Efficient Online Scheduling for Coflow-aware Machine Learning Clusters

Wenxin Li, Sheng Chen, Keqiu Li, *Senior Member, IEEE*, Heng Qi, Renhai Xu, and Song Zhang

Abstract—Distributed machine learning (DML) is an increasingly important workload. In a DML job, each communication phase can comprise a *coflow*, and there are dependencies among its coflows. Thus, efficient coflow scheduling becomes critical for DML jobs. However, the majority of existing solutions focus on scheduling single-stage coflows with no dependencies. While there are a few studies schedule dependent coflows of multi-stage jobs, they suffer from either practical or theoretical issues. Motivated by this situation, we study how to schedule dependent coflows of multiple DML jobs to minimize the total JCT in a shared cluster. We present a formal mathematical formulation for this problem and prove its NP-hardness. To solve this problem without job size information, we present an online coflow-aware optimization framework called *Parrot*. The core idea in *Parrot* is to infer the job with the shortest remaining processing time (SRPT) each time and dynamically control the inferred job's bandwidth based on how confident it is an SRPT job while being mindful of not starving any other job. Specifically, in the design of *Parrot*, we present a least per-coflow attained service (LPCAS) policy to infer the SRPT job. We further propose a dynamic job weight assignment mechanism and a linear program (LP) based weighted bandwidth scaling strategy for sharing bandwidth among DML jobs. We have proved that *Parrot* algorithm has a non-trivial competitive ratio. The results from large-scale trace-driven simulations further demonstrate that our *Parrot* can reduce the total JCT by up to 58.4%, compared to the state-of-the-art Aalo solution.

Index Terms—Distributed Machine Learning; Coflow Scheduling; Multi-Stage Job; Dependent Coflows

1 Introduction

Recently, machine learning (ML) has shown a remarkable success in not only the computing industry but also the fields such as health care and education, and is deriving many key products [1–4]. To train large models on increasingly large data sets with acceptable training time, distributed ML training has become a standard practice. Therefore, IT giants such as Google, Microsoft and Facebook have begun to use large clusters consisting of hundreds to thousands of servers to run distributed ML (DML) jobs [3–5].

Intuitively, DML jobs have been generally considered to be computation-intensive. Fortunately, the GPU speed has been increased by $35\times$ over the last few years [6], and many other hardware accelerators are getting faster either [7]. Such fast GPUs and accelerators can have high computational throughput and can process more data batches per time unit, leading to more data flow transfers in the network [8]. These flow transfers can account for a significant portion (as high as 90%) of the total job training time [8–10], even in high-speed networks. More precisely, as revealed by [8], when training the VGG19-22K model on a 16-server cluster with 40GbE Ethernet and one Titan X GPU per server, the parameter updates will bottleneck the network; it may even be slower than single machine when increasing the number of servers to a sufficiently large value (i.e., 32). So, the performance bottleneck of a DML job is witnessed to be shifted from computation to communication. There are some recent proposals (e.g., [8, 11–13]) that make efforts to improve DML communication performance. Nevertheless, they are only applicable to single job scenario and are insufficient to improve job performance in a shared cluster with hundreds to thousands of DML jobs running simultaneously [14, 15]. *How to share the network efficiently among multiple DML jobs remains an open research topic.*

Coflow scheduling can help for such network sharing, as DML jobs typically will generate coflows (see Section 2 for details). However, the conventional wisdom of scheduling coflows to optimize coflow completion time (CCT) (e.g., [16–25]) does not necessarily lead to shorter job completion time (JCT). The crux is that DML jobs are inherently multi-stage, and their coflows have dependencies. For instance, in an iteration of a DML job, the coflow in the parameter pull phase depends on that in the parameter push phase. Meanwhile, the push-phase coflow, in turn, depends on the pull-phase coflow of the previous iteration. Such dependency relationship represents that a coflow cannot start until its dependent

This work was supported partly by the National Key R&D Program of China under Grant 2019YFB2102404; the NSFC under Grants 61772112; the Science Innovation Foundation of Dalian under Grant 2019J12GX037. (Corresponding author: Keqiu Li)

- *W. Li is with the Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong 999077. Email: toliwenxin@gmail.com.*
- *S. Chen, K. Li, R. Xu and S. Zhang are with the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, No 135, Yaguan Road, Tianjin 300350, China. E-mail: {chensheng, keqiu, xurenhai, zhang_song}@tju.edu.cn.*
- *H. Qi is with the School of Computer Science and Technology, Dalian University of Technology, No 2, Linggong Road, Dalian 116023, China. E-mail: hengqi@dlut.edu.cn.*

one has finished.

Unfortunately, very little work has been done to consider such dependent coflows:

- Aalo [20] is the state-of-the-art, yet the only heuristic solution. It treats all coflows in the same job as a single entity and uses the least attained service strategy to perform inter-entity scheduling. Coflows in each entity are prioritized based on their dependency order. Nonetheless, Aalo cannot provide an upper bound on the total JCT of a given set of DML jobs.
- There is also one theoretical solution [26] aiming at scheduling dependent coflows to minimize total weighted JCT. Whereas, it needs to solve a linear program (LP), which is a large-scale yet complicated problem with vast jobs in large networks. Besides, it cannot be used in online cases as all job statistics are required to be known in advance to solve the relevant LP.

In this paper, we focus on the problem of scheduling dependent coflows of DML jobs to minimize the total JCT. To solve this problem, we present an online coflow-aware scheduler *Parrot*. The core idea in *Parrot* is to infer the job with the shortest remaining processing time (SRPT) each time and dynamically increase the bandwidth of the inferred job based on how confident it is an SRPT job while being mindful of starvation-free.

For inferring the SRPT job, *Parrot* leverages a *least per-coflow attained service (LPCAS)* heuristic, which seamlessly combines the information of bytes sent and number of completed coflows for a job. Such information is readily available, and more information will make the inferred job more confident.

For considering the starvation-free need and the inevitable mis-inference, *Parrot* strives to prevent the inferred SRPT job from monopolizing the network. More specifically, it decides an occupancy ratio of link capacity that the inferred job can use. This ratio increases with the growth of the amount of inference information. To obey this ratio in competition, *Parrot* assigns the inferred SRPT job a well-designed weight and makes all other job weight to be 1. With these weights, *Parrot* rescales the individual flow bandwidths of each active coflow in each job from a LP-based single-coflow optimization. This LP is small in scale and has an analytic solution. Since each job has a weight of at least one at any time, *Parrot* will not starve any job for an arbitrarily long period.

We have conducted rigorous theoretical analysis to prove that *Parrot* has a non-trivial competitive ratio in minimizing the total JCT for any given set of DML jobs. We have also conducted large-scale simulations based on a realistic workload from Microsoft [27] to demonstrate that *Parrot* can reduce the total JCT by up to 58.4%, compared to the state-of-the-art Aalo solution.

In summary, the main highlights of this paper include:

- We study the problem of scheduling the dependent coflows of multiple DML jobs to minimize the total JCT in machine learning clusters. We develop the mathematical model and present a formal formulation for this problem. We also prove that this problem is NP-hard.
- We present a novel online coflow-aware scheduler, *Parrot*, to solve the problem above. In *Parrot*, we propose a LPCAS heuristic for inferring the SRPT job. We also develop a dynamic job weight assignment as well as a LP-based weighted bandwidth scaling mechanism to share network capacity among all concurrent jobs.
- We conduct rigorous theoretical analysis to demonstrate that *Parrot* can guarantee an upper bound of the total JCT for any given set of DML jobs. We conduct extensive trace-driven simulations to evaluate the performance of *Parrot*, in terms of reducing total JCT.

The rest of this paper is organized as follows. In Section 2, we show some background, describe our problem and present the key ideas for this paper. In Section 3, we develop the mathematical model and present our problem formulation. We show an overview of *Parrot* in Section 4. We show the design details of *Parrot* in Section 5. The simulation details and results are presented in Section 6. We discuss current limitations of *Parrot* and relevant future research in Section 7. Section 8 discusses the related work and Section 9 concludes this paper.

2 Background, Problem Statement and Key Ideas

2.1 Distributed Machine Learning

An ML algorithm is to iteratively optimize its model (usually a set of parameters) until it converges to describe or interpret the input data [28]. Since the input data is usually enormous, processing all input data on a single machine can suffer from significant slowdowns. Hence, distributed ML becomes the most common strategy to speed up data processing. Meanwhile, the most favored paradigm for DML is *data parallelism* [12, 28–33], where the input data is distributed among multiple worker machines. Each machine will then work on its data and periodically communicate with each other to synchronize the parameter updates from other machines.

To efficiently synchronize and manage model parameters between worker machines, the *parameter server* (PS) [12] architecture has been widely adopted in many ML systems including TensorFlow [30], Caffe [34] and MXNet [35]. Fig. 1 shows an overview of such parameter server architecture. In this architecture, each worker stores a replica of the global ML model and the training data is partitioned among all the workers. The ML training process carries out in an iterative fashion, and each iteration contains four phases: *(1)*, each worker trains independently on its own data to decide what changes should be made to get closer to the optimal model parameters. *(2)*, each worker pushes its updates to the relevant PS. *(3)*, the PSs aggregate the updates from all workers, and apply

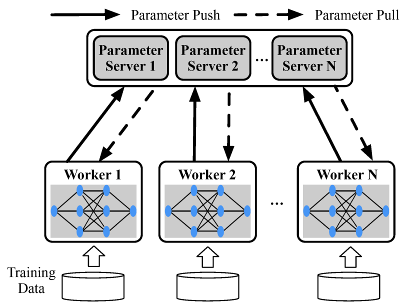


Fig. 1. An overview of the parameter server architecture.

them to the parameters. *(4)*, the updated parameters will be pulled back by the workers, with which they can start the next iteration.

2.2 Dependent Coflows in DML

The parameter exchange phases described above (i.e., push phase (2) and pull phase (4)) typically follow the BSP model, where there is a strict barrier at the end of each phase. For instance, the phase (3) cannot start until the phase (2) finishes, i.e., all the workers have successfully pushed their updates to the PSs; Or, in a certain iteration, the workers cannot proceed to phase (1) before the phase (4) in the previous iteration finishes. It should be noted that there are indeed other synchronization models, i.e., stale synchronous parallel (SSP) [36] and total synchronous parallel (TAP) [37]. However, BSP is the most commonly used one in production [14, 30].

