Computer Networks 184 (2021) 107686

Contents lists available at ScienceDirect

Computer Networks

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

Scheduling coflows of multi-stage jobs under network resource constraints

Yue Zeng [a], Baoliu Ye [a,*], Bin Tang [b,*], Songtao Guo [c], Zhihao Qu [b]

[a] National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
[b] School of Computer and Information, Hohai University, Nanjing 211100, China
[c] School of Computer Science, Chongqing University, Chongqing 400044, China

ARTICLE INFO

Keywords:
Coflow
Multi-stage job
Network resource
Scheduling algorithm
Approximation algorithm

ABSTRACT

As an emerging network abstraction, coflow greatly improves the communication performance of data-parallel computing jobs. Many existing studies have focused on the design of coflow scheduling to minimize the completion time of jobs. However, they treat the underlying network as a large non-blocking switch without considering the constraints of network resources, which may increase network bottlenecks, reduce link capacity utilization, and extend job completion time. In this paper, we take network resource constraints into account and study how to schedule coflows in multi-stage jobs with the objective of minimizing the total weighted job completion time. We first formalize this multi-stage job scheduling problem as nonlinear programming and prove its NP-hardness. By introducing a priority order of jobs using a linear programming relaxation-based approach, we propose a polynomial-time algorithm with a performance guarantee, which can achieve a constant approximation ratio in many typical scenarios. Simulation results based on a Facebook trace show that, compared with a state-of-the-art approach, our algorithm can shorten the average weighted job completion time by up to 48.46% and run faster by up to 19.12×.

1. Introduction

In data centers, data-parallel cluster computation frameworks (e.g., MapReduce [1], Dryad [2], Spark [3], and Hadoop [4]) provide powerful computing ability for cloud computing, artificial intelligence, and big data. In these frameworks, performing a job requires to manipulate the data stored on thousands of machines, involving massive amounts of data transfer. According to [5], the data transfer phase is critical for each job, accounting for more than 50% of its completion time.

With the traditional network abstraction, "flow" has been used to improve the data transmission performance of jobs, such as file transfers and web access [6]. Among them, flow abstraction captures a series of packets from a single source to a single destination. To cater to the data-parallel computing paradigm, a new network abstraction "coflow" is proposed [7]. Coflow is defined as a collection of flows between two groups of machines with associated semantics and a collective objective, which is more suitable for data-parallel computing jobs. This is mainly because, in the data-parallel computing paradigm, the data transmission of a job usually involves a collection flows, rather than individual flows. To improve the communication performance of data-parallel computing jobs, many scheduling methods [8–14] have been proposed to optimize Coflow Completion Time (CCT). These research efforts greatly accelerate single-stage jobs, i.e. jobs that contain only one coflow, wherein optimizing CCT means optimizing Job Completion Time (JCT).

In many well-known data-parallel cluster computation frameworks, such as Dryad [2] and Hadoop [4], multi-stage jobs are common. Among them, each multi-stage job contains multiple coflows with dependencies, where a coflow cannot start until another coflow is completed, or a coflow cannot be completed until another coflow is completed. We call the former $starts\text{-}after$ dependency, and the latter $finishes\text{-}before$ dependency. In these multi-state job scenarios, minimizing CCT does not mean minimizing JCT and there is a large gap between them [15]. To tackle this problem, several heuristic algorithms [16] and approximation algorithms [15,17] were proposed to optimize JCT recently.

In the above research work focusing on minimizing JCT, the underlying network is treated as a non-blocking big switch, in which a flow in jobs can be transmitted as long as its source and destination are idle. However, in real systems, network resources are limited. Scheduling methods that do not consider network resource constraints may increase both network bottlenecks and flow completion time, while reducing link capacity utilization, thereby increasing JCT.

* Corresponding authors.
E-mail addresses: zengyue@smail.nju.edu.cn (Y. Zeng), yebl@nju.edu.cn (B. Ye), cstb@hhu.edu.cn (B. Tang), stguo@swu.edu.cn (S. Guo), quzhihao@hhu.edu.cn (Z. Qu).

https://doi.org/10.1016/j.comnet.2020.107686
Received 31 July 2020; Received in revised form 20 October 2020; Accepted 15 November 2020
Available online 19 November 2020

In this paper, we take the network resource constraints into account and investigate how to schedule coflows in multi-stage jobs to minimize the total weighted JCT, where important jobs are assigned a higher weight. However, this problem is challenging when multiple jobs compete for network resources due to the following facts: (1) the release times, transmission requests, and importance are quite different among jobs, which makes it difficult to determine which jobs are eligible to be scheduled. (2) each job contains multiple coflows with dependencies, which further makes it difficult to determine which coflows within a job can be scheduled preferentially without violating the coflow dependencies. To clarify the multi-stage job scheduling problem under network resource constraints, we first formulate it as nonlinear programming and show its NP-hardness. Then, we relax it to linear programming, which suggests a priority order for jobs to occupy network resources. Based on this job priority order, we propose a polynomial-time algorithm to schedule coflows in jobs, in which the coflow in the job with higher priority can be prioritized. To ensure a feasible solution, we check coflow dependency constraints and link capacity constraints before scheduling in the algorithm. Finally, we conduct a performance analysis of the algorithm, which shows that it can achieve constant approximation ratios in many typical network scenarios. The main contributions of this paper are as follows.

- To the best of our knowledge, we are the first to study how to schedule multi-stage jobs under network resource constraints, so as to minimize the total weighted JCT.
- We formally model the multi-stage job scheduling problem under network resource constraints and then prove its NP-hardness.
- We propose a polynomial-time algorithm with a performance guarantee, and prove that our algorithm can achieve constant approximation ratios in many typical network scenarios.
- We conduct extensive simulations based on a real-world data trace collected from Facebook. Simulation results show that our algorithm is close to the optimal job priority solution in total weighted JCT. Compared with the state-of-the-art, our algorithm can shorten the average weighted JCT by up to 48.46% and the execution time is 19.12× faster.

The remainder of this paper is organized as follows. We introduce related work in Section 2 and our motivation in Section 3. Section 4 presents the formulation of the multi-stage job scheduling problem with network resource constraints and shows its NP-hardness. Section 5 gives relaxation of the problem and proposes a polynomial algorithm based on the relaxation. The performance of this algorithm is analyzed in Section 6 and evaluated in Section 7. Section 8 concludes this paper.

2. Related work

Existing research work on coflow scheduling mainly focuses on two performance metrics: CCT or JCT. Next, we introduce related work from these two aspects.

CCT: The concept of "coflow" is first officially defined as a network abstraction in [7], which shows great advantages in optimizing communication requirements in data-parallel computing frameworks. As an important performance metric of coflow, CCT has been optimized by many studies [8–14]. [8] introduced an effective heuristic algorithm that greedily schedules coflow based on its bottleneck's completion time to minimize CCT, while meets the coflow deadline. Considering the importance of different coflows, each coflow is assigned a weight to indicate their importance. [9] provided the first deterministic algorithm with a constant approximation ratio of $\frac{67}{3}$ to minimize the total weighted CCT. To improve the approximation ratio, Shafiee et al. relaxed the problem to linear programming, and presented a 12-approximation ratio algorithm in [10], and further proposed a 5-approximation ratio algorithm in [11].

However, the coflow scheduling mechanisms mentioned above abstract the underlying network topology as a non-blocking big switch so that network resource constraints are ignored, which may reduce link capacity utilization and extend JCT. Then, several research efforts consider network resource constraints and joint routing and scheduling to minimize CCT [12–14]. Rapier [12] is the first algorithm to integrate routing and scheduling to minimize CCT. However, its solution to multiple coflows is heuristic which cannot provide a theoretical performance guarantee. For further improvement, Tan et al. [13] presented a rounding-based randomized algorithm for a single coflow and an online algorithm with a performance guarantee for multi-coflow routing and scheduling. Recently, Zhang et al. [14] considered network resource constraints and proposed a distributed bottleneck-aware coflow scheduling algorithm to minimize CCT by reducing network bottlenecks and improving link capacity utilization.

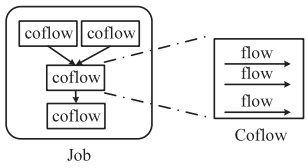


Fig. 1. Multi-stage job structure.

JCT: The above work proposes various methods to optimize CCT or weighted CCT. This is valid for a single-stage job, where each job contains only one coflow. However, in a multi-stage job scenario, there are dependencies among coflows, and minimizing CCT may fail to minimize JCT. Therefore, several research efforts focused on reducing JCT by scheduling methods [15–17]. Aalo [16] is the first effective heuristic algorithm that considers coflow dependencies in multi-stage jobs, and attempts to minimize JCT by scheduling coflows. Further, Tian et al. [15,17] relaxed the multi-stage job scheduling problem to linear programming and provided a deterministic algorithm with an M-approximation ratio, where M represents the number of machines. Unfortunately, these scheduling methods for multi-stage jobs abstract the network into a non-blocking big switch and ignoring the network resource constraints.

