Scheduling Dependent Coflows with Guaranteed Job Completion Time

Yang Liu, Wenxin Li, Keqiu Li, Heng Qi, Xiaoyi Tao, Sheng Chen
School of Computer Science and Technology, Dalian University of Technology, China
Keqiu Li is the corresponding author, keqiu@dlut.edu.cn

Abstract—**Today's data center jobs typically follow a coflow model. Each coflow consists of multiple concurrent data flows, while each job is comprised of multiple coflows. Only completing all flows in all coflows is meaningful to a job. To guarantee the job completion time, job deadlines and coflow dependencies must be jointly considered. However, existing solutions mainly consider the coflow scheduling, which are insufficient to guarantee the completion time of jobs with multiple dependent coflows. In this paper, we study the dependent coflow scheduling problem with constraints on job deadlines. Specifically, we formulate a deadline- and dependency-aware optimization problem, and accordingly propose a two-level scheduling method to solve this problem. The first level is to schedule at the job level with a most-bottleneck-first heuristic algorithm. The second level is an intra-job scheduling, which seamlessly combine a prioritized scheduling and a weighted fair scheduling, with the aim of accounting for different coflow dependencies. We conduct comprehensive simulations to evaluate the performance of our two-level scheduling method. Extensive results show that our scheduling method can reduce the job completion time by up to 18%, and accommodate 21% more jobs with deadlines guaranteed, compared to the conventional shortest-job-first method.**

I. Introduction

Data centers have become the major computing platforms for a growing number of data-intensive jobs, such as MapReduce [1], Dryad [2] and Spark [3]. Many of these jobs typically have structured communication pattern, in which a group of data flows need to pass through a sequence of intermediate computation stages before generating the final results. These intermediate flow transfers can account for more than 50% of job completion time, and ultimately make these data-intensive jobs become network-bound [4]. In such a case, an emerging problem of scheduling such intermediate flow transfers is becoming increasingly important for job completion time.

Recent studies by Chowdhury *et al.* [5, 6] have shown that scheduling those flow transfers at the level of coflow rather than the individual flow level can bring potential benefits for job completion time. The coflow is defined as the set of concurrent flows transferring between two computation stages of a job. Such coflow abstraction builds upon the *all-of-nothing* property observed in many of the data-parallel computing jobs—all flows must be finished before the coflow is considered to be completed. From a job's perspective, the coflow abstraction can account for the job-specific communication requirements, and provides opportunities to decrease the job completion time by optimizing such coflow's completion time.

Simply optimizing such coflow transfers for decreasing job completion time encounters two significant challenges from a practical application point of view. First, many of these data-intensive jobs are composed of multiple computation stages with complex traffic interactions. For example, in pipelining jobs (e.g., Dryad [2], SCOPE [7], MapReduce Online [8]), or jobs with explicated barrier and iterative computation requirements (e.g., Spark and its variants [9, 10], Pig[11], Hive[12]), multiple computation stages should be successfully executed. And the intermediate flow transfer issued by a computation stage depends on whether it has been generated by this stage. Nevertheless, such generation actually depends on the completion of flow transfer issued by the prior computation stage. This eventually leads to the dependencies between coflows of a job, and thus scheduling at the coflow level may not always result in faster job completion time. Second, many of those jobs have strong deadline requirements, especially in the cloud computing environments (e.g., Amazon EC2 [13] and Microsoft Azure[14]) where tenants renting the virtual machines based on the time they used. For instances, in the well-known Interactive Analytical Processing jobs or the On-Line Analytical Processing jobs, complex queries need to be processed with temporal requirements [15]. These jobs typically need the results being generated before their pre-specified temporal deadlines. So, if the job scheduling policies are unaware of such temporal deadlines, jobs may suffer unpredictable performance, which directly impacts the payment of the job sponsors (tenants in the cloud). Bearing these points in mind, we believe that job deadlines and coflow dependencies must be jointly considered, so that the job completion time can have a chance to be guaranteed. However, existing solutions for network flow optimization have significant limitations. Some of them may provide deadline guarantees on the flow scheduling, but are coflow-agnostic; existing solutions on coflow optimization may reduce the coflow completion time, but they are unaware of the coflow dependencies.

In this paper, we focus on scheduling the dependent coflows with the primary objective of guaranteeing the deadlines of jobs. Specifically, we formulate a deadline- and dependency-aware optimization problem. With this optimization, we incorporates various coflow dependencies, taking into account practical constraints of both link capabilities and job deadlines. To solve this problem, we present a two-level scheduling method. At the first level, we present a most-bottleneck-first heuristic algorithm to schedule at job-level. At the second level, we seamlessly combine a prioritized scheduling and a weighted fair scheduling. This level can account for both strong and weak dependencies between coflows. Finally, we conduct comprehensive simulations to evaluate the performance of our proposed method. Simulation results have shown that our proposed scheduling method can reduce the job completion time by up to 18% and at the same time, can accommodate 21% more jobs with deadlines guaranteed, compared to the traditional shortest-job-first method [16].