With explicit barriers, the flows in each parameter exchange phase can comprise a *coflow* semantically [38]. Each coflow cannot be considered to be completed until all its flow transfers have finished. Given that a DML job typically has numerous iterations, and each iteration contains two parameter exchange phases, multiple coflows will be generated. Further, the coflows belonging to the same DML job have dependencies. For example, the coflow in the phase (4) depends on that in the phase (2). In the presence of the explicit barrier in BSP, there exists only one type of coflow dependencies—*Starts-After*[1], as defined in the following:

Definition 1 (Starts-After Dependency). *If a coflow $C1$ depends on another one $C2$, i.e., $C2 \mapsto C1$, then $C1$ cannot start until $C2$ has finished.*

Definition 2 (Chain Job). *Considering the iterative nature, each DML job can be viewed as a chain, where each coflow has only one (or zero if it is the first coflow) preface coflow and one subsequent coflow. In other words, multiple coflows will not depend on the same one, and a coflow will not simultaneously depend on more than one coflows.*

2.3 Problem Statement

It has been widely accepted in literature [17, 20–23, 25] that the cluster network can be abstracted as a non-blocking switch interconnecting all machines, given the recent advances in full bisection bandwidth topologies [39, 40]. As shown in Fig. 2(a), each ingress port of the switch receives data from the outgoing link of the connected machine, while each egress port pushes data to the incoming link of the connected machine.

Given multiple DML jobs that contain numerous dependent coflows, we study the problem of *how to assign the bandwidth on the outgoing/incoming links of each physical machine to each flow in each coflow at each time, so as to minimize the total JCT of DML jobs*. However, this problem is inherently challenging due to: 1) it is NP-hard, which we will show in Section 3; 2) jobs can dynamically arrive, with the job size information being unknown.

2.4 The Design Rationales

We now present a walk through the design rationales:

Inferring the SRPT job is a must. SRPT is the most commonly used scheduling policy, which schedules flow with the feast bytes to transmit. It has been shown to provide near-optimal average flow completion time [41–43]. Taking a step further, researchers have generalized SRPT to the case of coflows and have also achieved near-optimal performance [17, 25]. We believe that SRPT can also be generalized to the case of DML jobs to reduce the average JCT. The reason is that one can view each DML job as a "super-coflow" that consists of a chain of "micro-coflows". One key step for applying SRPT is to find the job with the smallest amount of time remaining until completion each time when a coflow completes, or a new coflow from a new job is added.

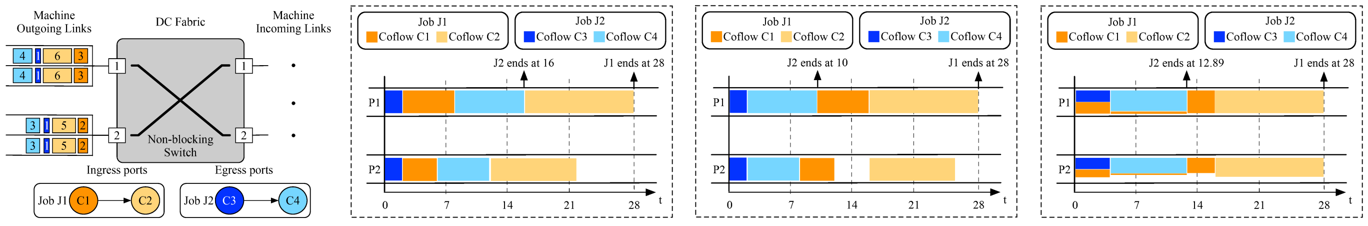
Bytes sent and coflows completed are useful information. Finding the SRPT job relies on accurate job size information. While the size of each coflow can be known once it arrives, it is hard to obtain the job size because of the unknown number of iterations (or coflows) caused by non-smooth loss curves and non-deterministic termination in practice [27]. Fortunately, we can readily acquire the already known information for each job, such as bytes sent and coflows completed. With this information, we infer an SRPT job by seamlessly combing the following two rules: 1) the job with the smallest bytes sent is more likely to have the least remaining data, which is in line with the basic assumption in LAS (Least Attained Service) [44]; 2) the more coflows have been completed, the fewer coflows will remain for a job.

More information leads to higher confidence for job inference. In the initial state, no information can be used to infer the SRPT job, and we, therefore, have no idea about which job has the shortest remaining progress. As time goes by, the bytes sent and coflows completed from all the jobs increase, we will gain more information and thus will have higher confidence to say the inferred job indeed has the shortest remaining time.

The inferred job must not monopolize link bandwidth. It is unavoidable in practice to misidentify a job as the SRPT one. Such errors may negatively impact the total JCT if the misidentified job exclusively occupies the link bandwidth. To mitigate such impact, we should never

1. In contrary to *Starts-After*, *Starts-Before* is another type of coflow dependencies, representing that a coflow cannot finish until its dependent one has finished. However, *Starts-Before* is common for pipeline jobs rather than DML jobs, and hence is not the focus of this paper.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TCC.2020.3040312, IEEE Transactions on Cloud Computing



(a) Example settings (b) A scheme with minimum CCT (c) A scheme with minimum JCT (d) Our approach

Fig. 2. Allocation of ingress port capacities using different schemes for the jobs in Fig. 2(a). (b) When using a scheme that purely optimizes CCT, the incurred total JCT will be 44. (c) The optimal scheme towards minimizing JCT will incur a total JCT of 38, which is hard to be obtained in practice. (d) On the premise that job size is unknown a prior, our approach can make the average JCT to be 40.89.

allow the inferred SRPT job to monopolize the network. Meanwhile, doing this is also for avoiding starvation.

It depends on how confident it is an SRPT job. In the initial state, we treat all jobs fairly and assign the bandwidth at competing links evenly among them. When more information can be used, the inferred job will gain more confidence, and we believe it deserves more bandwidth. So, we increase the bandwidth for the inferred SRPT job as the amount of inference information grows, but will not exceed a predefined upper bound to avoid it monopolize the network.

2.5 A Motivating Example

For a better intuition of our problem and the design rationales, we use a motivating example in Fig. 2(a), where there are 2 simultaneously arrived jobs (i.e., J1 and J2) and 2 machines. Job J1 has 2 coflows C1 and C2 with the dependency that C2 cannot start until C1 has finished. J2 also has 2 coflows (i.e., C3 and C4) and C4 cannot start until the completion of C3. C1 has 4 flows transferring 3/3/2/2 units of data; C2 has 4 flows with sizes of 6/6/5/5; C3 has 4 flows with each having 1 unit of data; C4 has 4 flows having sizes of 4/4/3/3. The virtual input queues at the ingress ports are used for convenience to illustrate the sources and destinations. For example, at the first virtual queue of machine 1, a flow of C1 transfers 3 units of data to machine 2. Each ingress/outgoing port can accept one unit of data each time.

As shown in Fig. 2(b), a scheme that minimizes CCT, i.e., Varys [17], will prefer the shortest coflow each time and execute the 4 coflows in the order of (C3, C1, C4, C2). The resulting total CCT is $2+8+16+28=54$, while the corresponding total JCT is $16+28=44$. As shown in Fig. 2(c), if we schedule coflows in the order of (C3, C4, C1, C2), the total CCT will be increased to $2+10+16+28=56$, while the total JCT can be reduced to $10+28=38$. The result in Fig. 2(c) looks great, which, however, is hard to be achieved as it relies on accurate job size information. By contrast, our approach considers unknown job size but uses the already known information such as bytes sent and coflows completed. Also, our approach assumes the coflow size is known once it is valid for scheduling. As shown in Fig. 2(d), at $t=0$, no job-level information is available and the two jobs (J1 and J2) have the same weight (e.g., 1). Then, when the two jobs meet on the same link, our approach uses these weights to scale the flow bandwidths of their coflows from corresponding LP-based solutions. For example, the LP solution for C1 allocates 1 unit of bandwidth to its flow on P1 and 2/3 to the flow on P2. When sharing with C3, C1 will then get $1 * \frac{1}{1+1} = 0.5$ units of bandwidth on P1 and $\frac{2}{3} * \frac{1}{1+1} = \frac{1}{3}$ on P2. Then, at $t=4$, J2 will be inferred as the SRPT job because its bytes sent is the same with J1 while it has one completed coflow. Hence, J2 will be assigned more bandwidth than J1. In this example, we consider an SRPT job cannot use up to 90% of the capacity on bottleneck link. In this case, J2 and J1 can transfer 0.9 and 0.1 units of data, respectively, on P1, until $t=12.89$ at which J2 completes. After that, J1 monopolizes the network until completion. As a result, the total JCT is $12.89+28=40.89$. This result can practically be obtained yet has a significantly lower JCT than that in Fig. 2(b).

3 Modeling and Problem Formulation

In this section, we develop the mathematical model to study the problem of scheduling dependent coflows of DML jobs to minimize the total JCT.

3.1 Notations

We consider an ML cluster with a set of servers denoted by $\mathcal{N} = \{1, 2, \ldots, N\}$. We denote U_i^E and U_i^I as the outgoing and incoming link capacities for server $i \in \mathcal{N}$, respectively. The cluster is shared by a set of DML jobs denoted as $\mathcal{J} = \{1, 2, \ldots, M\}$, with the job $m \in \mathcal{J}$ arriving at time τ_m. We consider that the parameter servers and workers are already fixed for each job, and multiple coflows will be submitted as iteratively training keeps going. We denote the set of coflows from all jobs as $\mathcal{C}$, with the set of coflows in job m being $\mathcal{C}_m$. We use $k' \prec k$ $(k, k' \in \mathcal{C}_m)$ to represent that the coflow k cannot start until the completion of k', i.e., k depends on k'. Each coflow has multiple flows that are used for the parameter exchange between the servers. We consider that the coflows in different iterations of a job may not necessarily have the same size. The reason is that the workers may update only part of the parameters in each iteration due to the training policy such as dropout [45] or some gradient filters like [11]. So, without loss of generality, we represent the flow volume from server i to j in coflow k of job m as $V_{m,k,i,j}$; the corresponding flow

TABLE 1
Important notations used throughout this paper.