Therefore, in this paper, we address the problem of minimizing the total weighted JCT of multi-stage jobs under network resource constraints.

3. Motivation

In this section, we first introduce the structure of multi-stage jobs and then illustrate our motivation with a simple example.

The structure of a multi-stage job is shown in Fig. 1, where a job contains multiple coflows with dependencies, and each coflow contains multiple flows. Coflow dependencies can be divided into two types, namely $starts\text{-}after$ dependency and $finishes\text{-}before$ dependency [16]. The former indicates that one coflow cannot start until another coflow is completed, while the latter indicates that one coflow cannot be completed until another coflow is completed. A job is completed when all coflows in this job are completed. A coflow is completed when all flows in this coflow are completed.

The scheduling method without considering network resources may extend JCT. To demonstrate this, let us consider a simple example in Fig. 2, in which dropping, retransmission, ACKs are abstracted; network scheduling without considering network resources would lead to even worse performance when the effects of these aspects are non-ignorable. As shown in Fig. 2(a), there are two jobs waiting to be scheduled in the network, where job j_1 (j_2) contains only one coflow c_1 (c_2), and coflow c_1 (c_2) contains only one flow f_1 (f_2). Both flows f_1 and f_2 have a unit size. As shown in Fig. 2(b), each link has unit capacity. The default routing paths for flows f_1 and f_2 are $h_1 \rightharpoonup s_1 \rightharpoonup s_2 \rightharpoonup h_2$ and $h_3 \rightharpoonup s_1 \rightharpoonup s_2 \rightharpoonup h_4$, and they share a link (s_1, s_2).

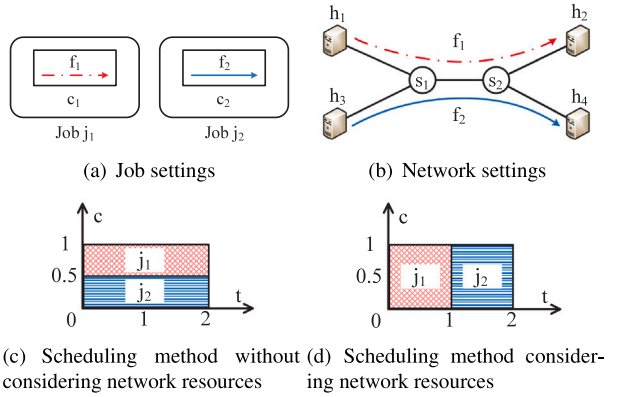


Fig. 2. A simple example to show the advantages of considering network resource constraints.

We first analyze a scheduling method that does not consider network resources as shown in Fig. 2(c). It treats the network as a non-blocking switch, and each flow exclusively consumes the source and destination. Once the source and destination of a flow are idle, the flow is scheduled. Since the source and destination of f_1 and f_2 are both idle, f_1 and f_2 are scheduled and they share the link capacity fairly on link (s_1, s_2). Obviously, the JCT of j_1 and j_2 are 2, respectively, and the total JCT is 4. Next, we analyze a scheduling method that considers network resources as shown in Fig. 2(d), which allows each flow to exclusively occupy links on its routing path. Since the routing paths of f_1 and f_2 share a link (s_1, s_2), they are scheduled in sequence, occupying the entire capacity of the links on its routing path. So the JCT of job j_1 and j_2 are 1 and 2, respectively, and the total JCT is 3.

From the above example, we can see that a scheduling method that considers network resource constraints can improve link capacity utilization, shorten flow completion time, and thus shorten JCT. Additionally, it can also contribute to energy saving by using a sleep mechanism [18–21], due to the shorter job duration. However, scheduling multi-stage jobs in an arbitrary topology is challenging. First, jobs have different release times, transmission requests, and importance, which makes it difficult to determine which jobs are eligible to be scheduled when they compete for network resources. Besides, coflows in jobs have dependencies and may compete for network resources, making it difficult to determine which coflows in jobs can be scheduled preferentially.

4. Formulation

In this section, we first introduce the basic settings, then formulate the Multi-stage Job Scheduling (MJS-NRC) problem with Network Resource Constraints, and finally prove its NP-hardness. The notations to be used are listed in Tables 1 and 2.

4.1. Settings

We consider a network $G = (V, L)$, where V and L indicate the node set and link set respectively. Each link $(u, v) \in L$ in the network has an equal capacity of c, just like a typical data center topology as BCube [22] or Dcell [23], etc.

There are N jobs in the network, denoted by $j_1, \dots, j_N$. Each job j_n contains K_n coflows, denoted by $c_{n,1}, \dots, c_{n,K_n}$. Each coflow $c_{n,k}$ contains $M_{n,k}$ flows, denoted by $f_{n,k,1}, \dots, f_{n,k,M_{n,k}}$. Job information can be obtained in advance by traffic prediction technology. As with [8,9,16,24,25], we assume that the job structure and flow properties are known as priori, such as the source, destination, and bytes of each flow in it. Each flow $f_{n,k,m}$ has a static default routing path $p_{n,k,m}$. As in [15], we focus on network scheduling, so the computation duration is ignored.

Table 1
Notations of constants.

Symbol	Definition
N	The number of jobs
K_n	The number of coflows in job j_n
$M_{n,k}$	The number of flows in coflow $c_{n,k}$
V	The node set
L	The link set
$D_{n,k,k'}$	The binary constant indicates whether there is a dependency: coflow $c_{n,k}$ starts after coflow $c_{n,k'}$
c	The capacity of each link
r_n	The release time of job j_n
w_n	The weight of job j_n
$b_{n,k,m}$	The bytes of flow $f_{n,k,m}$
$p_{n,k,m}$	The default routing path of flow $f_{n,k,m}$
$p^{u,v}_{n,k,m}$	The binary constant indicates whether link (u, v) is on the default routing path of flow $f_{n,k,m}$.

Table 2
Notations of variables.

Symbol	Definition
J_n	The completion time of job j_n
$C_{n,k}$	The completion time of coflow $c_{n,k}$
$F_{n,k,m}$	The completion time of flow $f_{n,k,m}$
$F'_{n,k,m}$	The starting transmission time of flow $f_{n,k,m}$
$f_{n,k,m}(t)$	The traffic size of flow $f_{n,k,m}$ at time t
$f^{u,v}_{n,k,m}(t)$	The traffic size of flow $f_{n,k,m}$ at time t on link (u, v)
$X_{n,k,m}(t)$	The binary variable indicates whether flow $f_{n,k,m}$ is scheduled at time t

4.2. Problem formulation

Considering that important jobs are prioritized by assigning higher weights, each job j_n is assigned with a weight w_n. We try to optimize the overall performance of weighted jobs by minimizing the total weighted JCT

$$\min \sum_{n=1}^{N} w_n J_n$$

where J_n denotes the completion time of job j_n. The objective is subject to the following constraints.

Completion time constraints: As mentioned above, each job j_n contains K_n coflows, and each coflow $c_{n,k}$ contains $M_{n,k}$ flows. The completion time of a job depends on the completion time of the latest coflow that composes it. So we have the following constraints

$$J_n = \max_{k \in \{1,2,\dots,K_n\}} C_{n,k}, \forall n \in \{1, 2, \dots, N\}, \tag{1}$$

where $C_{n,k}$ represents the completion time of coflow $c_{n,k}$.

Similarly, the completion time of a coflow depends on the completion time of the latest flow that composes it. Hence,

$$C_{n,k} = \max_{m \in \{1,2,\dots,M_{n,k}\}} F_{n,k,m}, \quad \forall n \in \{1, 2, \dots, N\}, k \in \{1, \dots, K_n\}, \tag{2}$$

where $F_{n,k,m}$ represents the completion time of flow $f_{n,k,m}$. For ease of reading, $\forall n \in \{1, 2, \dots, N\}, k \in \{1, 2, \dots, K_n\}, m \in \{1, 2, \dots, M_{n,k}\}$ is simplified to $\forall n, k, m$, below.

Start time constraints: Flows in a job can start its transmission only after the job is released. So

$$F'_{n,k,m} \geq r_n, \forall n, k, m, \tag{3}$$

where $F'_{n,k,m}$ denotes the starting transmission time of flow $f_{n,k,m}$ and r_n denotes the release time of job j_n.

In the context of multi-stage jobs, there can be two types of coflow dependencies: *starts-after* and *finishes-before* [16]. As in [15,17], we focus on *starts-after* type multi-stage jobs, which is common in Sawzall [26], Pig [27], Hive [28], etc. *finishes-before* type jobs are

left for future work. In *starts-after* dependency, a coflow cannot start its transmission until the coflow it depends on completes transmission. Naturally, flows in a coflow cannot start their transmission until the coflow it depends on completes transmission. The dependency constraint can be expressed as

$$F'_{n,k,m} \geq D_{n,k,k'} C_{n,k'}, \forall n, k, k', m, \tag{4}$$

where $D_{n,k,k'}$ is a binary indicator which is equal to 1 if there is a dependency that coflow $c_{n,k}$ starts after coflow $c_{n,k'}$ or 0 if otherwise.