In summary, our main contributions are as follows:

1) we observe that deadlines and coflow dependencies must be jointly considered, in order to guarantee the completion of today's data center jobs, and accordingly formulate an optimization problem.
2) We propose a two-level scheduling method to solve the problem. The first level is to schedule at the job level with a most-bottleneck-first heuristic algorithm. The second level is an intra-job scheduling performed upon a dependency graph. This level seamlessly combines a prioritized scheduling and a weighted fair scheduling, and can account for different coflow dependencies.
3) We conduct comprehensive simulation to show the efficiency of our proposed two-level scheduling method, with respect to the job completion time and the ratio of jobs with deadlines guaranteed.

The rest of our paper is organized as follows. We describe the model of data center network and problem formulation in Section II. In Section III, we propose a two-level scheduling method, which are the main contribution of this paper. Then, we conduct series of formulation experiments to evaluate the performance of our algorithm in Section IV. Section V summarizes the related work. Finally, we conclude this paper in Section VI.

II. System Model

We first describe the model of data center network in this section, and then present the problem formulation.

A. Data Center Network Model

In our analysis, we abstract the entire data center fabric as a non-blocking switch interconnecting all the servers. This abstraction has been widely used for simple, yet practical design of flow scheduling in data center networks [6, 16, 17]. The key insight is that existing works such as multi-path routing (e.g., [18]) and multi-tree topologies (e.g., [19, 20]) have considerably improved bisection bandwidth, and the recent techniques can actually enforce the edge constraints into the network [21, 22]. So, we only need to focus on a full bisection bandwidth network where the physical bandwidth of servers can be fully utilized without considering the bottleneck links inside the data center.

In this paper, we consider a data center consisting of M servers, as shown in Fig. 1. Let $\mathcal{M} = \{s_1, s_2, \cdots, s_M\}$ denote the set of servers in the data center. Each server s_m is connected to a non-blocking switch via dedicated uplink s_m^{out} and downlink s_m^{in}. To indicate the link bandwidth capacity,

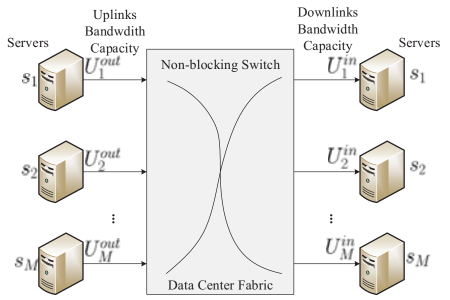


Fig. 1. Data center network model, where all servers are connected to a non-blocking switch.

let U_m^{in} and U_m^{out} denote the bandwidth capacities for s_m^{in} and s_m^{out}, respectively. In such a case, the only sources of contention take place in these uplinks and downlinks when given a set of jobs in a certain time period. To more precisely capture this point, let $[\Gamma_0, \Gamma_1]$ be a fixed time interval, during which a set $\mathcal{J} = \{J_1, J_2, \cdots, J_J\}$ of jobs has to be handled.

As aforementioned, jobs in the cloud have strong deadline requirements, and important jobs have shorter deadlines in comparison to less urgent jobs. To capture such deadline diversity, let D_i denote the pre-specified deadline by job J_i. Specifically, let t_i represent the start time of J_i. Accordingly, each job $J_i \in \mathcal{J}$ must be completed within the time interval $[t_i, D_i]$. It should be noted that the constraint $\Gamma_0 \le t_i < D_i \le \Gamma_1, \forall J_i \in \mathcal{J}$ should be satisfied when jobs specify the corresponding deadlines. Otherwise, jobs will be considered not must to be handled in the interval $[\Gamma_0, \Gamma_1]$, and will be either rejected or shifted to the next time interval. Let T_i denote the time at which J_i is completed.

As we know, many jobs involve multiple computation stages. Due to the complex traffic interaction between these stages, multiple coflows can be generated by a job. So, let $\mathcal{C}_i$ denote the coflow set of job J_i. Specifically, let $c_{i,u}$ represent the u-th coflow of job J_i. Let $d_{m,n}^{i,u}$ denote the amount of data belonged to $c_{i,u}$, that needs to be transferred from server s_m to server s_n. The corresponding bandwidth allocated to such data transfer $d_{m,n}^{i,u}$ at time t is denoted by $r_{m,n}^{i,u}(t)$, which is the decision variable throughout this paper. Given the coflow set of each job, each server actually host a set of jobs that desire the ingress/egress bandwidth on this server. More precisely, let $\mathcal{J}_{s_m^{out}}$ and $\mathcal{J}_{s_m^{in}}$ denote the collections of jobs on both the uplink s_m^{out} and downlink s_m^{in}, respectively. Recall that a coflow contains multiple parallel flows [5],everyone of which needs to be finished before the coflow is considered to be completed. From the coflow's point of view, let $T_{i,u}$ denote the time at which the coflow $c_{i,u}$ is finished. Moreover, since the explicit barriers or pipelining requirements are desired for some jobs, multiple coflows of a same job may have some dependencies. Here, we consider the following two types of dependencies:

1) *Strong coflow dependency* ($c_{i,u} > c_{i,v}$): In presence of explicit barriers applied in jobs, a coflow $c_{i,v}$ can not

start until its dependent coflow $c_{i,u}$ has finished.