Symbols	Definition
$\mathcal{N}$	the set of servers in the ML cluster
U_i^E	the egress link bandwidth capacity of server $i \in \mathcal{N}$
U_i^I	the ingress link bandwidth capacity of server $i \in \mathcal{N}$
$\mathcal{J}$	the set of DML jobs
τ_m	the release time of job $m \in \mathcal{J}$
$\mathcal{C}$	the set of coflows from all jobs in $\mathcal{J}$
$\mathcal{C}_m$	the set of coflows belonging to the job $m \in \mathcal{J}$
$f_{m,k,i,j}$	the flow from i to j in coflow $k \in \mathcal{C}_m$
$V_{m,k,i,j}$	the data volume of flow $f_{m,k,i,j}$
$b_{m,k,i,j}(t)$	the amount of bandwidth allocated to $f_{m,k,i,j}$ at t
T_m	the job completion time of $m \in \mathcal{J}$
$T_{m,k}$	the coflow completion time of $k \in \mathcal{C}_m$
$\Gamma_{m,k}$	the effective coflow completion time of $k \in \mathcal{C}_m$

is denoted as $f_{m,k,i,j}$. We denote T_m as the JCT of job m and use $T_{m,k}$ to represent the CCT of coflow $k \in \mathcal{C}_m$. Important notations used throughout this paper are listed in Table 1. It should be noted that we focus on the network scheduling for DML jobs, thus the computation time during job training is ignored in the mathematical analysis.

3.2 Mathematical Model

Scheduling decisions: To indicate the decision variable, we denote $b_{m,k,i,j}(t)$ as the amount of bandwidth allocated to coflow k in job m for supporting its data transmission between server i and j at time t. It is a positive value, as shown in the following:

$$b_{m,k,i,j}(t) \geq 0, \forall m \in \mathcal{J}, \forall k \in \mathcal{C}_m, \forall i,j \in \mathcal{N}, \forall t. \quad (1)$$

Guaranteeing the completion of data transmission: All flows must finish their data transmissions between their release time and the completion time of their parent job. Thus, we have the following two constraints:

$$\int_{\tau_m}^{T_m} b_{m,k,i,j}(t)dt = V_{m,k,i,j}, \forall m \in \mathcal{J}, \forall k \in \mathcal{C}_m, \forall i,j \in \mathcal{N} \quad (2)$$

$$\int_{0}^{\tau_m} b_{m,k,i,j}(t)dt = 0, \forall m \in \mathcal{J}, \forall k \in \mathcal{C}_m, \forall i,j \in \mathcal{N} \quad (3)$$

Eq. (2) enforces that each flow $f_{m,k,i,j}$ should be transmitted within $[\tau_m, T_m]$, while Eq. (3) means that each flow cannot transmit any data until the job it belongs to has been released.

Coflow dependency constraints: The coflows belonging to the same DML job has dependency that a coflow cannot start until its dependent coflow has finished. This can be translated into the following constraint:

$$\int_{\tau_m}^{T_{m,k'}} b_{m,k,i,j}(t)dt = 0, \forall m \in \mathcal{J}, \forall k' \prec k \in \mathcal{C}_m, \forall i,j \in \mathcal{N} \quad (4)$$

This constraint essentially means that if the coflow $k \in \mathcal{C}_m$ cannot start any data transmission before all of its dependent coflows have finished.

Capacity constraints: When scheduling coflows, both the ingress and egress link capacities of each server must be satisfied. Thus, we have

$$\sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \sum_{j\in\mathcal{N}} b_{m,k,i,j}(t) \leq U_i^E, \forall i \in \mathcal{N}, \forall t \quad (5)$$

$$\sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \sum_{i\in\mathcal{N}} b_{m,k,i,j}(t) \leq U_j^I, \forall j \in \mathcal{N}, \forall t \quad (6)$$

Eq. (5) indicates that the total amount of data transmitted on the outgoing link of server i must not exceed the link capacity U_i^E at any time t, while, similarly, Eq. (6) specifies that each incoming link transmits at most U_i^I amount of data each time.

3.3 Problem Formulation

Before presenting the problem formulation, we give two definitions that will be used in the rest of this paper.

Definition 3 (Valid Coflow). *A coflow is valid if it has arrived at the network, and it has no dependent coflows, or all of its dependent cofows have finished.*

Definition 4 (Effective CCT). *The effective CCT of a coflow is defined as the time between it being valid for scheduling and being completed.*

Given definitions above, we denote $\Gamma_{m,k}$ as the effective CCT of the coflow k in job m (i.e., $k \in \mathcal{C}_m$). Recall that each DML job can be viewed as a chain of dependent coflows. As such, the JCT of a job can be computed by adding up its arrival time and the total effective CCTs of its all coflows

$$T_m = \tau_m + \sum_{k\in\mathcal{C}_m} \Gamma_{m,k}, \forall m \in \mathcal{J} \quad (7)$$

Similarly, the CCT of coflow $k \in \mathcal{C}_m$ can be calculated by adding up its arrival time, the total effective CCTs of its preface coflows, and its own effective CCT

$$T_{m,k} = \tau_m + \Gamma_{m,k} + \sum_{k'\prec k\in c_m} \Gamma_{m,k'}, \forall m \in \mathcal{J}, \forall k \in \mathcal{C}_m \quad (8)$$

We now formulate the problem of scheduling the dependent coflows of multiple DML jobs to minimize the total JCT, as shown in the following formulation **O1**:

$$\min_{b_{m,k,i,j}(t)} \sum_{m\in\mathcal{J}} T_m \quad (9)$$

Subject to: Eqs. (1)(2)(3)(4)(5)(6)

The problem **O1** is inherently hard to be resolved, due to the following two challenges:

First, it is NP-hard, as shown in the following theorem.

Theorem 1. *The problem **O1** is NP-hard.*

Proof: As the problem of minimizing the total CCT of coflows, denoted **P**, has been proved to be NP-hard in [17]. Therefore, we will prove this theorem by reducing **O1** to **P**. Specifically, given any instance of **P**, we construct a corresponding instance of **O1** as follows: consider a problem **P** with K coflows where the k-th coflow's arrival time is a_k. We construct K jobs with each job having one coflow only. Each job's arrival time is just that of the coflow in it. Since the given instance of **P** is NP-hard, the constructed instance of **O1** is NP-hard as well. □

Second, solving **O1** relies on job arrival information and job size information, neither of which, is available in practice. Hence, an online solution is called for.

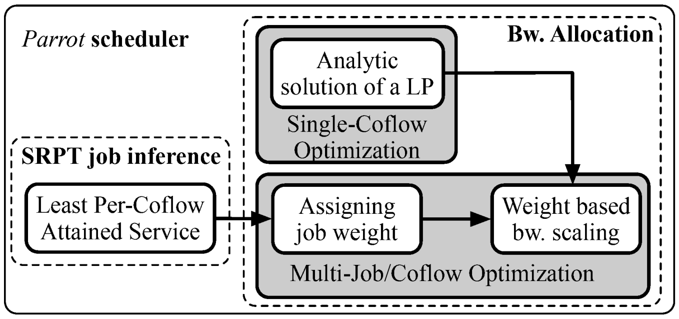

Parrot scheduler
Bw. Allocation
Analytic solution of a LP
Single-Coflow Optimization
SRPT job inference
Least Per-Coflow Attained Service
Assigning job weight
Weight based bw. scaling
Multi-Job/Coflow Optimization

Fig. 3. The overview of *Parrot* optimization framework.

4 Design Overview of *Parrot*

In response to the challenges of solving the original optimization **O1**, we present an efficient online coflow-aware scheduler called *Parrot*. In this section, we first list the realistic assumptions and desirable properties of *Parrot*. Then, we show an overview of *Parrot*.

4.1 Desirable properties

Parrot schedules the dependent coflows of DML jobs under the following realistic assumptions and constraints:

- **Online job arrival**: The DML jobs are submitted in an online fashion. For each job, the placements of PSs and workers are given but unknown prior to its arrival. *Parrot* only deals with the scheduling of the intermediate coflows for submitted jobs.
- **Unknown job size information:** While we assume that the coflow size information can be known once a coflow becomes valid for scheduling, the job size information is typically unknown because the number of iterations for a job is unknown.
- **Only one active coflow per job:** At any time, for any given DML job, there is only one coflow that is active to transmit data in the network, since the DML job is a chain of dependent coflows.

Parrot has the following three goals:

- **Practicality**: *Parrot* is necessarily an online coflow-aware optimization framework. Each time when observing a scheduling event, i.e., a new job arrives or an existing job has finished a coflow, *Parrot* must quickly decide the coflow scheduling decision. Therefore, the *Parrot* algorithms must run in real-time with low time complexity.
- **Upper-bound guarantee**: *Parrot*'s algorithms must be able to provide a non-trivial competitive ratio when solving the problem **O1** for any given set of DML jobs, such that the total JCT can be guaranteed with an upper bound.
- **Starvation-free & Work-conserving:** *Parrot* must not starve any job for an arbitrarily long period, meaning that there is barely any sign of waiting in any job. In addition, we require *Parrot* to be work-conserving to fully utilize link capacity and to minimize JCT.

Our *Parrot* is suitable to traditional distributed training scenarios [12] where the communication and computation perform sequentially and each job can only have one active coflow at a time. On the other hand, we note that the known WFBP [8] (wait-free backward propagation) is supported in existing ML frameworks (e.g., TensorFlow, PyTorch), which may allow some communication with computation. However, the communication stall may still exist because existing ML frameworks (e.g., TensorFlow, PyTorch) typically adopt a global barrier between adjacent iterations. In such a case, we can view the flows in each iteration as a coflow and accordingly our *Parrot* can still work.

4.2 *Parrot* in a nutshell

Parrot makes a scheduling decision whenever an existing coflow completes, or a coflow becomes valid. Fig. 3 presents an overview of our *Parrot* scheduler. At a high-level, *Parrot* infers the SRPT job first, and then performs bandwidth allocation among all concurrent jobs including the inferred job.

SRPT job inference: At the heart of the job inference component is a least per-coflow attained service heuristic, which integrates the per-job information of bytes sent and the number of coflows completed. With this heuristic, each time the job with least per-coflow attained service will be inferred as the SRPT one.