Traffic constraints: To improve link capacity utilization and shorten flow completion time [19,21], exclusive routing is adopted [29–31]. Once a flow $f_{n,k,m}$ is scheduled at time t, the flow occupies the full capacity on its routing path. Let $X_{n,k,m}(t)$ be a binary variable indicating whether flow $f_{n,k,m}$ is scheduled at time t. Constraints can be expressed as

$$X_{n,k,m}(t) \in \{0, 1\}, \forall n, k, m, t, \tag{5}$$

$$f_{n,k,m}(t) = c X_{n,k,m}(t), \forall n, k, m, t, \tag{6}$$

$$f^{u,v}_{n,k,m}(t) = f_{n,k,m}(t) p^{u,v}_{n,k,m}, \forall n, k, m, (u, v) \in L, t, \tag{7}$$

where $f_{n,k,m}(t)$ represents the traffic size of flow $f_{n,k,m}$ at time t, $f^{u,v}_{n,k,m}(t)$ denotes the traffic size of flow $f_{n,k,m}$ at time t on link (u, v) and $p^{u,v}_{n,k,m}$ is a binary constant indicating whether link (u, v) is on the default routing path of flow $f_{n,k,m}$.

Flow transmission constraints: When a flow completes its transmission, all its bytes should have been transmitted. Hence, we have

$$\int_{F'_{n,k,m}}^{F_{n,k,m}} f_{n,k,m}(t)\, dt = b_{n,k,m}, \forall n, k, m, \tag{8}$$

where $b_{n,k,m}$ denotes the bytes of flow $f_{n,k,m}$. It should be noted that once a flow $f_{n,k,m}$ starts its transmission, $f_{n,k,m}(t)$ is always equal to 1 until the flow completes its transmission or the network resources on its routing path are preempted by other flows. The flow $f_{n,k,m}$ is preempted by a finite number of flows. Therefore, in the interval $[F'_{n,k,m}, F_{n,k,m}]$, $f_{n,k,m}(t)$ has at most a finite number of discontinuities, and thus $f_{n,k,m}(t)$ is integrable.

Link capacity constraints: The traffic on each link should not exceed the link capacity. The constraint can be expressed as

$$\sum_{n,k,m} f^{u,v}_{n,k,m}(t) \leq c, \forall (u, v) \in L, t. \tag{9}$$

To simplify the constraints, we combine Eq. (6) (7) (9) and (6) (8) to obtain

$$\sum_{n,k,m} X_{n,k,m}(t) p^{u,v}_{n,k,m} \leq 1, \forall (u, v) \in L, t, \tag{10}$$

$$\int_{F'_{n,k,m}}^{F_{n,k,m}} c X_{n,k,m}(t)\, dt = b_{n,k,m}, \forall n, k, m, t. \tag{11}$$

In summary, MJS-NRC problem can be formulated as

$$\min \sum_{n=1}^{N} w_n J_n$$

s.t.(1)(2)(3)(4)(5)(10)(11).

4.3. NP-hardness

The following result shows that it is impossible to solve MJS-NRC optimally in polynomial time unless P = NP.

Theorem 1. *MJS-NRC problem is NP-hard.*

Proof. We will show the result by establishing a reduction from the Single Machine Scheduling (SMS-RTP) problem with a Release time while the Preemption is allowed, to MJS-NRC. SMS-RTP is a classic NP-complete problem [9,32] whose definition is given as follows.

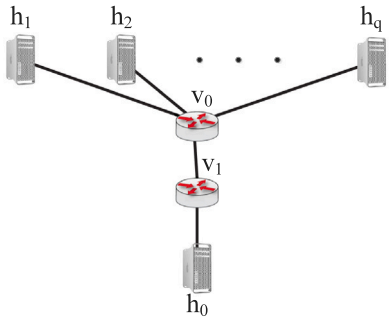


Fig. 3. A special case of MJS-NRC.

SMS-RTP Problem: Given q independent tasks and a single machine, each task needs to be assigned to this machine for processing. Each task has a weight w_i and a release time r_i. Each task can be preempted by other tasks. The goal is to find a scheduling method to minimize the total weighted completion time.

In the following, we will introduce a special case of MJS-NRC, which is equivalent to SMS-RTP. Consider a special case of MJS-NRC as shown in Fig. 3. There are q jobs in the network, each job contains a coflow, and each coflow contains a flow. In other words, each job contains only one flow. Each flow is from h'_i to h_0, $1 \leq i \leq q$. Each link in the network has the same capacity of 1. If we treat the link between v_0 and v_1 as a single machine, and each job as a task, then it is straightforward to see that this special case of MJS-NRC is equivalent to SMS-RTP. Hence, MJS-NRC is NP-hard. The proof is accomplished. □

5. Algorithm design

In this section, we propose MJS, an efficient polynomial-time algorithm for MJS-NRC problem. MJS consists of two main components. One is responsible for assigning priorities to jobs using a heuristic linear programming (LP) relaxation-based approach. Another one is responsible for scheduling coflows in active jobs, i.e., the jobs that have been released but not yet completed, based on priority order. In the following, we will first introduce the two components and then give the whole description of MJS.

5.1. Determining priority order based on LP relaxation

When there are multiple jobs in the network waiting to be scheduled, they may compete for network resources. We need to determine the priority order of jobs to occupy network resources, and jobs with higher priority can prioritize network resources. Here we relax MJS-NRC problem to a linear programming and assign priorities to jobs based on the solution to the linear programming.

In MJS-NRC problem, Eqs. (10) (11) contain an infinite variable $X_{n,k,m}(t)$, and the latter is nonlinear, which makes the optimization problem difficult to solve. In the following, we first relax these constraints to linear constraints with finite variables. By integrating Eq. (10) in $[r_n, J_n]$ and $[0, J_n]$, we obtain the following two formulas

$$J_n - r_n \geq \sum_{n',k,m} \int_{r_n}^{J_n} X_{n',k,m}(t) p^{u,v}_{n',k,m}\, dt, \forall n, (u, v) \in L. \tag{12}$$

$$J_n \geq \sum_{n',k,m} \int_{0}^{J_n} X_{n',k,m}(t) p^{u,v}_{n',k,m}\, dt, \forall n, (u, v) \in L. \tag{13}$$

Based on Eqs. (1) (2) (3) (11) and a series of relaxations, Eq. (12) can be relaxed as

$$\begin{aligned} J_n - r_n &\geq \sum_{k,m} \int_{r_n}^{J_n} X_{n,k,m}(t) p_{n,k,m}^{u,v} \, dt \\ &\geq \sum_{k,m} \int_{F'_{n,k,m}}^{F_{n,k,m}} X_{n,k,m}(t) p_{n,k,m}^{u,v} \, dt \\ &\geq \sum_{k,m} \frac{b_{n,k,m}}{c} p_{n,k,m}^{u,v}, \forall n, (u,v) \in L. \end{aligned} \tag{14}$$

In order to relax Eq. (13) to a linear constraint without losing the order between jobs conveyed therein, we introduce variables $Y_{n',n}$ defined by

$$Y_{n',n} = \begin{cases} 1, \text{if } J_{n'} \leq J_n, \\ 0, \text{otherwise.} \end{cases} \tag{15}$$

In other words, $Y_{n',n}$ is equal to 1 if $n' = n$ or job $j_{n'}$ is completed before job j_n. Then, similar to Eqs. (14), (13) can be relaxed as

$$\begin{aligned} J_n &\geq \sum_{n' \preceq n,k,m} \int_0^{J_n} X_{n',k,m}(t) p_{n',k,m}^{u,v} \, dt \\ &\geq \sum_{n' \preceq n,k,m} \int_{F'_{n,k,m}}^{F_{n,k,m}} X_{n',k,m}(t) p_{n',k,m}^{u,v} \, dt \\ &\geq \sum_{n' \preceq n,k,m} \frac{b_{n',k,m}}{c} p_{n',k,m}^{u,v}, \\ &\geq \sum_{n',k,m} \frac{b_{n',k,m}}{c} p_{n',k,m}^{u,v} Y_{n',n}, \forall n, (u,v) \in L, \end{aligned} \tag{16}$$

where $n' \preceq n$ means that job $j_{n'}$ is completed before job j_n.

Note that the variables $Y_{n',n}$ satisfy the following conditions

$$Y_{n',n} \in \{0,1\}, \forall n', n, n' \neq n, \tag{17}$$

$$Y_{n',n} + Y_{n,n'} = 1, \forall n', n, n' \neq n, \tag{18}$$

$$Y_{n',n} = 1, \forall n' = n, \tag{19}$$

where Eq. (18) represents the asymmetry of the order relationship between jobs. So based on the above relaxations, MJS-NRC can be relaxed to the following Integer Linear Programming (ILP)

$$\text{(ILP) } \min \sum_{n=1}^{N} w_n J_n$$
$$s.t. (14)(16)(17)(18)(19).$$

By allowing $Y_{n',n}$ to take values from the whole [0,1] interval, we now have the following LP relaxation

$$\text{(LP) } \min \sum_{n=1}^{N} w_n J_n$$
$$s.t. (14)(16)(18)(19),$$
$$0 \leq Y_{n',n} \leq 1, \forall n', n, n' \neq n. \tag{20}$$

Since LP is relaxed from MJS-NRC problem, we have

$$OPT_{LP} \leq OPT_{MJS-NRC}, \tag{21}$$

where OPT_{LP} and $OPT_{MJS-NRC}$ represent the optimal solutions of the LP and MJS-NRC problem, respectively.