2) *Weak coflow dependency* ($c_{i,u} \geq c_{i,v}$): In jobs using pipelining to avoid explicit barriers, $c_{i,v}$ can coexist with $c_{i,u}$, but it cannot finish until $c_{i,u}$ has finished.

It should be noted that coflows in different jobs can be unrelated to each other. Important notations used throughout this paper is listed in Table I.

B. Problem Formulation

The goal of this paper is to schedule a set of jobs in a given time interval for minimizing the average job completion time and, at the same time, being aware of both job deadlines and coflow dependencies within each job. By giving the data center network model above, we can formulate the following deadline and dependency aware job scheduling problem (D2JSP).

$$\min \quad \frac{1}{|\mathcal{J}|}\sum_{J_i\in\mathcal{J}}(T_i - t_i) \tag{1}$$

$$T_i = \max_{c_{i,u}\in\mathcal{C}_i} T_{i,u}, \forall J_i \in \mathcal{J}; \tag{2}$$

$$\int_{t_i}^{T_{i,u}} r_{m,n}^{i,u}(t)dt = d_{m,n}^{i,u}, \forall c_{i,u}\in\mathcal{C}_i, \forall J_i\in\mathcal{J}, \\ \forall s_m, s_n \in \mathcal{M}, s_m \neq s_n, \forall T_{i,u}\in(t_i, D_i]; \tag{3}$$

$$\sum_{s_m\in\mathcal{M}}\sum_{s_n\in\mathcal{M}} r_{m,n}^{i,v}(t) = 0, \forall J_i \in \mathcal{J}, \\ \forall c_{i,u}, c_{i,v}\in\mathcal{C}_i, c_{i,u} > c_{i,v}, \forall t\in[t_i, T_{i,u}]; \tag{4}$$

$$T_{i,v} \geq T_{i,u}, \forall J_i\in\mathcal{J}, \forall c_{i,u}, c_{i,v}\in\mathcal{C}_i, c_{i,u}\geq c_{i,v}; \tag{5}$$

$$\sum_{J_i\in\mathcal{J}}\sum_{c_{i,u}\in\mathcal{C}_i}\sum_{s_n\in\mathcal{M}, s_n\neq s_m} r_{m,n}^{i,u}(t) \leq U_m^{out}, \\ \forall s_m\in\mathcal{M}, \forall t\in[\Gamma_0, \Gamma_1]; \tag{6}$$

$$\sum_{J_i\in\mathcal{J}}\sum_{c_{i,u}\in\mathcal{C}_i}\sum_{s_m\in\mathcal{M}, s_m\neq s_n} r_{m,n}^{i,u}(t) \leq U_n^{in}, \\ \forall s_n\in\mathcal{M}, \forall t\in[\Gamma_0, \Gamma_1]; \tag{7}$$

$$0 \leq r_{m,n}^{i,u}(t) \leq \min\{U_m^{out}, U_n^{in}\}, \forall c_{i,u}\in\mathcal{C}_i, \forall J_i\in\mathcal{J}, \\ \forall s_m, s_n\in\mathcal{M}, s_m\neq s_n, \forall t\in[t_i, D_i]; \tag{8}$$

$$r_{m,n}^{i,u}(t) = 0, \forall c_{i,u}\in\mathcal{C}_i, \forall J_i\in\mathcal{J}, \\ \forall s_m, s_n\in\mathcal{M}, s_m\neq s_n, \forall t\in[\Gamma_0, t_i)\cup(D_i, \Gamma_1]. \tag{9}$$

Clearly, the objective in Eq. (1) is to minimize the average job completion time across all jobs in $\mathcal{J}$. Specifically, each job $J_i \in \mathcal{J}$ is considered to be completed only when all coflows of it have finished, as shown in Eq. (2). A schedule is called feasible if all flows in each job J_i can be accomplished within the corresponding valid time interval $[t_i, T_{i,u}]$, as shown in Eq. (3). Given the coflow dependencies, we get two constraints for the decision variable $r_{m,n}^{i,v}(t)$. First, the constraint in Eq. (4) ensures that if two coflows have strong dependency (i.e., $c_{i,u} > c_{i,v}$), then all flows of $c_{i,v}$ cannot get any bandwidth when $c_{i,u}$ is running. That is, a certain coflow should wait until its strong dependent coflow has finished. Second, the constraint in Eq. (5), enforces the completion time of a certain coflow $c_{i,v}$ to be larger than that of its weak dependent coflow $c_{i,u}$. It is obvious that across the time interval $[\Gamma_0, \Gamma_1]$, the total amount of consumed bandwidth on the uplink s_m^{out} (downlink s_m^{in}) of each server $s_m \in \mathcal{M}$ should not exceed the corresponding bandwidth capacity U_m^{out} (U_m^{in}), as shown in Eq. (6) (Eq. (7)). Finally, Eq. (8) and Eq. (9) present the value range for the decision variable $r_{m,n}^{i,u}(t)$. Specifically, for every flow of a certain job, the value of $r_{m,n}^{i,u}(t)$ should be limited from 0 to $\min\{U_m^{out}, U_n^{in}\}$, within the transmission time interval $[t_i, D_i]$ of this job (Eq. (8)). In addition, when the time is not at the range of $[t_i, D_i]$ of J_i, all flows in this job shall not get any bandwidth (Eq. (9)).