Bandwidth allocation: It first decides each job's weight and then allocates link bandwidth among concurrent jobs according to these weights. To compute job weights, *Parrot* first determines the ratio of network capacity that the inferred SRPT job can occupy when it encounters competitors, based on how much information can be used in job inference component. To obey this occupancy ratio, it then assigns all non-SRPT jobs a weight of 1 and carefully computes the weight for the inferred SRPT job. Rather than sharing network bandwidth among concurrent jobs directly according to computed job weights, we formulate a relevant LP to minimize the effective CCT for each active coflow first. Then, we scale coflow bandwidth in its LP solution according to its parent job weight. Scaling bandwidth from the LP makes a chance to provide a theoretical guarantee for minimizing the total JCT. Note that though a LP is formulated, it is small in scale[2], and we do not need to solve it explicitly, because it has an analytic solution[3].

5 Algorithm Design

5.1 Inferring the SRPT Job

When multiple jobs coexist in the network, *Parrot* seeks to infer which job could have the shortest remaining processing time, such that we can allocate it relatively

2. As compared to the LP in [26] that minimizes the total weighted JCT of all jobs with each job having multiple dependent coflows, our LP is per-coflow problem that minimizes the CCT of single coflow.

3. There is no need to use a solver (e.g., MOSEK) or design an algorithm to derive the LP solution. We can calculate its solution through Eqs. (26)(27) directly.

more bandwidth to minimize the JCT. To infer the SRPT job, we leverage the already known per-job information, i.e., bytes sent and coflows completed. As centralized architecture has shown a great success in many large-scale infrastructure developments [21, 46], we consider that there is a central controller in the cluster to gather such per-job information from all end-hosts, and our *Parrot* scheduler can actually run within this controller.

To ease the presentation, we denote S_m and Z_m as the bytes sent and the number of coflows completed till now for job m, respectively. To mimic the SRPT, we propose a *Least Per-coflow Attained Service (LPCAS)* heuristic, which relies on the following definition:

Definition 5 (Per-coflow Attained Service (PAS)). *We denote A_m as the per-coflow attained service of job m, which is calculated as its bytes sent divided by the number of completed coflows, i.e., $A_m = S_m/Z_m$.*

With the definition above, we infer $m = \arg\min_m A_m$ as the SRPT job. We can easily check that selecting the job with least per-coflow attained service matches the design rationales mentioned in Section 2. First, the smaller the bytes sent S_m for a job m, the less the per-coflow attained service and hence the higher probability m will have to be the SRPT job. Second, the job having more completed coflows (i.e., Z_m is larger) has less per-coflow attained service, and will be more likely to be identified to remain fewer unfinished coflows. Note that PAS may not directly reflect the remaining processing time of a job. However, it could be a good indicator for inferring which job is more likely to be the SRPT one, especially when DML job duration exhibits a heavy-tailed distribution [27]. And, the idea of using already attained service to mimic SRPT has also been widely adopted in many information-agnostic scenarios, e.g., [47], [27], [20].

5.2 Allocating Bandwidth among Jobs

After the SRPT job has been inferred, the next step is to allocate bandwidth to it as well as to other remaining jobs. To this end, we assign each job a weight and then take advantage of a LP-based single-coflow solution to scale the flow bandwidths of each active coflow based on its parent job's weight.

5.2.1 Dynamic job weight assignment

The inferred job may not turn out to be the SRPT one. Thus, it should not fully occupy the network. Meanwhile, we should give relatively more bandwidth to the inferred job if we have enough confidence to say it is indeed an SRPT job. Therefore, we are motivated to assign each job a weight. The job weights are dynamically changed, such that the amount of bandwidth allocated to the inferred job can be increased based on how confident it is an SRPT job. We consider that the more the total bytes have been sent by all jobs, the higher confidence the inferred job will have. Details are as follows:

We first determine an occupancy ratio of the network that the inferred SRPT job can use. We denote $\mathcal{J}_\Omega$ as the set of concurrent jobs. Define the total bytes have been sent by all job in $\mathcal{J}_\Omega$ as $S_{\mathcal{J}_\Omega}$. Then, we can calculate the occupancy ratio for the inferred SRPT job as

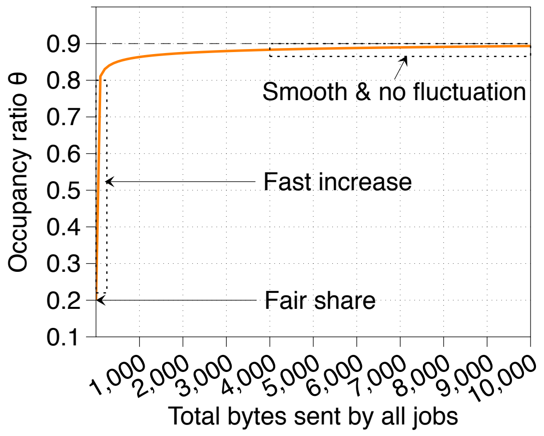


Fig. 4. An example of the occupancy ratio θ when $\theta_{max} = 0.9$ and there are 5 concurrent jobs (i.e., $|\mathcal{J}_\Omega| = 5$).

$$\theta \leftarrow \min\{\theta_{max}, (\frac{1}{|\mathcal{J}_\Omega|})^{1/(\log(S_{\mathcal{J}_\Omega}+1)+1)}\} \tag{10}$$

where θ_{max} is the maximum occupancy ratio. Fig. 4 shows an example of θ under varying total bytes sent, with θ_{max} and $|\mathcal{J}_\Omega|$ being 0.9 and 5 respectively. We can observe that the dynamics of θ have the following properties: (i) When there exists no information to infer the SRPT job, i.e., $S_{\mathcal{J}_\Omega} = 0$, we have $\theta = 1/|\mathcal{J}_\Omega| = 0.2$, which means that the inferred SRPT job will fairly share the link bandwidth with others. (ii) Once we have some information to infer which job might be the SRPT one, we should allow it to quickly occupy the majority of network capacity to minimize the JCT. (iii) Finally, the occupancy rate θ can be maintained at the maximum value θ_{max} with no fluctuation.

Given θ, we now assign each job a weight. Specifically, we set the weight for the inferred SRPT job to

$$W_{max} = \frac{\theta}{1-\theta}(|\mathcal{J}_\Omega| - 1) \tag{11}$$

Meanwhile, we let all other jobs in $\mathcal{J}_\Omega$ to have the same weight of $W_{min} = 1$. As such, when the jobs in $\mathcal{J}_\Omega$ competes a bottleneck link with weighted fair sharing in use, the ratio of the link bandwidth that the inferred job can occupy is exactly $\frac{\frac{\theta}{1-\theta}(|\mathcal{J}_\Omega|-1)}{\frac{\theta}{1-\theta}(|\mathcal{J}_\Omega|-1)+(|\mathcal{J}_\Omega|-1)*1} = \theta$.

5.2.2 LP-based weighted bandwidth scaling

Intuitively, after job weights are determined, a straightforward way for network sharing is to assign bandwidth in proportional to each job's weight. However, it may be far away from the optimum. Hence, we seek to minimize the effective CCT of each active coflow and then scale its flow bandwidths based on the weight of its parent job.

Single coflow optimization: Whenever a coflow becomes valid, we will formulate a relevant LP for it by assuming it monopolizes the network. The details for the LP are shown in the following problem **O2**:

$$\min_{\{b_{m,k,i,j}(t)\}} \Gamma_{m,k} \tag{12}$$

$$\text{s.t: } \int_0^{\Gamma_{m,k}} b_{m,k,i,j}(t)dt = V_{m,k,i,j}, \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \tag{13}$$

$$\sum_{j\in\mathcal{N}} b_{m,k,i,j}(t) \le U_i^E, \forall i \in \mathcal{N}, \forall t \tag{14}$$

$$\sum_{i\in\mathcal{N}} b_{m,k,i,j}(t) \le U_j^I, \forall j \in \mathcal{N}, \forall t \tag{15}$$

The objective in **O2** is clearly to minimize the effective CCT. Eq. (13) enforces all data transmissions of the coflow must be completed within $[0, \Gamma_{m,k}]$. Eq. (14) and Eq. (15) are bandwidth capacity constraints on egress/ingress link of each server, respectively. The integration in **O2** can be eliminated without incurring any performance degradation, as shown in the following:

Theorem 2. *Suppose that $\tilde{b}_{m,k,i,j}$ and $\tilde{\Gamma}_{m,k}$ are the optimal solution to the following problem **O3**:*

$$\min_{\{b_{m,k,i,j}\}} \Gamma_{m,k} \tag{16}$$

$$\text{s.t: } \Gamma_{m,k} b_{m,k,i,j} = V_{m,k,i,j}, \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \tag{17}$$

$$\sum_{j\in\mathcal{N}} b_{m,k,i,j} \le U_i^E, \forall i \in \mathcal{N} \tag{18}$$

$$\sum_{i\in\mathcal{N}} b_{m,k,i,j} \le U_j^I, \forall j \in \mathcal{N} \tag{19}$$

Then, $b_{m,k,i,j}(t) = \begin{cases} \tilde{b}_{m,k,i,j}, & t \in (0, \tilde{\Gamma}_{m,k}] \\ 0, & t \in (\tilde{\Gamma}_{m,k}, \infty] \end{cases}$ *and* $\Gamma_{m,k} = \tilde{\Gamma}_{m,k}$ *are the solutions to achieve the optimal objective of **O2**. Note here that the $b_{m,k,i,j}$ in **O3** can be viewed as the time-averaged bandwidth allocated to the $i \to j$ flow of coflow $k \in \mathcal{C}_m$.*

Proof of Theorem 2: Consider that $b^*_{m,k,i,j}(t)$ and $\Gamma^*_{m,k}$ are the optimal solution of **O2**. We set

$$b_{m,k,i,j} = \frac{\int_0^{\Gamma^*_{m,k}} b^*_{m,k,i,j}(t)dt}{\Gamma^*_{m,k}} \tag{20}$$

Then, it is obvious that

$$b_{m,k,i,j}\Gamma^*_{m,k} = \int_0^{\Gamma^*_{m,k}} b^*_{m,k,i,j}(t)dt = V_{m,k,i,j} \tag{21}$$

By integrating both sides of Eq. (14) from 0 to $\Gamma^*_{m,k}$, we yield

$$\int_0^{\Gamma^*_{m,k}} \sum_{j\in\mathcal{N}} b^*_{m,k,i,j}(t)dt \le U_i^E \Gamma^*_{m,k} \tag{22}$$

By swapping the order between the integration and summations, we have

$$\begin{aligned} &\int_0^{\Gamma^*_{m,k}} \sum_{j\in\mathcal{N}} b^*_{m,k,i,j}(t)dt = \sum_{j\in\mathcal{N}} \int_0^{\Gamma^*_{m,k}} b^*_{m,k,i,j}(t)dt \\ &= \sum_{j\in\mathcal{N}} b_{m,k,i,j}\Gamma^*_{m,k} \le U_i^E \Gamma^*_{m,k} \end{aligned} \tag{23}$$

By eliminating $\Gamma^*_{m,k}$ from both sides, we get Eq. (18). We can infer Eq. (19) in the similar way.