We can solve the above LP efficiently to get its optimal solution. Let $\tilde{J}_n$ be the corresponding value of J_n in the optimal solution. Based on these $\tilde{J}_n$, we then introduce a priority order of the jobs where a job j_n has a higher priority than $j_{n'}$ if $J_n < J_{n'}$, so that a job with a higher priority is more likely to be completed earlier which is consistent with the optimal solution of the LP relaxation. A formal description of the determination of the priority order is given in Algorithm 1.

Algorithm 1 Determining Priority Order of Jobs

Require: Parameters in LP.
Ensure: Priority order of jobs $\mathcal{P}$.
1: Solve LP and get its optimal solution $\widetilde{J}_n$.
2: Assign each job j_n with a priority $\mathcal{P}_n$ so that

$$\mathcal{P}_n > \mathcal{P}_{n'}, \text{if } \widetilde{J}_n \leq \widetilde{J}_{n'}. \tag{22}$$

3: $\mathcal{P} = \{\mathcal{P}_1, \mathcal{P}_2, ..., \mathcal{P}_N\}$.

5.2. Scheduling active jobs

Algorithm 2 is designed to schedule active jobs based on their priorities. In other words, it is responsible for determining which flows in jobs are eligible to occupy network resources to be scheduled. We first initialize the scheduled flow set to empty and all link states to idle. Then, we schedule active jobs one by one according to their priority order. The flow in a job may be blocked due to coflow dependence or lack of network resources. So, before scheduling a coflow, we need to check whether this coflow depends on other active coflows. A coflow can be scheduled only when this coflow does not depend on any other active coflow. Similarly, before scheduling a flow, we need to check whether all links on the routing path of this flow are idle. A flow can be scheduled only when all links on the flow's routing path are idle. Finally, when a flow is scheduled, we mark all links on its routing path as occupied.

Algorithm 2 Scheduling Active Jobs

Require: S: link status; $P_{n,k,m}$: default routing path of flow $f_{n,k,m}$; $\mathbb{J}$: active job set; $\mathbb{J}_n$: active coflow set; $\mathbb{C}_{n,k}$:active flow set; $f_{n,k,m}$: flow; $\mathcal{P}$: job priority set.
Ensure: $\mathbb{F}_{schedule}$: Scheduled flows.
1: $\mathbb{F}_{schedule} \leftarrow \emptyset$.
2: $S_{u,v} \leftarrow 0, \forall (u,v) \in L$.
3: **for** $j_n \in \mathbb{J}, n = \arg\max_i \mathcal{P}_i$ **do**
4: **for** $c_{n,k} \in \mathbb{J}_n$ **do**
5: **if** $c_{n,k}$ not dependent on $c_{n,k'}, \forall c_{n,k'} \in \mathbb{J}_n$ **then**
6: **for** each flow $f_{n,k,m} \in \mathbb{C}_{n,k}$ **do**
7: **if** $S_{u,v} = 0, \forall u, v \in P_{n,k,m}$ **then**
8: $\mathbb{F}_{schedule} \leftarrow \mathbb{F}_{schedule} + \{f_{n,k,m}\}$.
9: **for** $u, v \in P_{n,k,m}$ **do**
10: $S_{u,v} \leftarrow 1$ // mark the link as occupied
11: **end for**
12: **end if**
13: **end for**
14: **end if**
15: **end for**
16: **end for**

5.3. Whole description of MJS

MJS algorithm is shown in Algorithm 3. We first call Algorithm 1 to solve LP and obtain the priority of all jobs. Then, we repeatedly schedule active jobs until all the jobs are completed. Specifically, when a new job is released or a flow is completed, the network state changes, so we need to determine or re-determine how to schedule active jobs. We first update the currently active job set, coflow set, flow set, and

flows. This is to get information about the job set, coflow set, flow set, and flows that have been released but not yet completed. Then, we call Algorithm 2 to determine which flows in active jobs can be scheduled based on job priority. Finally, we suspend those flows that are not allowed to be scheduled, and schedule those that are allowed to be scheduled.

Algorithm 3 MJS algorithm

Require: L: link set; $P_{n,k,m}$: default routing path of flow $f_{n,k,m}$; $\mathbb{J}$: active job set; $\mathbb{J}_n$: active coflow set; $\mathbb{C}_{n,k}$:active flow set; $\mathbb{F}_{active}$: active flows; $\mathbb{F}_{suspend}$: suspended flows.
Ensure: $\mathbb{F}_{schedule}$: Scheduled flows.
1: $\mathcal{P} \leftarrow$ calling Algorithm 1.
2: **repeat**
3: **if** a job j_n is released or a flow $f_{n,k,m}$ completes transmission **then**
4: Update $\mathbb{J}$, $\mathbb{J}_n$, $\mathbb{C}_{n,k}$, $\mathbb{F}_{active}$, $\forall n, k$.
5: $\mathbb{F}_{schedule} \leftarrow$ calling Algorithm 2.
6: $\mathbb{F}_{suspend} \leftarrow \mathbb{F}_{active} - \mathbb{F}_{schedule}$.
7: Suspend flows in $\mathbb{F}_{suspend}$.
8: Schedule flows in $\mathbb{F}_{schedule}$.
9: **end if**
10: **until** All job are completed

6. Algorithm analysis

In this section, we analyze the performance of MJS algorithm, including feasibility, complexity, and approximation ratio.

6.1. Feasibility analysis

Theorem 2. *The solution obtained by MJS algorithm is always feasible.*

Proof. Here, we need to prove that the solution of MJS algorithm satisfies all constraints in MJS-NRC problem. Constraints (1) (2) (3) (4) (5) (7) (8) describe the basic characteristics of the job, coflow and flow. So, we only need to prove that the solution of MJS algorithm satisfies constraints (6) (9), that is, coflow dependency constraints and link capacity constraints. As described in Algorithm 2 in Section 5.2, a flow can be scheduled only when these two constraints are satisfied. Therefore, the solution of MJS algorithm satisfies all constraints and is always a feasible solution. □

6.2. Complexity analysis

Theorem 3. *MJS algorithm is a polynomial-time algorithm.*

Proof. In MJS algorithm, we first call Algorithm 1 (line 1 in Algorithm 3) to get the priority of jobs, which needs to solve LP. It should be noted that solving LP dominates MJS algorithm. As LP shows, J_n and $Y_{n,n'}$ contain N and N^2 variables, respectively. Eqs. (14), (16), (18), (19) and (20) require $N|L|$, $N|L|$, $N(N-1)$, N and $N(N-1)$ constraints, where $|L|$ indicates the number of links. So, LP contains $N + N^2$ variables and $2N|L| + 2N^2 - N$ constraints. It can be solved in polynomial time by using the ellipsoid algorithm [33] or projective algorithm [34]. Obviously, the remainder of Algorithm 1 (lines 2–3) and Algorithm 2 run in polynomial time. Based on the above analysis, as shown in Algorithm 3, MJS algorithm is a polynomial time algorithm. □

6.3. Approximation ratio

In order to prove the approximation ratio of MJS algorithm, we first define several variables. The effective transmission duration of job j_n on link (u, v) is defined as

$$\Gamma_n^{u,v} = \sum_{k,m} \frac{b_{n,k,m}}{c} p_{n,k,m}^{u,v}, \forall n, (u,v) \in L. \tag{23}$$

The default routing path for flow $f_{n,k,m}$ is $p_{n,k,m}$. P_{max} and P_{min} indicate the maximum and minimum length of the default routing path for all flows, respectively. So, the length of the default routing path for any flow satisfies

$$P_{min} \le \sum_{(u,v)\in p_{n,k,m}} p_{n,k,m}^{u,v} \le P_{max}, \forall n, k, m. \tag{24}$$

Theorem 4. *MJS algorithm is a* $(min\{1 + 2\kappa_{max}P_{max}, 1 + \frac{2l}{P_{min}}\})$*-approximation algorithm, where* l *and* κ_{max} *refer to the number of links in the network and the maximum number of stages among all jobs, respectively.*

Proof. Here, we first prove Lemmas 1, 2, 3 and finally prove Theorem 4 based on these three lemmas.

Lemma 1. *In LP,* $\widetilde{J}_n \ge \frac{1}{2} max_{(u,v)\in L} \sum_{n'\le n} \Gamma_{n'}^{u,v}$.