TABLE I
NOTATIONS AND DEFINITIONS

Symbol	Definition
$\mathcal{M}$	The set of servers in the data center, $\mathcal{M} = \{s_1, s_2, \cdots, s_M\}$
s_m^{out}	The uplink of server s_m
s_m^{in}	The downlink of server s_m
U_m^{out}	The bandwidth capacity of the uplink s_m^{out}
U_m^{in}	The bandwidth capacity of the downlink s_m^{in}
$\mathcal{J}$	The set of jobs in a given time interval $[\Gamma_0, \Gamma_1]$, $\mathcal{J} = \{J_1, J_2, \cdots, J_J\}$
$\mathcal{J}_{s_m^{out}}$	The set of jobs on s_m^{out}
$\mathcal{J}_{s_m^{in}}$	The set of jobs on s_m^{in}
t_i	The release time of job $J_i \in \mathcal{J}$
D_i	The deadline pre-specified by job J_i
T_i	The time at which job J_i is completed
$\mathcal{C}_i$	The coflow set of J_i
$c_{i,u}$	The u-th coflow in $\mathcal{C}_i$
$T_{i,u}$	The time at which coflow $c_{i,u}$ is completed
$d_{m,n}^{i,u}$	The amount of data, belonging to coflow $c_{i,u}$, that needs to be transferred from server s_m to s_n.
$r_{m,n}^{i,u}(t)$	The amount of bandwidth, at time t, that is allocated to the data transfer of $d_{m,n}^{i,u}$

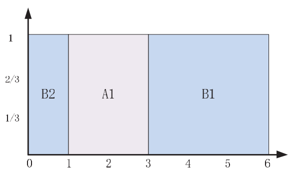
III. Two-Level Scheduling Method

In this section, we present a two-level scheduling model to resolve the D2JSP problem. The first level is to scheduling at the job level, while the second level is to perform an intra-job scheduling.

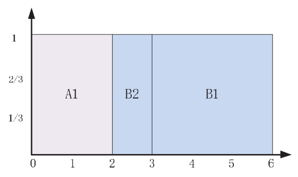
A. Inter-Job Scheduling

Instead of scheduling at the coflow level, we start by scheduling at the job level. The key insight is that optimizing the coflow completion time may not always lead to shorter job completion time. Moreover, scheduling at the coflow level can significantly break the dependencies between coflows belonged to the same job. To have a comprehensively understanding, consider a straightforward example, which is shown as follows:

Example 1: Suppose that there are two jobs A and B in a given time period. The job A has one coflow A_1, while B contains two coflows: B_1 and B_2. The time that these coflows (A_1, B_1, B_2) takes to finish, under uncontested environment, are $2t, 3t, t$, respectively. A possible schedule at the coflow level (e.g., shortest first [16]) can get a sequence of $\{B_2, A_1, B_1\}$, as shown in Fig.2(a). In such a case, the average job completion time is $4.5t$, with completion time for A and B being $3t$ and $6t$, respectively. If scheduling at the level of job, a possible way is to schedule job B



(a) Scheduling at coflow level



(b) Scheduling at job level

Fig. 2. A motivating example of jointly considering the coflow dependency and job deadline.

after A (e.g., $\{A_1, B_2, B_1\}$), as shown in Fig.2(b), and we obtain an average job completion time of $4t$. This implies that the job-level scheduling outperforms the coflow level scheduling, in terms of the job completion time. Moreover, when considering the deadlines (e.g., $D_A = 2.5t, D_B = 6t$) and the dependencies (e.g., $B_1 > B_2$), jobs are likely to miss their deadlines, and can even be failed to complete due to the disrupt of coflow dependencies.

Given the motivating example above, we first give a characterization of a job scheduling algorithm for a set of jobs in a fixed time interval. The characterization is based on the notion of a *bottleneck interval* for $\mathcal{J}$, which is an interval extended from [23], in which a group of jobs must be scheduled at maximum constant speed.

