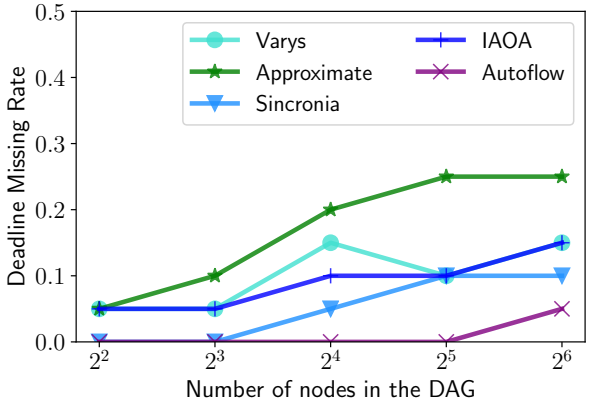

Figure 6: Deadline missing rate.

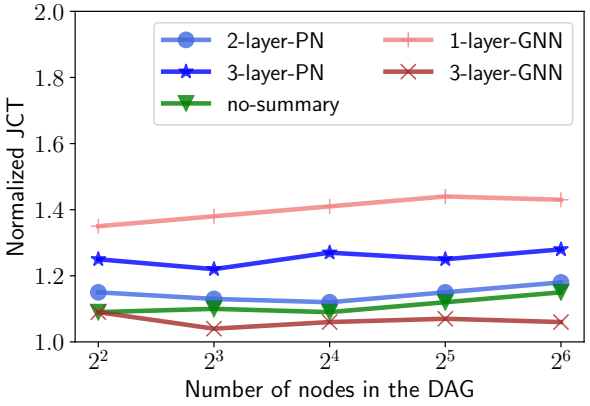

Figure 7: Normalized JCT under different neural networks (normalized to DeepWeave) The result of DeepWeave is one in all cases.

schemes' JCTs are at least twice longer than DeepWeave's. As the DAG's length grows from 4 to 7, Varys and Sincronia have relatively stable performance. In contrast, Approximate and IAOA's performance degrade significantly as the DAG's length increases since do not take the correlations of different coflows into full consideration.

Deadline Missing Rate

From the job level's perspective, the ultimate goal of coflow scheduling is to minimize the completion time of jobs. To this end, we use GGen [Cordeiro *et al.*, 2010] to generate DAGs with various shapes. We use the Fan-in/Fan-out method in GGen and set the number of vertices in the DAGs from 4 (2^2) to 64 (2^6). For DAG shape (i.e., the number of nodes), we generate 20 DAGs and fine-tune the deadline of each DAG to differentiate the performance of different schemes. We use the CustomTraceProducer in CoflowSim [Cof, 2014] to generate coflows for the DAGs and assume that the non-blocking fabric has 50 ingress ports and 50 egress ports. We use the deadline missing rate of job DAGs as the performance metric. The simulation result is shown in Figure 6. In this figure, DeepWeave has the lowest deadline missing rate among all the schemes, while Approximate has the highest deadline missing rate. This result is in accordance with the simulation results of Figure 5.

Impact of Different Parameters

DeepWeave uses two types of neural networks: a GNN and a Policy Network. The neural networks in DeepWeave is fine-tuned to get a good performance. Changing the neural network components may lead to a degradation of the performance. Figure 7 shows the impact of changing neural networks on JCT, which is normalized to JCT of DeepWeave. In this figure, 2-layer-PN denotes using a Policy Network with two hidden layers, 3-layer-PN denotes using a Policy Network with three hidden layers, no-summary denotes the GNN implementation without the summary node, 1-layer-GNN denotes the functions in GNN with one hidden layer, and 3-layer-GNN denotes the functions in GNN with three hidden layers. In the figure, we can see that the neural network change leads to performance degradation. Specifically, 1-layer-GNN performs worst because one hidden layer is not enough to extract enough information for job DAGs. On the other hand, 3-layer-GNN does not outperform the 2-layer-GNN (i.e., the one used in DeepWeave) due to the overfitting of the neural networks.

5 Conclusion

In this paper, we present DeepWeave to automatically generate coflow scheduling policies in job DAGs based on DRL. DeepWeave employs GNN to handle the input job DAG and provides a high generalization ability of the DRL framework on different job DAGs. After training, DeepWeave can generate efficient coflow scheduling policies without human expertise. The main challenge of applying DRL in the generation of coflow scheduling policy comes from the design of the neural network structure and the fine-tuning of the parameters. This work proves the effectiveness of machine learning in designing flow scheduling policies. In the future, we will enrich our work to achieve a more powerful coflow scheduling system on real applications.

Acknowledgements

This paper is supported by the National Key Research and Development Plan under Grant Number 2017YFB0803204, the National Natural Science Fund of China under Grant Numbers 61521003 and 61872382, and the Beijing Institute of Technology Research Fund Program for Young Scholars.

References

[Agarwal *et al.*, 2018] Saksham Agarwal, Shijin Rajakrishnan, Akshay Narayan, Rachit Agarwal, David Shmoys, and Amin Vahdat. Sincronia: near-optimal network design for coflows. In *ACM SIGCOMM*, pages 16–29, 2018.

[Chen *et al.*, 2016] Li Chen, Wei Cui, Baochun Li, and Bo Li. Optimizing coflow completion times with utility max-min fairness. In *IEEE INFOCOM*, pages 1–9, 2016.

[Chowdhury and Stoica, 2012] Mosharaf Chowdhury and Ion Stoica. Coflow: a networking abstraction for cluster applications. In *ACM HotNets*, pages 31–36, 2012.