Proof. Based on Eqs. (23), (16) can be relaxed as

$$J_i \ge \sum_{i'} \Gamma_{i'}^{u,v} Y_{i',i} \ge \sum_{i'\le n} \Gamma_{i'}^{u,v} Y_{i',i}, \tag{25}$$

which indicates

$$\widetilde{J}_i \Gamma_i^{u,v} \ge \sum_{i'\le n} \Gamma_{i'}^{u,v} \Gamma_i^{u,v} Y_{i',i}. \tag{26}$$

We can also get

$$\begin{aligned} \sum_{i\le n} \widetilde{J}_i \Gamma_i^{u,v} &\ge \sum_{i\le n} \sum_{i'\le n} \Gamma_{i'}^{u,v} \Gamma_i^{u,v} Y_{i',i} \\ &= \frac{1}{2} \sum_{i\le n} \sum_{i'\le n} \left(\Gamma_{i'}^{u,v} \Gamma_i^{u,v} (Y_{i',i} + Y_{i,i'}) \right). \end{aligned} \tag{27}$$

According to Eqs. (18) (19), the right side of Eq. (27) can be simplified to

$$\begin{aligned} \sum_{i\le n} \widetilde{J}_i \Gamma_i^{u,v} &\ge \frac{1}{2} \left(\sum_{i\le n} \sum_{i'\le n} \Gamma_{i'}^{u,v} \Gamma_i^{u,v} + \sum_{i\le n} (\Gamma_i^{u,v})^2 \right) \\ &= \frac{1}{2} \left((\sum_{i\le n} \Gamma_i^{u,v})^2 + \sum_{i\le n} (\Gamma_i^{u,v})^2 \right) \\ &\ge \frac{1}{2} (\sum_{i\le n} \Gamma_i^{u,v})^2. \end{aligned} \tag{28}$$

Since $\widetilde{J}_n = \max_{i\le n} \widetilde{J}_i$, we have

$$\begin{aligned} \widetilde{J}_n \sum_{i\le n} \Gamma_i^{u,v} &= \sum_{i\le n} \widetilde{J}_n \Gamma_i^{u,v} \ge \sum_{i\le n} \widetilde{J}_i \Gamma_i^{u,v} \\ &\ge \frac{1}{2} (\sum_{i\le n} \Gamma_i^{u,v})^2. \end{aligned} \tag{29}$$

Thus,

$$\widetilde{J}_n \ge \frac{1}{2} \sum_{i\le n} \Gamma_i^{u,v}, \tag{30}$$

which implies

$$\widetilde{J}_n \ge \frac{1}{2} \max_{(u,v)\in L} \sum_{n'\le n} \Gamma_{n'}^{u,v}. \quad \square \tag{31}$$

Lemma 2. *In MJS algorithm,* $J_n \le (1 + \frac{2l}{P_{min}}) \widetilde{J}_n$.

Proof. In MJS algorithm, jobs are scheduled according to their priority. Job j_n may be blocked by those jobs with higher priority. In the worst case, all jobs with higher priority than job j_n are released after it, and flows in these jobs can only be transmitted one by one, due to network resource constraints or coflow dependency constraints. Then, we have

$$J_n \le r_n + \sum_{n'\le n,k,m} \frac{b_{n',k,m}}{c}, \tag{32}$$

which indicates

$$J_n \le r_n + \sum_{n' \le n,k,m} \frac{\sum_{(u,v)\in p_{n',k,m}} p^{u,v}_{n',k,m} b_{n',k,m}}{\sum_{(u,v)\in p_{n',k,m}} p^{u,v}_{n',k,m} c}. \tag{33}$$

According to Eqs. (23) (24), (33) can be relaxed as

$$\begin{aligned} J_n &\le r_n + \sum_{n' \le n,k,m} \frac{\sum_{(u,v)\in L} p^{u,v}_{n',k,m} b_{n',k,m}}{P_{min} c} \\ &= r_n + \sum_{(u,v)\in L} \sum_{n' \le n,k,m} \frac{p^{u,v}_{n',k,m} b_{n',k,m}}{P_{min} c} \\ &\le r_n + \sum_{(u,v)\in L} \max_{(u,v)\in L} \sum_{n' \le n,k,m} \frac{p^{u,v}_{n',k,m} b_{n',k,m}}{P_{min} c} \\ &= r_n + \frac{l}{P_{min}} \max_{(u,v)\in L} \sum_{n' \le n,k,m} \frac{p^{u,v}_{n',k,m} b_{n',k,m}}{c} \\ &= r_n + \frac{l}{P_{min}} \max_{(u,v)\in L} \sum_{n' \le n} \Gamma^{u,v}_{n'}, \end{aligned} \tag{34}$$

where l and P_{min} indicate the number of links in the network and the shortest default path length.

Based on Lemma 1, Eq. (34) can be relaxed as

$$\begin{aligned} J_n &\le r_n + \frac{2l}{P_{min}} \widetilde{J}_n \le \widetilde{J}_n + \frac{2l}{P_{min}} \widetilde{J}_n \\ &\le (1 + \frac{2l}{P_{min}}) \widetilde{J}_n. \end{aligned} \tag{35}$$

□

Lemma 3. *In MJS algorithm,* $J_n \le (1 + 2\kappa_{max} P_{max}) \widetilde{J}_n$.

Proof. In a multi-stage job scenario, a job may contain multiple coflow dependency sequences, where the latter coflow depends on the previous one. Suppose $\{c_{n,k_1}, c_{n,k_2}, \dots, c_{n,k_{\kappa_n}}\}$ is the coflow dependency sequence with the latest completion time in job j_n, that is, $J_n = C_{n,k_{\kappa_n}}$. $f_{n,k_1,m^*}, f_{n,k_2,m^*}, \dots, f_{n,k_{\kappa_n},m^*}$ indicate the latest flows among coflow $c_{n,k_1}, c_{n,k_2}, \dots, c_{n,k_{\kappa_n}}$, respectively. Obviously, the latest flow in job j_n is f_{n,k_{κ_n},m^*}, that is, $J_n = F_{n,k_{\kappa_n},m^*}$. The routing paths of these flows are $p_{n,k_1,m^*}, p_{n,k_2,m^*}, \dots, p_{n,k_{\kappa_n},m^*}$, respectively. The set of links covered by these routing paths is denoted as $\mathbb{P}$, we have $\mathbb{P} = \{p_{n,k_1,m^*} \bigcup p_{n,k_2,m^*} \bigcup \dots \bigcup p_{n,k_{\kappa_n},m^*}\}$. Due to coflow dependency, f_{n,k_{κ_n},m^*} needs to wait for $f_{n,k_1,m^*}, f_{n,k_2,m^*}, \dots, f_{n,k_{\kappa_n-1},m^*}$ to complete their transmission. Flow f_{n,k_{κ_n},m^*} may be delayed by the traffic on the routing path of these flows. The total traffic on $\mathbb{P}$ is $\sum_{(u,v)\in\mathbb{P}} \sum_{n' \le n,k,m} b_{n',k,m} p^{u,v}_{n',k,m}$. Before flow f_{n,k_{κ_n},m^*} completes transmission, at least one link on $\mathbb{P}$ is transmitting traffic. So we have

$$\begin{aligned} J_n &= F_{n,k_{\kappa_n},m^*} \\ &\le r_n + \frac{\sum_{(u,v)\in\mathbb{P}} \sum_{n' \le n,k,m} b_{n',k,m} p^{u,v}_{n',k,m}}{c} \\ &\le r_n + \sum_{(u,v)\in\mathbb{P}} \max_{(u,v)\in\mathbb{P}} \sum_{n' \le n,k,m} \frac{b_{n',k,m}}{c} p^{u,v}_{n',k,m}. \end{aligned} \tag{36}$$

According to the definition, the length of each routing path does not exceed P_{max}, and the number of stages of each job does not exceed κ_{max}. Therefore, the number of links in $\mathbb{P}$ does not exceed $\kappa_{max} P_{max}$. Based on the above analysis, Lemma 1 and Eqs. (23), (36) can be relaxed as

$$\begin{aligned} J_n &\le r_n + \kappa_{max} P_{max} \max_{(u,v)\in\mathbb{P}} \sum_{n' \le n,k,m} \frac{b_{n',k,m}}{c} p^{u,v}_{n',k,m} \\ &\le r_n + \kappa_{max} P_{max} \max_{(u,v)\in L} \sum_{n' \le n,k,m} \frac{b_{n',k,m}}{c} p^{u,v}_{n',k,m} \\ &\le r_n + \kappa_{max} P_{max} \max_{(u,v)\in L} \sum_{n' \le n} \Gamma^{u,v}_{n'} \\ &\le \widetilde{J}_n + 2\kappa_{max} P_{max} \widetilde{J}_n \\ &= (1 + 2\kappa_{max} P_{max}) \widetilde{J}_n. \end{aligned} \tag{37}$$

□

Table 3
Approximation ratios in several typical scenarios.