Definition 1: The *intensity* of a time interval $I = [a, b]$ on the uplink s_m^{out} is defined by

$$\psi(I, s_m^{out}) = \frac{\sum_{[t_i,D_i]\subseteq[a,b]\wedge J_i\in\mathcal{J}} \sum_{c_{i,u}\in\mathcal{C}_i} \sum_{s_n\in\mathcal{M}} d_{m,n}^{i,u}}{a \sim b}, \tag{10}$$

where $a \sim b$ denotes the available time in interval $[a, b]$. Similarly, for the downlink s_n^{in}, we have

$$\psi(I, s_m^{in}) = \frac{\sum_{[t_i,D_i]\subseteq[a,b]\wedge J_i\in\mathcal{J}} \sum_{c_{i,u}\in\mathcal{C}_i} \sum_{s_m\in\mathcal{M}} d_{m,n}^{i,u}}{a \sim b}, \tag{11}$$

It is clear that $\psi(I, s_m^{out})$ ($\psi(I, s_m^{in})$) is the lower bound on the average transmission rate on the uplink s_m^{out} (downlink s_m^{in}). In the following, we present the definition for guiding the design of the job scheduling algorithm.

Definition 2: If an interval $I^* = [a, b]$ maximizes $\psi(I, s_m^{out})$ for the uplink of any $s_m \in \mathcal{M}$, we call I^* a bottleneck interval and s_m^{out} is the corresponding bottleneck uplink. The bottleneck downlink s_m^{in} can be defined in a similar way.

Now we present the job scheduling algorithm that greedily computes bottleneck intervals iteratively. **Algorithm** 1 shows the **Most-Bottleneck-First** schedule on the uplinks. It begins by greedily identifying a bottleneck interval I^* and bottleneck uplink $s_{m^*}^{out}$ through computing the maximum value of $\psi(I, s_m^{out})$ (Step 2), and schedule the jobs of J^* by using the *earliest deadline first* policy (Step 3). It then schedules each $J_i \in \mathcal{J}$ at transmission rate $\psi(I^*, s_{m^*}^{out})$ on the bottleneck uplink $s_{m^*}^{out}$ (Step 5). To maintain the coflow dependencies in advance and, at the same time, schedule at the job level, each J_i is scheduled on the involved uplinks except $s_{m^*}^{out}$, with guaranteeing that J_i can be completed within $[t_i^*, D_i^*]$ (Step 6). Finally, it updates the set of jobs that have not yet

Algorithm 1 Most-Bottleneck-First on uplinks

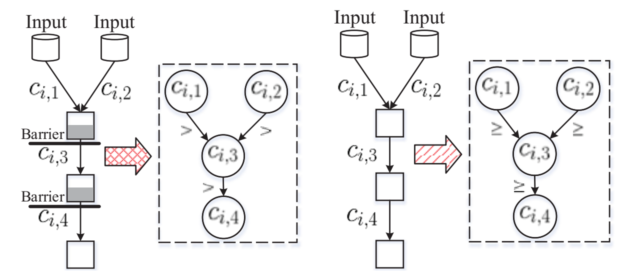
```
Input: $\mathcal{J}$; $\mathcal{J}_{s_m^{out}}, \mathcal{J}_{s_m^{in}}, \forall s_m \in \mathcal{M}$; $d_{m,n}^{i,u}, \forall c_{i,u} \in \mathcal{C}_i, \forall J_i \in \mathcal{J}, \forall s_m, s_n \in \mathcal{M}$
Output: transmission rate $r_{i,s_m^{out}}$ and transmission time interval $[t_i^*, D_i^*]$ for each $J_i \in \mathcal{J}$ on each $s_m \in \mathcal{M}$
 1: while $\mathcal{J} \neq \emptyset$ do
 2:     Find the bottleneck interval $I^*$ and the bottleneck uplink $s_{m^*}^{out}$. The jobs in this interval can be represented by $\mathcal{J}^* = \{J_i | [t_i, D_i] \subseteq I^* \wedge J_i \in \mathcal{J}_{s_{m^*}^{out}}\}$ and without loss of generality,
            $I^* = [a, b] = [\min_{J_i\in\mathcal{J}^*} t_i, \max_{J_i\in\mathcal{J}^*} D_i]$.
 3:     Schedule jobs in $\mathcal{J}^*$ with the Earliest Deadline First policy, and sort all jobs in $\mathcal{J}^*$ increasingly according to their deadlines.
 4:     for $J_i \in \mathcal{J}^*$ do
 5:         Calculate the transmission rate for $J_i$ on $s_{m^*}^{out}$
                $r_{i,s_{m^*}^{out}} = \frac{\sum_{J_i\in\mathcal{J}^*} \sum_{c_{i,u}\in\mathcal{C}_i} \sum_{s_n\in\mathcal{M}} d_{m^*,n}^{i,u}}{a \sim b}$,
            the transmission interval $[t_i^*, D_i^*]$ is also determined.
 6:         For each $s_{m'} \neq s_{m^*} \in \mathcal{M}$, if $J_i \in \mathcal{J}_{s_{m'}^{out}}$, then schedule $J_i$ on the uplink $s_{m'}^{out}$ at the interval $[t_i^*, D_i^*]$ and use the transmission rate
                $r_{i,s_{m'}^{out}} = \frac{\sum_{c_{i,u}\in\mathcal{C}_i} \sum_{s_n\in\mathcal{M}} d_{m',n}^{i,u}}{D_i^* - t_i^*}$,
 7:     end for
 8:     for $J_i \in \mathcal{J}^*$ do
 9:         $\mathcal{J} \leftarrow \mathcal{J} \backslash J_i$;
10:         For each $s_m \in \mathcal{M}$, if $J_i \in \mathcal{J}_{s_m^{out}}$, then mark the time interval $[t_i^*, D_i^*]$ as unavailable on $s_m^{out}$ and let $\mathcal{J}_{s_m^{out}} \leftarrow \mathcal{J}_{s_m^{out}} \backslash J_i$.
11:     end for
12: end while
```