Above discussions shows that $b_{m,k,i,j}$ and $\Gamma^*_{m,k}$ is a feasible solution to **O3**. Hence, we have

$$\Gamma^*_{m,k} \ge \tilde{\Gamma}_{m,k} \tag{24}$$

In addition, we can easily verify that the settings in Theorem 1 are also feasible solutions to problem **O2**. Therefore, we have

$$\tilde{\Gamma}_{m,k} \ge \Gamma^*_{m,k} \tag{25}$$

Accordingly, we have $\tilde{\Gamma}_{m,k} = \Gamma^*_{m,k}$. □

Theorem 1 shows that we can solve **O3** instead of **O2** to calculate the bandwidth allocation of each individual flow to minimize the effective CCT of a coflow when send all flows in a constant rate.

We can easily check that the problem **O3** is a LP. Though LP is efficient to be solved with standard solvers like MOSEK, we do not need to solve it explicitly. The reason is that we can directly derive its analytic solution, as shown in the following:

$$\tilde{\Gamma}_{m,k} = \max\left\{ \max_i \frac{\sum_{j\in\mathcal{N}} V_{m,k,i,j}}{U_i^E}, \max_j \frac{\sum_{i\in\mathcal{N}} V_{m,k,i,j}}{U_j^I} \right\} \tag{26}$$

$$b_{m,k,i,j} = \frac{V_{m,k,i,j}}{\tilde{\Gamma}_{m,k}}, \forall i \in \mathcal{N}, \forall j \in \mathcal{N} \tag{27}$$

Handling multiple jobs with multiple coflows: The last step is to rescale the bandwidth of the LP analytic solution for each coflow in each job according to the job weights. We denote $\mathcal{C}_\Omega$ as the set of active coflows in current time from all jobs in $\mathcal{J}_\Omega$. Recall that each job can only have one active coflow. Hence, $|\mathcal{C}_\Omega| = |\mathcal{J}_\Omega|$. We set the weight of a coflow to that of its parent job and scale its individual flows' bandwidth based on the portion of its weight in the sum of all coflow weights. To be specific, we denote $\{\Gamma_\ell^{(O3)}, b_{\ell,i,j}^{(O3)}\}$ as the LP solution for coflow $\ell \in \mathcal{C}_\Omega$. Then, we scale the bandwidth of coflow ℓ with $b_{\ell,i,j} \leftarrow b_{\ell,i,j}^{(O3)} \cdot \frac{W_\ell}{\sum_{\ell'\in\mathcal{C}_{\Omega^*}} W_{\ell'}}$. W_ℓ is the weight of coflow ℓ, which can be inherited from its parent job.

5.3 Analysis

The whole scheduling procedure of *Parrot* is summarized in Algorithm 1. It works in a *laissez-fair* manner. In other words, it will be invoked whenever observing a coflow becoming valid for scheduling or an existing coflow being completed (Step 1). According to the observed events, it updates $\mathcal{C}_\Omega$ and $\mathcal{J}_\Omega$, respectively (Step 2). Step 3 infers the SRPT job using the least per-coflow attained service heuristic described above. The job weight assignment is shown in Steps 4-6. Steps 7-9 depict the weighted bandwidth scaling process. Step 10 scales all the flow bandwidths by the same largest possible factor, to make use of the rest of the bandwidth.

We now analyze the theoretical performance achieved by Algorithm 1. In our analysis, we first derive the lower bound of the optimal JCT first and then compute the upper bound achieved by Algorithm 1 based on this lower bound. It should be noted that such kind of analysis method has been widely adopted in existing network scheduling literature [24, 26]. However, our analysis integrates some unique properties: 1) The number of concurrent coflows is no more than the number of

Algorithm 1 *Parrot* Scheduling Algorithm

```
1:  while observing a coflow from either newly arrived or currently active job becoming valid or an existing coflow being completed do
2:    Update the set of concurrent jobs $\mathcal{J}_\Omega$ and the set of active coflows $\mathcal{C}_\Omega$ respectively.
3:    Sort all the coflows in $\mathcal{C}_\Omega$ non-increasingly according to the per-coflow attained service of their parent jobs; $\mathcal{C}_{\Omega^*} \leftarrow \mathcal{C}_\Omega$. Similarly, sort all jobs with LPCAS; $\mathcal{J}_{\Omega^*} \leftarrow \mathcal{J}_\Omega$. Identify the first job in $\mathcal{J}_{\Omega^*}$ as the SRPT one.
4:    Update the occupancy ratio $\theta$ based on Eq. (10).
5:    Update $W_{max}$ based on Eq. (11).
6:    Set the weight of the inferred SRPT job to $W_{max}$ and all others to $W_{min} = 1$. Also, set the weights of all coflows in $\mathcal{C}_{\Omega^*}$ as those of their parent jobs.
7:    for each coflow $\ell$ in $\mathcal{C}_{\Omega^*}$ do
8:      $b_{\ell,i,j} \leftarrow b^{(O3)}_{\ell,i,j} \cdot \frac{W_\ell}{\sum_{\ell' \in \mathcal{C}_{\Omega^*}} W_{\ell'}}, \forall i, \forall j$, where $\{\Gamma^{(O3)}_\ell, b^{(O3)}_{\ell,i,j}\}$ is the optimum of O3 for $\ell$.
9:    end for
10:   Find a largest factor to scale the bandwidths of all flows in $\mathcal{C}_{\Omega^*}$ to pursue work conversing property.
11: end while
```

DML jobs; 2) At any time, there is only one DML job having a high weight and the remaining jobs' weights are all 1. Note that the analysis is not tight, and the competitive ratio derived in our paper depends on the number of jobs (i.e., M). For instance, in private clusters used for scientific research, the number of training jobs in a cluster can be well controlled, meaning that M may have an upper-bound. In this case, the competitive ratio is bounded. Whereas in production clusters, users may submit a large number of jobs. As such, M could be unbounded, resulting in an unbounded competitive ratio. Despite the weak analysis, the experiments in Sec. 6 show that our algorithm actually has superior performance. One may further wonder if the job mis-inference gap can be leveraged to make the analysis more tight, which, however, encounters the following challenges. First, it is hard to know if a job is mis-inferred as an SRPT job without prior knowledge of job size. Second, it is hard to quantify the mis-inference gap. Third, the impact of the mis-inference gap on the total JCT is unknown. Given the factors above, we leave this point to future work. The theoretical analysis is shown in the following theorems.

Theorem 3 (Lower Bound of Optimal JCT). *Let $T^{(O1)}$ denote the total JCT under the optimal solution of **O1**. Then, its lower bound is $T^{(O1)} \geq \sum_{m\in\mathcal{J}}(\tau_m + \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k})$.*

Proof: It is obvious that each job contributes to the total JCT with no less than its minimum completion time when it monopolizes the network. We denote $\Gamma^{(O3)}_{m,k}$ as the optimal effective CCT of coflow $k \in \mathcal{C}_m$ for **O3**. The minimum JCT of job m when it monopolizes the network can then be calculated as $\tau_m + \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k}$. When there are multiple jobs, each job's JCT must be larger than its minimum one. Therefore, we have $T^{(O1)} \geq \sum_{m\in\mathcal{J}}(\tau_m + \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k})$. Thus, proved. □

Theorem 4 (Upper Bound the Competitive Ratio). *The total JCT achieved by Algorithm 1 is given by $T^{(ALG)} \leq \frac{M}{1-\theta_{max}} T^{(O1)}$, where M is the number of jobs and θ_{max} is the maximum occupancy ratio of network bandwidth that the inferred SRPT job can use.*

Proof: We denote $T^{(ALG)}_m$ as the JCT of m under Algorithm 1, which can be calculated by

$$T^{(ALG)}_m = \tau_m + \sum_{k\in\mathcal{C}_m} \Gamma^{(ALG)}_{m,k} \tag{28}$$

where $\Gamma^{(ALG)}_{m,k}$ is the effective CCT of coflow $k \in \mathcal{C}_m$ achieved by Algorithm 1. Let $\mathcal{C}^{m,k}_{\Omega^*}$ denote the set of all active coflows when the coflow $k \in \mathcal{C}_m$ is scheduled. Since Algorithm 1 scales the flow bandwidths of each active coflow based on its weight, we have

$$\Gamma^{(ALG)}_{m,k} = \Gamma^{(O3)}_{m,k} / \frac{W_{m,k}}{\sum_{\ell'\in\mathcal{C}^{m,k}_{\Omega^*}} W_{\ell'}} \tag{29}$$

where $\Gamma^{(O3)}_{m,k}$ is the optimal effective CCT of $k \in \mathcal{C}_m$ for problem **O3** and $W_{m,k}$ is the relevant weight. Combining Eq. (28) and Eq. (29), we have

$$\begin{aligned} T^{(ALG)} &= \sum_{m\in\mathcal{J}} T^{(ALG)}_m \\ &= \sum_{m\in\mathcal{J}} \tau_m + \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k} / \frac{W_{m,k}}{\sum_{\ell'\in\mathcal{C}^{m,k}_{\Omega^*}} W_{\ell'}} \end{aligned} \tag{30}$$

We define $T^{(ALG)}_1 = \sum_{m\in\mathcal{J}} \tau_m$ and $T^{(ALG)}_2 = \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k} / \frac{W_{m,k}}{\sum_{\ell'\in\mathcal{C}^{m,k}_{\Omega^*}} W_{\ell'}}$. In the following, we focus on deriving an upper bound of $T^{(ALG)}_2$. Because each coflow's weight is at least W_{min}, we yield

$$T^{(ALG)}_2 \leq \Bigg(\sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \frac{\Gamma^{(O3)}_{m,k}}{W_{min}} \Bigg) \Bigg(\sum_{\ell'\in\mathcal{C}^{m,k}_{\Omega^*}} W_{\ell'} \Bigg) \tag{31}$$