Topology	κ_{max}	P_{max}	Approximation ratio
BCube	1	4	9
Dcell	1	5	11
Three-tier architecture	1	6	13

Based on Lemmas 2 and 3, we have

$$J_n \le (\min\{1 + 2\kappa_{max} P_{max}, 1 + \frac{2l}{P_{min}}\}) \widetilde{J}_n. \tag{38}$$

Thus, the solution of MJS algorithm satisfies

$$\begin{aligned} SOL_{MJS} &= \sum_{n=1}^{N} w_n J_n \\ &\le \min\{1 + 2\kappa_{max} P_{max}, 1 + \frac{2l}{P_{min}}\} \sum_{n=1}^{N} w_n \widetilde{J}_n. \end{aligned} \tag{39}$$

According to Eqs. (21) (39), the approximation ratio of MJS algorithm satisfies the following

$$\begin{aligned} \alpha &= \frac{SOL_{MJS}}{OPT_{MJS-NRC}} \le \frac{SOL_{MJS}}{OPT_{LP}} = \frac{\sum_{n=1}^{N} w_n J_n}{\sum_{n=1}^{N} w_n \widetilde{J}_n} \\ &\le \min\{1 + 2\kappa_{max} P_{max}, 1 + \frac{2l}{P_{min}}\}. \end{aligned} \tag{40}$$

□

6.4. Performance in typical scenarios

Here, we prove that MJS algorithm obtains constant approximation ratios in several typical scenarios and a special scenario.

Theorem 5. *MJS algorithm obtains constant approximation ratios in several typical scenarios, where each job has only one stage, and each flow uses its shortest path as the default routing path, and the topology is BCube, or Dcell, or a three-tier architecture.*

Proof. In data-parallel computing frameworks, single-stage jobs are common. For these jobs, we have $\kappa_{max} = 1$.

In data centers, BCube, Dcell and three-tier architecture (Fat-Tree [35] or VL2 [36]) are typical topologies. The shortest path is often used as the default routing path for flows, because it requires less network resources and has a lower propagation delay. In BCube and Dcell, the shortest path between any two nodes does not exceed 4 and 5 hops respectively. In the three-tier architecture, the shortest path between any two nodes does not exceed 6 hops. Therefore, we have $\kappa_{max} = 4$, $\kappa_{max} = 5$, $\kappa_{max} = 6$, corresponding to BCube, Dcell and three-tier architecture.

According to Theorem 4 and the above analysis, we obtain the approximate ratios of these three typical topologies as 9, 11, and 13, respectively, as shown in Table 3. This completes the proof. □

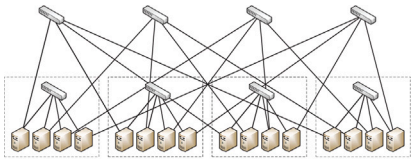
Theorem 6. *MJS algorithm is a* 3*-approximation algorithm when network resources are strictly restricted so that the default paths of all flows are intersect on a link.*

Proof. When the default paths of all flows intersect on a link (u', v'), we have $p^{u',v'}_{n,k,m} = 1$, $\forall n, k, m$. Based on this and Eq. (23), Lemma 1 can be relaxed as

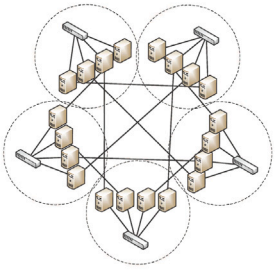
$$\begin{aligned} \widetilde{J}_n &\ge \frac{1}{2} \sum_{n' \le n} \Gamma^{u',v'}_{n'} \ge \frac{1}{2} \sum_{n' \le n,k,m} \frac{b_{n',k,m}}{c} p^{u',v'}_{n',k,m} \\ &= \frac{1}{2} \sum_{n' \le n,k,m} \frac{b_{n',k,m}}{c}. \end{aligned} \tag{41}$$

Combining Eqs. (32) (41), we get

$$J_n \le r_n + 2\widetilde{J}_n \le 3\widetilde{J}_n. \tag{42}$$



(a) BCube



(b) Dcell

Fig. 4. Two typical topologies.

Similar to Eq. (40), we obtain the approximation ratio of MJS algorithm

$$\alpha \le \frac{\sum_{n=1}^{N} w_n J_n}{\sum_{n=1}^{N} w_n \widetilde{J}_n} \le 3. \quad \square \tag{43}$$

7. Simulation and result

In this section, we first introduce the simulation settings. Then, we provide simulation results and detailed performance analysis.

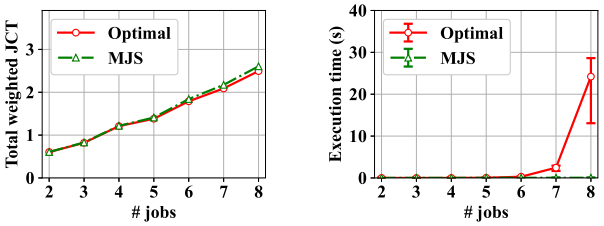
7.1. Simulation settings

Operating environment: Software configuration: our algorithm is implemented on the Ubuntu 19.04 operating system [37], the programming language is Python 3.7 [38]. Hardware configuration: Intel(R) Core(TM) i7-8700U CPU, 3.19 GHz, 8G RAM.

Topology: Servers are connected by two typical data center topologies, such as BCube [22] and Dcell [23], as shown in Fig. 4. The capacity of each link is set to 10 Gbps, which is the same as [8,16]. Each flow uses the shortest path as the default routing path. The network is generated by networkx [39].

Workload: Our workload is generated based on a Facebook trace [40], which is collected from a 3000 machine, 150-rack MapReduce cluster at Facebook. This trace is widely used for simulation [8,15,16,41]. It gives information about each coflow, such as sender machines, receiver machines, and transmitting bytes. However, the trace contains only information for each coflow, not the job. Therefore, we randomly select coflows to form the jobs, and the number of coflows contained in each job is subject to a discrete uniform distribution $DU[1,\alpha]$. The Map node and the Reduce node are also randomly selected, and the number of them is also subject to discrete uniform distribution $DU[1,\beta]$ and $DU[1,\gamma]$. We randomly selected δ machines from the trace as servers. The dependency among coflows in a job is randomly generated. The weight of each job obeys a discrete uniform distribution $DU[1,\theta]$, and the release time of each job obeys a Poisson distribution $P(\lambda)$ (in milliseconds).

Metrics: Here, we evaluate several metrics of MJS algorithm, including the total weighted JCT, average weighted JCT, and JCT cumulative distribution. Each data points is obtained by averaging 50 runs. Moreover, we evaluate the algorithm execution time, which includes the time to solve LP. The data points are collected from 50 runs.



(a) Total weighted JCT (b) Execution time

Fig. 5. Performance gap between MJS algorithm and OJP solution.

Benchmark algorithm: We compare MJS algorithm with the following algorithms:

- OJP (Optimal Job Priority) solution: The optimal job priority solution is obtained by exploring all possible job priority sequences.
- MCS algorithm: MCS algorithm schedules each coflow based on the optimal solution of a linear programming, which is the first approximation algorithm that can be applied to multi-stage job scheduling scenarios.

7.2. Compared with the optimal job priority solution

The job priority in MJS algorithm is obtained by solving linear programming. In order to evaluate the gap between our MJS algorithm and OJP solution, we compare their performance in terms of total weighted JCT and execution time. Here, we test our algorithm in BCube network and set $\alpha = 1$, $\beta = 1$, $\gamma = 1$, $\delta = 16$, $\theta = 10$, $\lambda = 0$. The simulation results are shown in Fig. 5.

As shown in Fig. 5(a), the total weighted JCT of our algorithm is close to OJP solution. In the worst case, the gap between our algorithm and OJP solution is only 4.30% (8 jobs). Besides, the total weighted JCT increases as the number of jobs increases. This is because more jobs compete for network resources and more bits need to be transmitted. As shown in Fig. 5(b), the average execution time of our algorithm is significantly shorter than OJP solution. For example, when there are 8 jobs, the execution time of our algorithm distributed from 47.67 ms to 77.61 ms, the average value is only 51.16 ms, and the standard deviation is 6.07 ms. In contrast, the execution time of MCS algorithm ranges from 13.07 s to 28.62 s, its average value exceeds 24.21 s, and the standard deviation is 2.97 s. This is because OJP solution is obtained by exploring 40320 (8!) possible job priority sequences, which is time-consuming.

Thus, we can conclude that MJS algorithm is close to OJP solution in terms of the total weighted JCT and has a shorter execution time.

7.3. Compared with MCS algorithm

In this subsection, we compare our algorithm with MCS algorithm in terms of average weighted JCT, execution time and JCT distribution in BCube ($\delta = 16$) and Dcell ($\delta = 20$) networks. We set $\alpha = 3$, $\beta = 2$, $\gamma = 2$, $\theta = 10$, $\lambda = 1000$.

7.3.1. Average weighted JCT performance

We evaluate the average weighted JCT performance of MJS algorithm and MCS algorithm in BCube and Dcell networks. The simulation results are shown in Fig. 6.