been scheduled, and the available time interval on each uplinks (Step 8-11). It should be noted that the schedule process on the downlinks of all servers is similar to **Algorithm** 1.

The **Most-Bottleneck-First** scheduling policy ensures that each job of J^* can be executed completely on all the involved links, and can be transmitted at the maximum rate on the bottleneck links without missing D . Therefore, by iteratively computing a sequence of bottleneck intervals as well as the corresponding bottleneck links, such **Most-Bottleneck-First** schedule immediately leads to an optimal schedule. The proof process is similar to that in [24], and will not be presented here due to the page limit.

B. Intra-Job Scheduling

Based on the above **Most-Bottleneck-First** scheduling policy, each job can obtain an amount of bandwidth on each of the involved uplinks and downlinks. Now, it's time to schedule the coflows and the corresponding individual flows of each job



(a) Strong coflow dependency graph for jobs with explicit barriers (b) Weak coflow dependency graph for jobs with pipelining

Fig. 3. An illustrative example of coflow dependency graph.

by allocating each job's bandwidth. We call such schedule as *intra-job scheduling*. More preciously, each coflow of a job J_i should be allocated an amount of bandwidth for transferring its flows completely, within the corresponding transmission time interval $[t_i^*, D_i^*]$.

To perform efficient intra-coflow scheduling, we construct a graph for describing the dependencies between multiple coflows belonged to a mutual job. It should be noted that each job has only one type of coflow dependency. For example, a job having iterative computing requirements only contains the strong dependency between its coflows, while a job with pipelining only has weak coflow dependency. Accordingly, we consider two types of such dependency graph, as shown in Fig. 3. In the strong dependency graph, a coflow must not be completed before its dependent coflow has finished (e.g., $c_{i,3} > c_{i,4}$ in Fig. 3(a)). While in the weak dependency graph, a coflow can coexist with its dependent coflows, but cannot be completed until its dependent coflow has finished (e.g., $c_{i,3} \geq c_{i,4}$ in Fig. 3(b)). Note that coflows in different branches of the dependent graph is unrelated to each other. Given such coflow dependency graph, we design *prioritized scheduling* and *weighted fair scheduling* methods, for jobs with strong and weak dependencies, respectively. The *prioritized scheduling* is to sort the coflows according to their in-degree value in the corresponding graph and, at each time, schedule a set of unrelated coflows. It should be noted that the unrelated coflows have the same priority, and can further be prioritized during contention. The *weighted fair scheduling* is to schedule all coflows in a job that contains weak dependencies simultaneously, the transmission rate for each coflow is based on coflow size. The rationale for such weighted fair scheduling is that all coflows can be completed at the same time. The detail process of intra-job scheduling is shown in **Algorithm** 2.

Algorithm 2 Intra-job scheduling

```
Input: $\mathcal{C}_i, [t_i^*, D_i^*], r_{i,s_m^{out}}, r_{i,s_m^{in}}, \forall J_i \in \mathcal{J}, s_m \in \mathcal{M}$;
Output: transmission rate $r_{m,n}^{i,u}, \forall s_m, s_n \in \mathcal{M}, \forall c_{i,u} \in \mathcal{C}_i, \forall J_i \in \mathcal{J}$
1: for $J_i \in \mathcal{J}^*$ do
2:     if $J_i$ only has strong coflow dependencies then
3:         Construct a strong coflow dependency graph, and sort coflows of $\mathcal{C}_i$ according to their in-degree values.
4:         Calculate the transmission rate for each coflow $c_{i,u} \in \mathcal{C}_i$, e.g., $r_{s_m^{out}}^{i,u} = r_{i,s_m^{out}}$, $r_{s_m^{in}}^{i,u} = r_{i,s_m^{in}}$.
5:     else
6:         Schedule all coflows of $\mathcal{C}_i$ at the same time and for each $c_{i,u}$, use the following transmission rate
               $r_{s_m^{out}}^{i,u} = r_{i,s_m^{out}} \times \frac{\sum_{s_n \in \mathcal{M}} d_{m,n}^{i,u}}{\sum_{c_{i,u} \in \mathcal{C}_i} \sum_{s_n \in \mathcal{M}} d_{m,n}^{i,u}}$,
               $r_{s_m^{in}}^{i,u} = r_{i,s_m^{in}} \times \frac{\sum_{s_n \in \mathcal{M}} d_{n,m}^{i,u}}{\sum_{c_{i,u} \in \mathcal{C}_i} \sum_{s_n \in \mathcal{M}} d_{m,n}^{i,u}}$.
7:     end if
8: end for
9: Enforcing all flows of a coflow are completed simultaneously by performing an intra-coflow schedule policy [4].
```