Let $\mathcal{C}^{max}_{\Omega^*}$ denote the largest set of active coflows across the entire scheduling time. Clearly, it has more elements than $\mathcal{C}^{m,k}_{\Omega^*}$. In addition, given that there is only one W_{max} for any set of active coflows, we can get

$$T^{(ALG)}_2 \leq \Bigg(\sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \frac{\Gamma^{(O3)}_{m,k}}{W_{min}} \Bigg) \Big(W_{max} + (|\mathcal{C}^{max}_{\Omega^*}| - 1) W_{min} \Big) \tag{32}$$

Substituting $W_{min} = 1$ and $W_{max} = \frac{\theta}{1-\theta}(|\mathcal{J}_\Omega| - 1)$ into the above inequality, then using the fact that each job has only one active coflow at any time, we obtain

$$T^{(ALG)}_2 \leq \Bigg(\frac{\theta}{1-\theta}(|\mathcal{C}_\Omega| - 1) + (|\mathcal{C}^{max}_{\Omega^*}| - 1) \Bigg) \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k} \tag{33}$$

Since $|\mathcal{C}^{max}_{\Omega^*}| \geq |\mathcal{C}_\Omega|$ and $\theta \leq \theta_{max}$, we yield

$$\begin{aligned} T^{(ALG)}_2 &\leq \frac{1}{1-\theta_{max}} (|\mathcal{C}^{max}_{\Omega^*}| - 1) \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k} \\ &\leq \frac{1}{1-\theta_{max}} M \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma^{(O3)}_{m,k} \end{aligned} \tag{34}$$

The last inequality is derived by using $|\mathcal{C}_{\Omega^*}^{max}| \leq M$. Combining the result in Theorem 3 and using the fact $\frac{M}{1-\theta_{max}} \geq 1$, we can get

$$\begin{aligned} T^{(ALG)} &= T_1^{(ALG)} + T_2^{(ALG)} \\ &\leq \sum_{m\in\mathcal{J}} \tau_m + \frac{M}{1-\theta_{max}} \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma_{m,k}^{(O3)} \\ &\leq \frac{M}{1-\theta_{max}} \left(\sum_{m\in\mathcal{J}} \tau_m + \sum_{m\in\mathcal{J}} \sum_{k\in\mathcal{C}_m} \Gamma_{m,k}^{(O3)} \right) \\ &\leq \frac{M}{1-\theta_{max}} T^{(O1)} \end{aligned} \tag{35}$$

□

Remarks: We now discuss how our *Parrot* algorithm can achieve the design goals listed in Section 4. *First*, our algorithm requires very little overhead since it only makes a scheduling decision when a coflow completes or a coflow becomes valid. Further, the decision making process is lightweight, as its dominate overhead lies in sorting jobs or coflows and can have a time complexity of $O(|\mathcal{J}_\Omega|\log(|\mathcal{J}_\Omega|))$. *Second*, Theorem 4 demonstrates that our algorithm can provide a theoretical guarantee for minimizing the total JCT. *Third*, we can observe that any job can get some bandwidth at any time, there will never exist a sign of waiting in any job. Thus, starvation-free can be ensured. The last line of Algorithm 1 corresponds to a speed-up operation which scales up the flow bandwidths with a factor to completely utilize the link capacity. This essentially means that our algorithm can purse work-conserving property.

6 Performance Evaluation

In this section, we perform extensive simulations with realistic workload generated from the production DML cluster to evaluate our *Parrot* scheduler. We compare the following schemes with *Parrot*:

- **Least-Bytes-Sent-First (Aalo):** each time schedules the job with least bytes sent and distributes residual bandwidth between remaining jobs to pursue the work-conserving property. This scheme is conceptually equivalent to Aalo [20].
- **Aalo without work-conserving (Aalo-wo-WC):** is a variant of Aalo, which disables work conserving when using Aalo, and hence underutilized link capacities will not be allocated to other jobs.
- **Least-Completed-Coflow-First (LCCF):** schedules the job that has the least number of completed coflows each time. Also, it will assign underutilized link capacities to the remaining jobs for achieving work-conserving.
- **LCCF without work-conserving (LCCF-wo-WC):** makes no attempt to fully utilize the residual link capacities after the job with the least number of completed coflows has been scheduled.
- **Parrot without work-conserving (Parrot-wo-WC):** shares all processes with *Parrot* except the work-conserving. More specifically, it disables Step 10 when running Algorithm 1.
- **Lower Bound (LB):** The lower bound of a job's JCT is calculated by assuming it to monopolize network, i.e., the sum of effective CCTs of all its child coflows. The lower bound on total JCT can then be computed directly.

6.1 Simulation setup

Network: We simulate an ML cluster consisting of 128 servers. We mainly consider two network scenarios, where the ingress/egress link capacities of all servers are set to 10Gbps and 40Gbps, respectively.

Workloads: We use Microsoft job trace [27] which has 60 ML jobs. For each job, the trace contains its arrival time, number of desired GPUs, number of iterations, ML model name, and duration. We scale up the number of GPUs for each job by two times. For each job, all the required GPUs serve for workers, and we construct an equal number of PSs and workers. Each GPU serves for one worker only, and each PS or worker can only be placed on one server. Note that we do not use GPUs for PSs because PSs simply aggregate parameter updates from all workers and typically do not need GPUs. We then randomly place each job's PSs and workers in the simulated ML cluster. We generate traffic between PSs and workers for each push/pull phase of a job by following the all-to-all traffic pattern. To fit job durations, we let the jobs with longer per iteration duration to have larger coflow length. To be particular, we set the largest coflow length among all jobs to 1000MB and then make the coflow length of each job to be proportional to its per iteration duration. After the length of a coflow has been determined, its individual flow sizes are randomly selected within its length. The results from Fig. 5 to Fig. 8 are based on this 60-job workload.

We further generate 100 jobs based on the above characteristics. Specifically, we randomly sample 1 job from the 60-job workload by 100 times and hence get 100 jobs. We keep all job information while enforcing their arrivals to follow a Poisson process. We vary the average inter-job arrival interval μ from 100ms to 1000ms, so as to evaluate the impact of network load on *Parrot*. The results from Fig. 9 are based on this 100-job workload.

Simulator: We evaluate *Parrot* with an event-based flow-level simulator by performing a replay of the above workloads. Our simulator preserves dependencies between coflows and assumes perfect computation optimization: a coflow can start immediately after its dependent coflow finishes. Our simulator makes scheduling decisions only when observing a coflow becoming newly valid or being completed. We denote the observing interval as η. We will vary η from 1ms to 100ms to evaluate its impact on *Parrot*.

Parameter settings: Unless otherwise specified, we set the observing interval η to 10ms, the maximum occupancy ratio of the inferred SRPT job θ_{max} to 0.9.

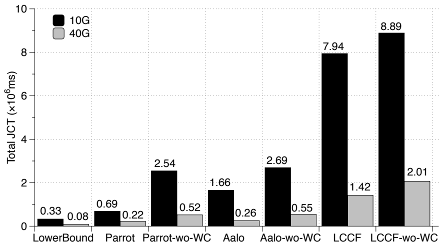


Fig. 5. Total JCTs achieved by various schemes under both 10G and 40G networks.

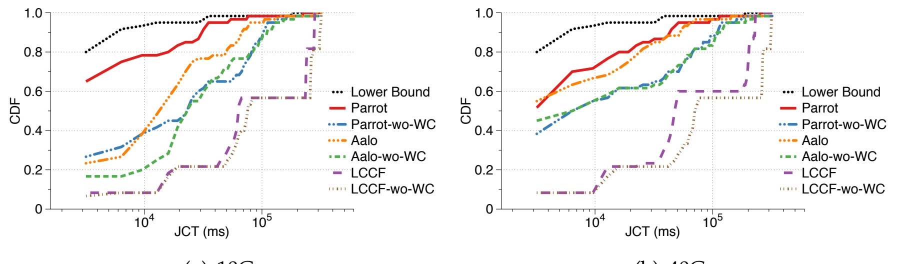


Fig. 6. CDFs of JCTs achieved by different schemes in both (a) 10G and (b) 40G networks. The X-axes are in logarithmic scale.

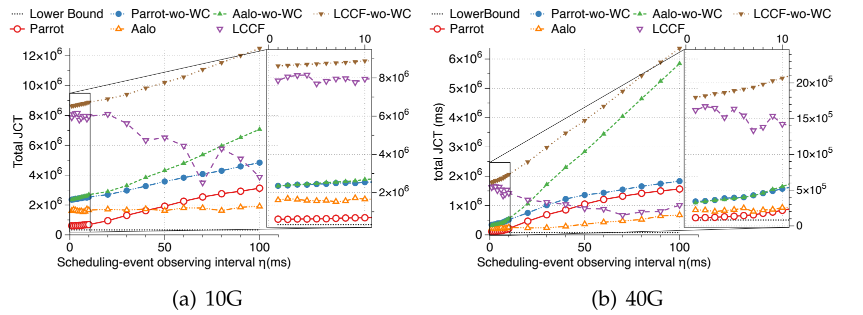


Fig. 7. Total JCTs achieved by different schemes under varying scheduling-event observing interval η in both (a) 10G and (b) 40G networks.

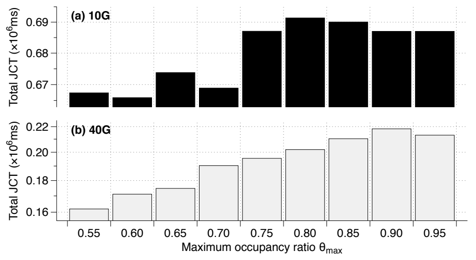


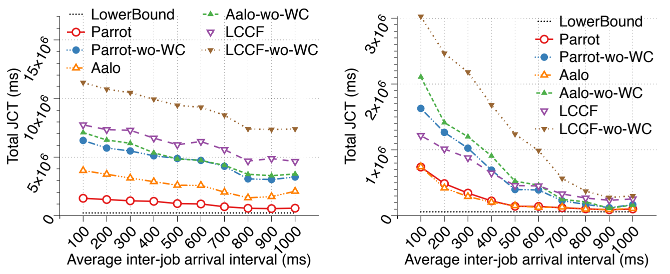
Fig. 8. Total JCT achieved by *Parrot* under varying maximum occupancy ratio θ_{max} in both 10G and 40G networks.