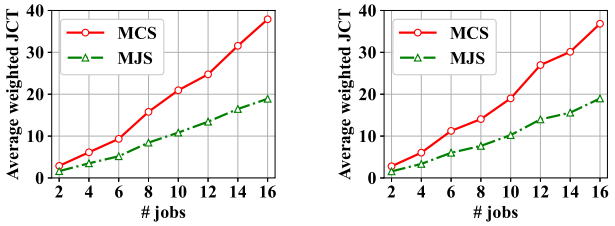


Fig. 6. Average weighted JCT of MJS algorithm and MCS algorithm in two typical topologies.

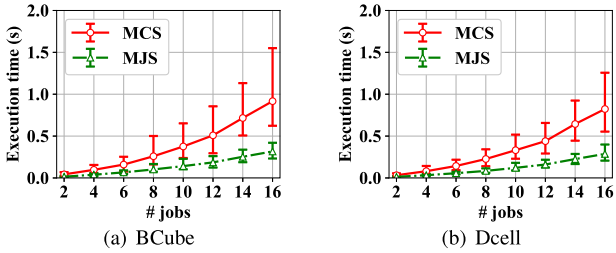


Fig. 7. Execution time of MJS algorithm and MCS algorithm in two typical topologies.

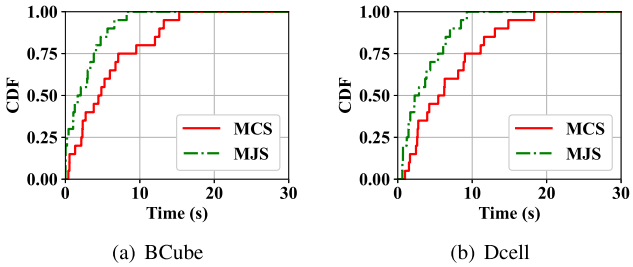


Fig. 8. JCT cumulative distribution of MJS algorithm and MCS algorithm in two typical topologies.

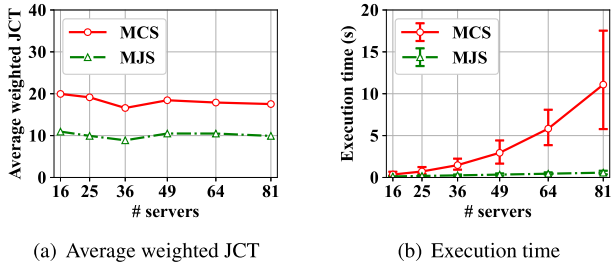


Fig. 9. Performance comparison of MJS algorithm and MCS algorithm under various network scales.

As shown in Fig. 6(a), the average weighted JCT of our algorithm is significantly shorter than that of MCS algorithm in BCube network. This is mainly because the network resource constraints are considered in our algorithm, which avoids sharing links, improves link capacity utilization, shortens the flow completion time, and further shortens the JCT of each job. Besides, as the number of jobs increases, more jobs need to compete for network resources, and they need to wait longer to obtain network resources, so the average weighted JCT of both algorithms is extended with the increase of jobs, as shown in Fig. 6(a). Also, as the number of jobs increases, the gap between MCS algorithm and MJS algorithm gradually widens. This is because the increased jobs exacerbate the competition of network resources, and the advantage of our algorithm is more obvious. On the other hand, similar results can be found in Fig. 6(b). In Dcell network, the average weighted JCT of our algorithm is always significantly shorter than MCS algorithm. For example, when there are 16 jobs, our algorithm can reduce the average weighted JCT of MCS algorithm by 48.46%.

7.3.2. Execution time

We evaluate the execution time performance of MJS algorithm and MCS algorithm in BCube and Dcell networks. The simulation results are shown in Fig. 7.

As shown in Fig. 7(a), the average execution time of our algorithm is significantly shorter than that of MCS algorithm. When there are 16 jobs, our algorithm only takes 0.32 s on average to schedule all jobs, while MCS algorithm takes 0.92 s. That is, our algorithm is 2.88× faster than MCS algorithm. This is mainly because the LP in our algorithm contains fewer variables than the LP in MCS algorithm. Besides, the standard deviations of the execution time of these two algorithms are 0.11 and 0.31 respectively, which shows that our algorithm is more stable. Moreover, as the number of jobs increases, the execution time of both algorithms increases. This can be attributed to the increase in the number of jobs leading to more constraints and variables in LP, and scheduling more jobs requires more time. In Dcell network, the execution time of the two algorithms is slightly shorter. For example, when there are 16 jobs, the average execution time of MJS algorithm and MCS algorithm are 0.29 s and 0.82 s, respectively. Because in Dcell network, there are fewer switches and links. In this case, the average execution time of our algorithm is 2.83× faster than MCS algorithm.

7.3.3. JCT distribution

To further explore the JCT distribution of each job, we evaluate the JCT distribution of MJS algorithm and MCS algorithm in BCube network. The number of jobs is set to 20. The simulation results are shown in Fig. 8.

As shown in Fig. 8(a), the JCT of our algorithm is significantly shorter than MCS algorithm. For example, all the jobs scheduled by MJS algorithm complete transmission within 10 s, however, only 80% of the jobs scheduled by MCS algorithm are completed. This is because each flow in jobs scheduled by our algorithm exclusively occupies all links on its routing path, which improves link capacity utilization and reduces network bottlenecks, thereby reducing the completion time of each flow, and further reduces the JCT of each job. Similar results can be found in Fig. 8(b), which further demonstrates that the advantages of our algorithm can be attributed to taking network resource constraints into account.

7.3.4. Different network sizes

To investigate the performance of our algorithms at different network sizes, we tested our algorithms in different sizes of BCube networks. The number of servers in the network varies from 16 to 81, and the number of jobs is set to 20. The simulation results are shown in Fig. 9.

As shown in Fig. 9(a), MJS algorithm is always superior to MCS algorithm in terms of average weighted JCT, regardless of network size. Also, the average weighted JCT of MJS algorithm and MCS algorithm is gradually reduced as the size of the network topology expands. This is mainly because expanding the network size increases the network resources, which relieves the pressure of jobs to compete for network resources, so that each job only waits for a short time. As shown in Fig. 9(b), the average execution time of MJS algorithm is significantly shorter than MCS algorithm. When there are 81 servers, MCS algorithm needs 11.09 s to schedule all jobs, and MJS algorithm only needs 0.58 s, which means that MJS algorithm is 19.12× faster than MCS algorithm. This is because the LP in MCS algorithm contains far more variables

than that of MJS algorithm, and it takes longer to solve. Besides, the execution time of both algorithms increases as the network size expands. Because solving LP in large-scale networks is more complicated, and more link states need to be considered in the two algorithms.

Therefore, we can conclude that our algorithm is always superior to MCS algorithm in average weighted JCT, JCT, and execution time, under different network topologies, network sizes and the number of jobs.

8. Conclusion

In this paper, we study how to improve the delay performance of multi-stage jobs under network resource constraints by scheduling methods. Specifically, we first formulated the multi-stage job scheduling problem with network resource constraints and proved its NP-hardness. Then, we relax the problem to a linear programming. Inspired by the linear programming solution, we proposed MJS algorithm with an approximation ratio of $(\min\{M, \frac{l}{P}\}+1)$, where M, l and P denote the total number of flows, the number of links and the length of the shortest default path, respectively, and demonstrated that the approximation ratio is a fixed constant in many typical scenarios. Finally, we conducted extensive trace-driven simulations, which demonstrate the superior performance of MJS in terms of both weighted JCT and execution time. The results verify that the scheduling method considering network resources can indeed effectively improve the delay performance of multi-stage jobs.

CRediT authorship contribution statement

Yue Zeng: Conceptualization, Methodology, Formal analysis, Software, Writing - original draft. **Baoliu Ye:** Supervision, Writing - review & editing, Project administration. **Bin Tang:** Supervision, Methodology, Writing - original draft. **Songtao Guo:** Validation, Writing - original draft. **Zhihao Qu:** Validation, Writing - original draft.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

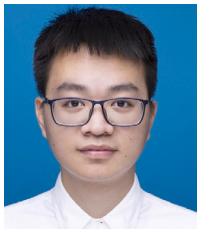
Acknowledgments

This work was supported in part by National Key R&D Program of China (Grant No. 2018YFB1004704), in part by the National Natural Science Foundation of China (Grant Nos. 61832005 and 61872171), in part by the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20190058), in part by the China Postdoctoral Science Foundation (Grant No. 2019M661709), and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization, China.