IV. Performance Evaluation

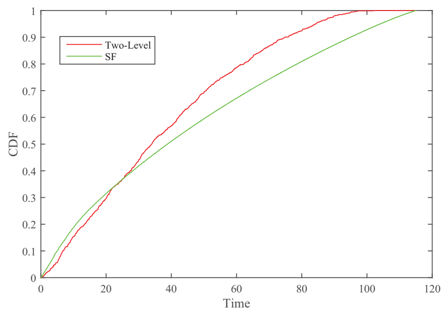
In this section, we conduct comprehensive simulations to evaluate the performance of our two-level scheduling method.

A. Experiment Setting

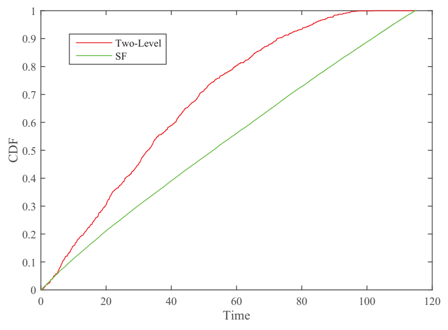
We conduct a series of simulations to show the efficiency of our two-level scheduling method. We consider that the start time, deadline of each job, and the information of flow data are all known in advance. It should be noted that we take coflow as the smallest scheduling unit. The flow information is mainly used for calculating coflow's or job's data size, and finding the most bottleneck sever. We simulate a data center with 128 servers, and generate 1000 jobs between these servers. Each job consists of multiple coflows, whose dependency relationships are randomly generated. The start times and deadlines of job are randomly selected in the interval of [0,100]. Without loss of generality, the capacity of uplinks and downlinks of severs are all set to 1 Gbps. We compare our two-level scheduling method with the shortest-job-first method (shorted as SF), which always schedules jobs with shortest job size (e.g., data flow size) [16]. In this method, once a job is scheduled, it will occupy the bandwidth until it has been finished, raising the risk of missing deadlines of other jobs. In the following, we show the simulation results with respect to the performance on both the completion time and deadline.

B. Performance on completion time

In order to evaluate the performance of our algorithm, we record the completion time of every job and even every coflow. Fig.4 shows the CDF of job completion time (Fig.4(a)) and coflow completion time (Fig.4(b)). As shown in Fig.4(a), SF has a higher CDF of completion time than our two-level method at the beginning. This is because that SF prefers jobs that take shorter time to complete. We can further observe that with our two-level method, more than 90% of jobs can be completed within 80 ms, while that value for SF is around 80%. For coflows (shown in Fig.4(b)), our two-level method always maintain a higher CDF figure than SF. The reason is that the data size of coflows is much less than that of

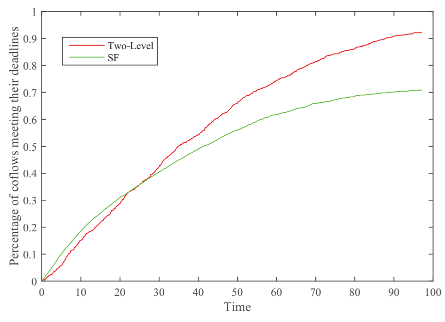


(a) CDF of job completion time

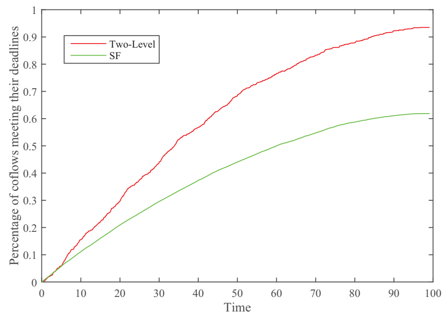


(b) CDF of coflow completion time

Fig. 4. The CDF of job and coflow completion time.



(a) Scheduling results at job level



(b) Scheduling results at coflow level

Fig. 5. The performance of scheduling at different level.

jobs. In addition, it reflects that our prioritized scheduling and weighted fair scheduling methods for coflow work well on guaranteeing job completion time. In conclusion, our two-level method can reduce the job and coflow completion time by up to 18%, compared with SF.