6.2 Simulation results

Performance on JCTs: Fig.5 first depicts the total JCTs obtained by different schemes with 10G and 40G networks. From this figure, we have the following observations. First, *Parrot* is closest to the Lower Bound, with the total JCT being up to 2.75× its lower bound, compared to other schemes. Second, across the 10G and 40G networks, *Parrot* can reduce the total JCT by up to 58.4% and 91.3%, compared to Aalo and LCCF, respectively. These results directly verify the efficiency of *Parrot*. Third, when work-conserving is disabled, each scheme incurs a relatively high total JCT. The reason is straightforward because some bandwidth will be wasted when disabling work-conserving. Fourth, the reduction in total JCT achieved by *Parrot* in the 40G network is less than that in the 10G network. The underlying reason is that the 40G network is not as congested as 10G one given the same workload, thus reducing the optimization space that *Parrot* can take effect for minimizing total JCT. Fifth, LCCF performs worse than Aalo because the job with the least bytes sent is more likely to be a short job, as compared to that with the least number of completed coflows.

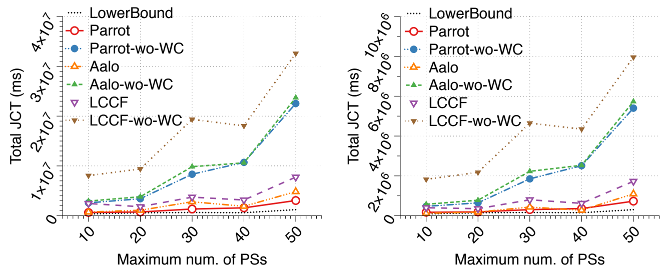
To understand the improvements of *Parrot* on a microscopic level, we further plot the CDFs of JCT for all jobs achieved by different schemes under both 10G and 40G networks in Fig. 6. The higher the CDF curve, the lower JCT the corresponding scheme can achieve. Clearly, in both 10G and 40G networks, the CDF curve of our *Parrot* is highest among the curves of all schemes excluding Lower Bound. Specifically, in the 10G network, 78.3% of jobs experience a JCT smaller than 10 seconds, while that factions for Aalo and LCCF are 48.3% and 8.3%, respectively. The results in Fig. 6 has the same trends as that in Fig. 5, i.e., for each scheme, disabling work-conserving leads to relatively high JCTs; the performance advantages of *Parrot* in 40G network is not as significant as it is in 10G network, given the same workload.

Impact of scheduling-event observing interval η: So far, the length of η is set to 10ms. Intuitively, a larger η will make the scheduler to miss some potential optimization opportunities. For example, if a coflow is completed in the middle of a specific observing interval, it can only be observed at the end of this interval. Afterward, the scheduling will be invoked. To quantify the impact of η, we use the same settings as above, but vary the observing interval η from 1ms to 100ms. Under varying η, Fig. 7 plots the total JCTs achieved by different schemes in both 10G and 40G networks. We can observe that for most schemes excluding LCCF, the total JCT increases with the growth of η. This is reasonable because a larger η makes the system to have less frequency in observing the scheduling events, resulting in fewer optimization opportunities for reducing total JCT. One may question why LCCF's total JCT does not grow as η increases. The reason might be that LCCF gives higher priorities to the jobs that are not shortest or SRPT ones, and hence larger η provides LCCF fewer opportunities to carry out such mis-prioritization. One may further wonder why *Parrot* performs worse than Aalo when η increases to 50ms/10ms in the 10G/40G network. The underlying reason is that with a larger η, *Parrot* cannot quickly observe a scheduling event, and hence the SRPT job will be delayed to be scheduled, causing a higher total JCT. Despite this, as the observing interval is usually set to 10ms or even sub-millisecond in existing network systems [21, 48], *Parrot* can show significant performance merits over all the other schemes.



(a) 10G (b) 40G

Fig. 9. Total JCTs achieved by various schemes under varying network load in both 10G and 40G networks.



(a) 10G (b) 40G

Fig. 10. Total JCTs achieved by various schemes under varying maximum number of PSs (i.e., Δ) in both 10G and 40G networks.

Impact of maximum occupancy ratio θ_{max}: Recall that *Parrot* ensures the inferred SRPT job not to monopolize the network in case that it is not the SRPT one. Hence, it configures a value θ_{max} to enforce the ratio of link bandwidth that the inferred job can use to not exceed θ_{max}. From Theorem 4, we note that the performance of *Parrot* depends on the value of θ_{max}. We vary θ_{max} from 0.55 to 0.95 and keep all the other settings unchanged, so as to evaluate the impact of θ_{max}. The results are shown in Fig. 8. We observe that the total JCT with a low value of θ_{max} (i.e., < 0.75) is smaller than that with a high value of θ_{max}. These results are approximately aligned with the implications in Theorem 4 that the larger the θ_{max}, the looser the upper bound is for the total JCT achieved by our *Parrot*. Despite that a larger θ_{max} leads to a looser upper bound for total JCT, our *Parrot* can still outperform Aalo and LCCF under a high $\theta_{max} = 0.9$.

Impact of network load: In this experiment, we use the 100-job workload described in Sec. 6.1. These jobs follow Poisson arrival patterns. To construct different network loads, we control the average inter-job arrival interval μ. In fact, the network load is closely related to μ. When all jobs arrived at the same time, i.e., $\mu = 0$, the network is always full utilized for a long time. When μ becomes larger and larger, less load will be injected into the network, and there will be fewer concurrent jobs sharing the network. We vary the inter-job arrival interval μ from 100ms to 1000ms and plot the total JCTs achieved by different schemes under varying μ in both 10G and 40G networks in Fig. 9. We can clearly observe that the total JCTs of all schemes decreases as the increasing of μ because a larger μ incurs a lower network load. We can further observe that *Parrot* can always achieve the lowest total JCT among all other schemes in the 10G network, irrespective of the change of μ. Specifically, compared to Aalo and LCCF, *Parrot* can reduce the total JCT by up to 61.6% and 80.9%, respectively. Under the 40G network, Aalo achieves nearly the same total JCT with *Parrot*. The reason is again that the 40G network enables *Parrot* to have fewer optimization opportunities. One can also observe that each scheme performs worse than its original scheme if its work-conserving is disabled. This is because that some bandwidth is inevitable to be wasted when disabling work-conserving.

Impact of PS number: The above experiments all consider an equal number of PSs and workers for each job. In this experiment, we make the number of workers and PSs different for each job. Specifically, we use the 60-job workload but control the number of PSs not exceeding a predefined threshold of Δ. In other words, the number of PSs for each job is randomly chosen in the range $[1, \Delta]$. The number of workers for each job is unchanged. We generate traffic for each job using the similar way as described in Sec. 6.1. Fig. 10 depict the total JCTs achieved by different schemes under varying values of Δ in both 10G and 40G networks. It is clear that *Parrot* achieves a lower total JCT than all other schemes across all settings of Δ in both 10G and 40G networks, and has a little gap to the ideal lower bound. More specifically, compared to Aalo and LCCF 10G (40G) network, *Parrot* can reduce the total JCT by up to 51.9% (2.9%) and 73.2% (61.6%), respectively. Across all settings, the total JCT achieved by *Parrot* can be as low as 1.26× it's lower bound and is at most 2.6× the lower bound. We can further observe that the total JCTs achieved by all schemes increase with the growth of Δ. The reason is that we use the all-to-all pattern to generate traffic between PSs and workers. Hence, a larger maximum number of PSs could lead to a higher traffic load in the network given the unchanged number of workers, thus leaving more space for our *Parrot* to take effect for reducing total JCT. These results directly demonstrate the efficiency of our *Parrot* in reducing the total JCT in scenarios with a different number of PSs and workers per job.

7 Discussion

Considering computation time: So far, our work only considers communication time for DML jobs. However, the workers take time to compute parameter updates, and the PSs also need time to aggregate updates from all workers. A simple way to consider such computation time would be to assume it to be a constant. To be particular, one can use $e_{m,k}$ to represent the computation time associated with the coflow $k \in \mathcal{C}_m$. As such, we need to make three changes to our model. First, the completion time of each job $m \in \mathcal{J}$ should be updated as

$$T_m = \tau_m + \sum_{k \in \mathcal{C}_m} (e_{m,k} + \Gamma_{m,k}) \tag{36}$$

Second, for every job m's every coflow $k \in\in \mathcal{C}_m$, its CCT should consider the computation time, as shown in the following.

$$T_{m,k} = \tau_m + e_{m,k} + \Gamma_{m,k} + \sum_{k' \prec k \in \mathcal{C}_m} (e_{m,k'} + \Gamma_{m,k'}) \tag{37}$$

Third, additional constraints should be introduced to prevent any coflows from transmitting data during the computation time. Specifically, $\forall m \in \mathcal{J}, \forall k' \prec k \in \mathcal{C}_m, \forall i, j \in \mathcal{N}$, we have the following equality

$$\int_{T_{m,k'}}^{T_{m,k'}+e_{m,k}} b_{m,k,i,j}(t)dt = 0, \tag{38}$$

With Eqs. (36)(37)(38), one can get a new model

$$\min_{b_{m,k,i,j}(t)} \sum_{m\in\mathcal{J}} T_m \text{ s.t. Eqs. (1)(2)(3)(4)(5)(6)(38)} \tag{39}$$