References

[1] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, Oper. Syst. Rev. 41 (3) (2007) 59–72.
[3] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M.J. Franklin, S. Shenker, I. StoicaStoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: Proceedings of NSDI, 2012, pp. 15–28.
[4] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Proceedings of IEEE MSST, 2010, pp. 1–10.
[5] M. Chowdhury, M. Zaharia, J. Ma, M.I. Jordan, I. Stoica, Managing data transfers in computer clusters with orchestra, in: Proceedings of ACM SIGCOMM, 2011, pp. 98—109.
[6] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, A. Vahdat, Sincronia: near-optimal network design for coflows, in: Proceedings of ACM SIGCOMM, 2018, pp. 16–29.
[7] M. Chowdhury, I. Stoica, Coflow: A networking abstraction for cluster applications, in: Proceedings of ACM HotNets, 2012, pp. 31–36.
[8] M. Chowdhury, Y. Zhong, I. Stoica, Efficient coflow scheduling with Varys, in: Proceedings of ACM SIGCOMM, 2014, pp. 443—454.
[9] Q. Zhen, C. Stein, Z. Yuan, Minimizing the total weighted completion time of coflows in datacenter networks, in: Proceedings of ACM SPAA, 2015, pp. 294–303.
[10] M. Shafiee, J. Ghaderi, Scheduling coflows in datacenter networks: Improved Bound for Total Weighted Completion Time, in: Proceedings of ACM SIGMETRICS, 2017, pp. 29—30.
[11] M. Shafiee, J. Ghaderi, An improved bound for minimizing the total weighted completion time of coflows in datacenters, IEEE/ACM Trans. Netw. 26 (4) (2018) 1674–1687.
[12] Y. Zhao, K. Chen, W. Bai, M. Yu, C. Tian, Y. Geng, Y. Zhang, D. Li, S. Wang, Rapier: Integrating routing and scheduling for coflow-aware data center networks, in: Proceedings of IEEE INFOCOM, 2015, pp. 424–432.
[13] H. Tan, S.H.-C. Jiang, Y. Li, X.-Y. Li, C. Zhang, Z. Han, F.C.M. Lau, Joint online coflow routing and scheduling in data center networks, IEEE/ACM Trans. Netw. 27 (5) (2019) 1771–1786.
[14] T. Zhang, R. Shu, Z. Shan, F. Ren, Distributed bottleneck-aware coflow scheduling in data centers, IEEE Trans. Parallel Distrib. Syst. 30 (7) (2018) 1565–1579.
[15] B. Tian, C. Tian, H. Dai, B. Wang, Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time, in: Proceedings of IEEE INFOCOM, 2018, pp. 864–872.
[16] M. Chowdhury, I. Stoica, Efficient coflow scheduling without prior knowledge, in: Proceedings of ACM SIGCOMM, 2015, pp. 393—406.
[17] B. Tian, C. Tian, B. Wang, B. Li, Z. He, H. Dai, K. Liu, W. Dou, G. Chen, Scheduling dependent coflows to minimize the total weighted job completion time in datacenters, Comput. Netw. 158 (2019) 193–205.
[18] C. Canali, L. Chiaraviglio, R. Lancellotti, M. Shojafar, Joint minimization of the energy costs from computing, data transmission, and migrations in cloud data centers, IEEE Trans. Green Commun. Netw. 2 (2) (2018) 580–595.
[19] D. Li, Y. Shang, W. He, C. Chen, EXR: Greening data center network with software defined exclusive routing, IEEE Trans. Comput. 64 (9) (2015) 2534–2544.
[20] H. Wang, Y. Li, D. Jin, P. Hui, J. Wu, Saving energy in partially deployed software defined networks, IEEE Trans. Comput. 65 (5) (2016) 1578–1592.
[21] Y. Zeng, S. Guo, G. Liu, Comprehensive link sharing avoidance and switch aggregation for software-defined data center networks, Future Gener. Comput. Syst. 91 (2019) 25–36.
[22] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, S. Lu, BCube: A high performance, server-centric network architecture for modular data centers, in: Proceedings of ACM SIGCOMM, 2009, pp. 63—74.
[23] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, S. Lu, Dcell: A scalable and fault-tolerant network structure for data centers, in: Proceedings of ACM SIGCOMM, 2008, pp. 75—86.
[24] S. Khuller, M. Purohit, Brief announcement: Improved approximation algorithms for scheduling co-flows, in: Proceedings of ACM SPAA, 2016, pp. 239–240.
[25] M. Shafiee, J. Ghaderi, Brief announcement: A new improved bound for coflow scheduling, in: Proceedings of ACM SPAA, 2017, pp. 91–93.
[26] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, Interpreting the data: Parallel analysis with Sawzall, Sci. Program. 13 (4) (2005) 277–298.
[27] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of ACM SIGMOD, 2008, pp. 1099–1110.
[28] Apache Hive, 2019, http://hadoop.apache.org/hive.
[29] H. Wang, X. Yu, H. Xu, J. Fan, C. Qiao, L. Huang, Integrating coflow and circuit scheduling for optical networks, IEEE Trans. Parallel Distrib. Syst. 30 (6) (2019) 1346–1358.
[30] H. Jahanjou, E. Kantor, R. Rajaraman, Asymptotically optimal approximation algorithms for coflow scheduling, in: Proceedings of ACM SPAA, 2017, pp. 45–54.
[31] S. Luo, H. Yu, Y. Zhao, S. Wang, S. Yu, L. Li, Towards practical and near-optimal coflow scheduling for data center networks, IEEE Trans. Parallel Distrib. Syst. 27 (11) (2016) 3366–3380.
[32] J.K. Lenstra, A.R. Kan, P. Brucker, Complexity of machine scheduling problems, in: Annals of Discrete Mathematics, vol. 1, Elsevier, 1977, pp. 343–362.
[33] L.G. Khachiyan, A polynomial algorithm in linear programming, in: Dokl. Akad. Nauk SSSR, 244, (5) 1979, pp. 1093–1096.
[34] N. Karmarkar, A new polynomial-time algorithm for linear programming, in: Proceedings of ACM STOC, 1984, pp. 302–311.

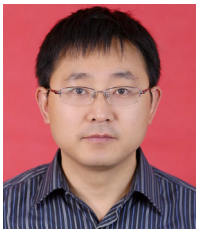
[35] M. Al-Fares, A. Loukissas, A. Vahdat, A scalable, commodity data center network architecture, in: Proceedings of ACM SIGCOMM, 2008, pp. 63—74.
[36] A. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, VL2: a scalable and flexible data center network, in: Proceedings of ACM SIGCOMM, 2009, pp. 51–62.
[37] Ubuntu 19.04, 2019, http://releases.ubuntu.com/19.04/.
[38] Python 3.7.2, 2019, https://www.python.org/ftp/python/3.7.2/.
[39] Networkx, 2019, https://pypi.python.org/pypi/networkx/.
[40] FaceBookTrace, 2019, https://github.com/coflow/coflow-benchmark.
[41] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, Y. Geng, CODA: Toward automatically identifying and scheduling coflows in the dark, in: Proceedings of ACM SIGCOMM, 2016, pp. 160–173.



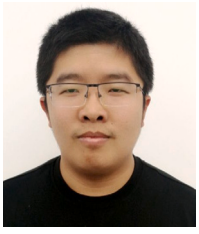
Yue Zeng received the M.S. degree in the department of electronic information engineering from Southwest University, Chongqing, China, in 2019. He is currently working toward the Ph.D. degree in the department of computer science and technology in Nanjing University, China. His research interests include flow scheduling in data center networks, software defined networking and edge computing.

Baoliu Ye is a full professor at Department of Computer Science and Technology, Nanjing University. He received his Ph.D. in computer science from Nanjing University, China in 2004. He served as a visiting researcher of the University of Aizu, Japan from March 2005 to July 2006, and the Dean of School of Computer and Information, Hohai University since January 2018. His current research interests mainly include distributed systems, cloud computing, wireless networks with over 70 papers published in major conferences and journals. Prof. Ye served as the TPC co-chair of HotPOST12, Hot-POST11, P2PNet10. He is the regent of CCF, the Secretary-General of CCF Technical Committee of Distributed Computing and Systems, and a member of IEEE.

Bin Tang received his B.S. and Ph.D. degree in computer science from Nanjing University, Nanjing, China, in 2007, and 2014, respectively. He was an assistant researcher at Nanjing University from 2014 to 2020, and also a research fellow at The Hong Kong Polytechnic University in 2019. He is currently a professor at Hohai University. His research interests lie in the area of communications, network coding, and distributed computing. He is a member of IEEE and ACM.



Songtao Guo received the BS, MS, and PhD degrees in computer software and theory from Chongqing University, Chongqing, China, in 1999, 2003, and 2008, respectively. He is currently a full professor at Chongqing University, China. His research interests include wireless networks, mobile cloud computing and parallel and distributed computing. He has published more than 100 scientific papers in leading refereed journals and conferences. He has received many research grants as a principal investigator from the National Science Foundation of China and Chongqing and the Postdoctoral Science Foundation of China. He is an IEEE/ACM Senior member.



Zhihao Qu received his B.S. and Ph.D. degree in computer science from Nanjing University, Nanjing, China, in 2009, and 2018, respectively. He is currently an assistant researcher in the College of Computer and Information at Hohai University. His research interests are mainly in the areas of wireless networks, edge computing, and distributed machine learning.