C. Performance on deadline

As aforementioned, jobs may miss their deadlines before or after being scheduled. So, we measure the number of jobs and coflows that meet their deadlines along the time axis. As illustrated in Fig.5(a), our most bottleneck first scheduling method can accommodates 90% percentages of jobs with deadlines guaranteed, while the percentage for SF is only 70%. We also observe that SF has a little more jobs meeting their deadlines before 30 ms. It confirms that our two-level method guarantees job deadlines by firstly allocating bandwidth for the most bottleneck sever instead of simply scheduling jobs based on completion time or data size, which has the limitations on large jobs or flows with early deadlines. From another point of view, it also means that our two-level method focuses on a long-term optimization.

With SF method, the ratio of meeting deadlines for coflow is about 10 percent smaller than that for the job; but with our two-level method, it is almost the same ratio for both job and coflow. Considering that one job consists of a random number of coflows, this result reflects that the 30% jobs missing their deadlines consists of 40% coflows by SF and the result will even fluctuate with the change of the job set. Consequently, the performance of our two-level method is more stable, irrespective of the job set, and accommodates 21% more jobs with deadlines guaranteed, compared to the conventional shortest-job-first method.

V. Related work

There are plenty of related works on network flow optimization. However, none of them are in place to provide guaranteed job completion time, with both job deadlines and coflow dependencies considered. Regarding the network flow optimization, existing solutions can be categorized into two folds: the flow-level scheduling and the coflow-level scheduling. The flow-level scheduling cannot account for the collective behaviors of flows due to the lack of application-level semantics [25]. The coflow scheduling can capture such application-level semantics to some extent, but it cannot account for the coflow dependencies, thus being efficient to provide guaranteed job completion time [4, 6]. Regarding the scheduling on guarantee deadlines, there exist many related works such as [16, 26]. They use priority methods to schedule flows. For example, PDQ provides a distributed priority method based on flow size and deadline. pFabric proposes a light-weight priority queue to schedule flows. These methods may provide deadline guarantees, but they only focus on flow optimization ignoring the significance that performance of a job is determined by all flows.

VI. Conclusion

We study the problem of scheduling dependent coflows with the aim of providing guaranteed job completion time. We present a two-level scheduling method, with the first level applying a Most-Bottleneck-First scheduling algorithm and the second level conducting an intra-job scheduling. Specifically, the Most-Bottleneck-First employs a new definition of intensity to search a most-bottleneck sever heuristically, and then allocate bandwidth and time slots to the set of jobs on this server. The intra-job scheduling performs upon a coflow dependency graph, and seamlessly combines a prioritized scheduling and a weighted fair scheduling to account for different coflow dependencies. Extensive simulation results show that our two-level method can outperform the conventional shortest-job-first method with respect to the performance on both the job completion time and the number of jobs with deadlines guaranteed.

Acknowledge

This work is supported by the National Science Foundation for Distinguished Young Scholars of China (Grant No. 61225010); the State Key Program of National Natural Science of China (Grant 61432002).

References

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Prooceedings of ACM EuroSys*, 2007.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of USENIX HotCloud*, 2010.

[4] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *Proceedings of ACM SIGCOMM*, 2011.

[5] M. Chowdhury and I. Stoica, "Coflow: An application layer abstraction for cluster networking," in *Proceedings of ACM Hotnets*, 2012.

[6] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proceedings of ACM SIGCOMM*, 2014.

[7] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proceedings of VLDB Endowment*, 2008.

[8] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of USENIX NSDI*, 2010.

[9] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters," in *Proceedings of USENIX HotCloud*, 2012.

[10] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of USENIX NSDI*, 2012.

[11] "Apache hama," http://hama.apache.org.

[12] "Apache hive," http://hadoop.apache.org/hive.

[13] "Amazon ec2," https://aws.amazon.com/ec2/.

[14] "Microsoft azure," https://azure.microsoft.com/.

[15] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of VLDB Endowment*, 2012.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," in *Proceedings of ACM SIGCOMM*, 2013.

[17] M. Chowdhury and I. Stoica, "Efficient coflow scheduling without prior knowledge," in *Proceedings of ACM SIGCOMM*, 2015.

[18] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley, "Improving datacenter performance and robustness with multipath tcp," in *Proceedings of ACM SIGCOMM*, 2011.

[19] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," in *Proceedings of ACM SIGCOMM*, 2009.

[20] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "Portland: a scalable fault-tolerant layer 2 data center network fabric," in *Proceedings of ACM SIGCOMM*, 2009.

[21] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *Proceedings of ACM SIGCOMM*, 2011.

[22] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi, "Participatory networking: An api for application control of sdns," in *Proceedings of ACM SIGCOMM*, 2013.

[23] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proceedings of IEEE FOCS*, 1995.

[24] L. Wang, F. Zhang, K. Zheng, A. V. Vasilakos, S. Ren, and Z. Liu, "Energy-efficient flow scheduling and routing with hard deadlines in data center networks," in *Proceedings of IEEE ICDCS*, 2014.

[25] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks," in *Proceedings of USENIX NSDI*, 2010.

[26] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," in *Proceedings of ACM SIGCOMM*, 2012.