[Chowdhury and Stoica, 2015] Mosharaf Chowdhury and Ion Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM CCR*, 45(4):393–406, 2015.

[Chowdhury *et al.*, 2011] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM CCR*, 41(4):98–109, 2011.

[Chowdhury *et al.*, 2014] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. Efficient coflow scheduling with varys. *ACM SIGCOMM CCR*, 44(4):443–454, 2014.

[Cof, 2014] Coflowsim. https://github.com/coflow/coflowsim, 2014.

[Cordeiro *et al.*, 2010] Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram, Jean-Marc Vincent, and Frédéric Wagner. Random graph generation for scheduling simulations. In *SIMUTOOLS 2010*, page 60, 2010.

[Dogar *et al.*, 2014] Fahad R Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. Decentralized task-aware scheduling for data center networks. *ACM SIGCOMM CCR*, 44(4):431–442, 2014.

[Gao *et al.*, 2016] Yuanxiang Gao, Hongfang Yu, Shouxi Luo, and Shui Yu. Information-agnostic coflow scheduling with optimal demotion thresholds. In *IEEE ICC*, pages 1–6, 2016.

[Greensmith *et al.*, 2004] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *JMLR*, 5:1471–1530, 2004.

[Huang *et al.*, 2015] Zhe Huang, Bharath Balasubramanian, Michael Wang, Tian Lan, Mung Chiang, and Danny HK Tsang. Need for speed: Cora scheduler for optimizing completion-times in the cloud. In *IEEE INFOCOM*, pages 891–899, 2015.

[Huang *et al.*, 2016] Xin Sunny Huang, Xiaoye Steven Sun, and T Sunflow Ng. Sunflow: Efficient optical circuit scheduling for coflows. In *ACM CONEXT*, pages 297–311, 2016.

[Kelley Jr, 1961] James E Kelley Jr. Critical-path planning and scheduling: Mathematical basis. *Operations research*, 9(3):296–320, 1961.

[Kingma and Ba, 2014] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[Li *et al.*, 2016] Ziyang Li, Yiming Zhang, Dongsheng Li, Kai Chen, and Yuxing Peng. Optas: Decentralized flow monitoring and scheduling for tiny tasks. In *IEEE INFOCOM*, pages 1–9, 2016.

[Mao *et al.*, 2019] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning scheduling algorithms for data processing clusters. In *ACM SIGCOMM*, pages 270–288, 2019.

[Qiu *et al.*, 2015] Zhen Qiu, Cliff Stein, and Yuan Zhong. Minimizing the total weighted completion time of coflows in datacenter networks. In *ACM SPAA*, pages 294–303, 2015.

[Rusek *et al.*, 2019] Krzysztof Rusek, José Suárez-Varela, Albert Mestres, Pere Barlet-Ros, and Albert Cabellos-Aparicio. Unveiling the potential of graph neural networks for network modeling and optimization in sdn. In *ACM SOSR*, pages 140–151, 2019.

[Shafiee and Ghaderi, 2018] Mehrnoosh Shafiee and Javad Ghaderi. An improved bound for minimizing the total weighted completion time of coflows in datacenters. *IEEE/ACM TON*, 26(4):1674–1687, 2018.

[Sutton *et al.*, 2000] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, pages 1057–1063, 2000.

[Tan *et al.*, 2019] Haisheng Tan, Shaofeng H-C Jiang, Yupeng Li, Xiang-Yang Li, Chenzi Zhang, Zhenhua Han, and Francis Chi Moon Lau. Joint online coflow routing and scheduling in data center networks. *IEEE/ACM ToN*, 27(5):1771–1786, 2019.

[tpc, 2019] TPC Benchmark DS (TPC-DS). http://www.tpc.org/tpcds, 2019.

[Wang *et al.*, 2018a] Shuo Wang, Jiao Zhang, , Tao Huang, Tian Pan, Jiang Liu, and Yunjie Liu. Multi-attributes-based coflow scheduling without prior knowledge. *IEEE/ACM ToN*, 26(4):1962–1975, 2018.

[Wang *et al.*, 2018b] Shuo Wang, Jiao Zhang, Tao Huang, Jiang Liu, Tian Pan, and Yunjie Liu. A survey of coflow scheduling schemes for data center networks. *IEEE Communications Magazine*, 56(6):179–185, 2018.

[Wang *et al.*, 2019] Zhiliang Wang, Han Zhang, Xingang Shi, Xia Yin, Yahui Li, Haijun Geng, Qianhong Wu, and Jianwei Liu. Efficient scheduling of weighted coflows in data centers. *IEEE TPDS*, 30(9):2003–2017, 2019.

[Xin *et al.*, 2013] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: Sql and rich analytics at scale. In *ACM SIGMOD*, pages 13–24, 2013.

[Xu and Tang, 2015] Xiaoyong Xu and Maolin Tang. A new approach to the cloud-based heterogeneous mapreduce placement problem. *IEEE TSC*, 9(6):862–871, 2015.

[Zhang *et al.*, 2016] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. Coda: Toward automatically identifying and scheduling coflows in the dark. In *ACM SIGCOMM*, pages 160–173, 2016.

[Zhao *et al.*, 2015] Yangming Zhao, Kai Chen, Wei Bai, Minlan Yu, Chen Tian, Yanhui Geng, Yiming Zhang, Dan Li, and Sheng Wang. Rapier: Integrating routing and scheduling for coflow-aware data center networks. In *IEEE INFOCOM*, pages 424–432, 2015.

[Zhou *et al.*, 2018] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.