To solve this new model, one can use the unmodified Algorithm 1. However, this will make Theorem 4 invalid, and no guarantee can be provided on the upper bound of total JCT. The crux is that the number of concurrent jobs (i.e., $|\mathcal{J}_\Omega|$) may not be equal to that of concurrent coflows (i.e., $|\mathcal{C}_\Omega|$) since some jobs may be in the computation stage. One possible way would be to always find the set of concurrent jobs (e.g., $|\hat{\mathcal{J}}_\Omega|$) that are in communication stages. Then, one can define the occupancy ratio θ and the weight W_{max} for the inferred SRPT job with this new set $|\hat{\mathcal{J}}_\Omega|$. In such a case, we can still guarantee the total JCT to be no more than $\frac{M}{1-\theta_{max}}$ times the optimum. Specifically, one can update the lower bound as $T^{(O1)} \geq \sum_{m\in\mathcal{J}}(\tau_m + \sum_{k\in\mathcal{C}_m}(e_{m,k} + \Gamma_{m,k}^{(O3)}))$. Then, one can define $T_3^{(ALG)} = \sum_{m\in\mathcal{J}}\sum_{k\in\mathcal{C}_m} e_{m,k}$. Since each job has only one active coflow at any time, $|\hat{\mathcal{J}}_\Omega|$ now equals to $|\mathcal{C}_\Omega|$ and Eqs. (33)(34) still hold. As such, one can similarly have $T^{(ALG)} = T_1^{(ALG)} + T_2^{(ALG)} + T_3^{(ALG)} \leq \sum_{m\in\mathcal{J}}\tau_m + \sum_{m\in\mathcal{J}}\sum_{k\in\mathcal{C}_m} e_{m,k} + \frac{M}{1-\theta_{max}}\sum_{m\in\mathcal{J}}\sum_{k\in\mathcal{C}_m}\Gamma_{m,k}^{(O3)} \leq \frac{M}{1-\theta_{max}}\Big(\sum_{m\in\mathcal{J}}\tau_m + \sum_{m\in\mathcal{J}}\sum_{k\in\mathcal{C}_m}(e_{m,k} + \Gamma_{m,k}^{(O3)})\Big) \leq \frac{M}{1-\theta_{max}}T^{(O1)}$.

Allowing multiple active coflows per job: Our work assumes that there is only one active coflow per DML job at any time. However, this assumption would be invalid in scenarios supporting overlapping communication with computation [8, 49] because a DML job may have multiple active coflows. While dealing with multiple active coflows per job is out of our paper's scope, a possible design would be to divide the time into discrete timeslots and leverage a centralized arbiter to determine which flows can be transmitted in each timeslot. Specifically, in each time slot, the arbiter could first employ a two-stage ordering method to compute an order in which the concurrent flows are scheduled. The first stage is to order the flows in the job-level, meaning that the flows in higher priority jobs should be prioritized over those in lower priority ones. The policy in determining jobs' priorities could be shortest-effective-bottleneck-first. The effective bottleneck of a job can be computed as the longest layer-wise communication time per iteration. The second stage is to arrange the flows in the individual flow-level, where one can let the flow with the latest arrive time have the highest priority to ensure the forward computation of the former layer being started earlier. For flows having similar arrive times, one can prioritize larger flows over shorter flows to ensure the bottleneck flow in a coflow can be finished first. With such an order, the arbiter can process the flows in order, i.e., greedily allocating a source-destination pair if allocating the pair does not violate the bandwidth constraint. As such, the maximal matching can be achieved, meaning that none of the unallocated flows can be allocated while maintaining the bandwidth constraints. We leave this as future work.

Predicting job duration: So far, our work predicts the remaining progress of a job using the already attained service, which is essentially a heuristic and relies on heavy-tailed workload distribution to achieve good performance. A promising approach is to use ML technique. For example, one can use (online) model fitting [50] to predict the number of iterations required to achieve convergence for a job and then estimate the remaining progress by multiplying the observed average iteration time and the remaining iterations. This approach may require the DML job to have smooth loss curves. However, for many poor models during a trial-and-error exploration, their loss curves are not as smooth as the curves of the best model ultimately picked at the end of exploration. So, a more promising approach would be to learn job size from past system traces. We leave it to future work.

8 Related Work

There are tons of literature related to *Parrot*. We only review the most related ones below.

Coflow-aware network scheduling: Coflow scheduling has been widely used to improve the communication performance for distributed data-parallel jobs because coflow abstraction can capture the application-level semantics better than traditional individual flow model. Existing solutions focus on scheduling coflows from either single-stage or multi-stage jobs. Single-stage solutions concentrate on the primary performance metric—coflow completion time (CCT). They leverage efficient heuristics or approximation algorithms to minimize the average CCT [16–21, 24, 25, 51, 52], minimize the total weighted CCT [53], guarantee CCT within a specific deadline [17, 54, 55], and ensure fairness among coflows with respect to CCT [22, 23, 56, 57]. However, optimizing CCT does not help for multi-stage jobs, because their coflows have dependencies. Hence, optimizing JCT is more relevant for multi-stage jobs. To this end, Aalo employs a heuristic [20], which cannot provide a theoretical guarantee on the average JCT. Hence, Tian et al. [26] propose an approximation algorithm, which, however, relies on complete job information and involves solving a complicated LP program, making it being impractical.

Network optimizations for DML jobs: There are some other techniques to optimize the network performance for DML jobs. For instance, Hsieh et al. propose Gaia [11], which only sends significant gradients to servers in wide-area networks to speed up DML job's completion. Similar ideas are adopted in [12] and [13], which employs various filters to reduce communication between workers and parameter servers. Zhang et al. [8]

present a communication architecture for DML on GPUs, which explores layered ML model structures to hide the communication overhead behind backpropagation, and hence accelerate DML jobs. Taking a step further, Jayarajan et al. [58] propose P3, which can overlap communication with both forward and backward propagation using priority scheduling in MXNet architecture. Peng et al. use a similar idea and present ByteScheduler [49], which is different from P3 as it is a generic communication scheduler and can support multiple ML frameworks, including MXNet, PyTorch, and TensorFlow. Bao et al. further design PACE [59], which uses preemptive communication scheduling and tensor fusion to guarantee maximal overlapping of communication with computation for DAG-based DNN training and achieve high bandwidth utilization as well. TicTac [10] proposes finding the best scheduling order of network transfers through critical path analysis on the underlying computational graph, so as to guarantee a near-optimal overlap of communication and computation, improving the iteration time. While the works above can improve network performance efficiently for DML jobs, they mainly focus on accelerating communication for single job and are orthogonal and complementary to our work.

There are also some other efforts adopting the idea of prioritizing the short flows or coflows while avoiding them monopolize the network. For example, Varys [17] allocates the least amount of bandwidth to each scheduled (and short) coflow, so as to make all the flows in a coflow to keep the same pace and allow other coexisting coflows to make progress as well. Sincronia [25] uses an SRPT-like policy to order all unfinished coflows first and then sets the priorities of flows to be the order of their corresponding coflows. As such, flow scheduling and bandwidth allocation can be done by existing priority-enabled transport where different priority queues typically share the link bandwidth in a weighted manner, and weights are based on the queues' priorities. OMCoflow [24] leverages weighted sharing to allow large coflows to have more bandwidth and small coflows relatively less bandwidth. The solutions above all require prior knowledge of coflow characteristics like the number of flows and their sizes. Of course, there also exist information-agnostic solutions. For example, PIAS [47] and Aalo [20] use multiple-level feedback queues (MLFQ) and move flows/coflows gradually from higher-priority queues to lower-priority queues based on their total bytes sent, with the aim of prioritizing short flows/coflows over large ones. Besides, weighted fair queuing is adopted across queues to avoid starvation. However, the gradual priority demotion mechanism takes time to move large flows/coflows to low-priority queues, increasing the risk of allowing short and large flows/coflows to share the congested link and hence prolonging the short flows/coflows. By contrast, our *Parrot* can reduce such risk. Once a job is identified as an SRPT one, *Parrot* can separate it from other jobs by assigning a high weight to it and maintaining the same and relatively low weight for the remaining jobs.

9 Conclusions

In this paper, we study the problem of scheduling dependent coflows of multiple DML jobs to minimize the total JCT. We formulate the problem and prove its NP-hardness. We present an online coflow-aware scheduler called *Parrot*. Each time, *Parrot* employs a LPCAS heuristic to infer the SRPT job first. It then allocates the inferred job a relatively large amount of bandwidth while not starving any other job by proposing a dynamic job weight assignment mechanism and a LP-based weighted bandwidth scaling strategy. We have conducted rigorous theoretical analysis to prove that our *Parrot* algorithm can provide a non-trivial competitive in minimizing the total JCT for any given set of jobs. Extensive simulations based on realistic workloads demonstrate that *Parrot* outperforms the state-of-the-art solution, with the total JCT being reduced by 58.4%.

References

[1] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang *et al.*, "Large scale distributed deep networks," in *Proc. of NIPS*, 2012.

[2] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *arXiv preprint arXiv:1706.02677*, 2017.

[3] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *Proc. of IEEE HPCA*, 2018.

[4] W. Xiao, R. Bhardwaj, R. Ramjee, M. Sivathanu, N. Kwatra, Z. Han, P. Patel, X. Peng, H. Zhao, Q. Zhang *et al.*, "Gandiva: Introspective cluster scheduling for deep learning," in *Proc. of USENIX OSDI*, 2018.

[5] X. Jia, S. Song, W. He, Y. Wang, H. Rong, F. Zhou, L. Xie, Z. Guo, Y. Yang, L. Yu *et al.*, "Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes," *arXiv preprint arXiv:1807.11205*, 2018.

[6] L. Luo, J. Nelson, L. Ceze, A. Phanishayee, and A. Krishnamurthy, "Parameter hub: a rack-scale parameter server for distributed deep neural network training," in *Proc. of ACM SoCC*, 2018.

[7] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proc. of ACM/IEEE ISCA*, 2017.

[8] H. Zhang, Z. Zheng, S. Xu, W. Dai, Q. Ho, X. Liang, Z. Hu, J. Wei, P. Xie, and E. P. Xing, "Poseidon: An efficient communication architecture for distributed deep learning on gpu clusters," in *Proc. of USENIX ATC*, 2017.

[9] D. Alistarh, D. Grubic, J. Li, R. Tomioka, and M. Vojnovic, "Qsgd: Communication-efficient sgd via gradient quantization and encoding," in *Proc. of NIPS*, 2017.

[10] S. H. Hashemi, S. A. Jyothi, and R. H. Campbell, "Tictac: Accelerating distributed deep learning with communication scheduling," in *Proc. of SysML*, 2019.

[11] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-distributed machine learning approaching lan speeds," in *Proc. of USENIX NSDI*, 2017, pp. 629–647.

[12] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, "Scaling distributed machine learning with the parameter server," in *Proc. of USENIX OSDI*, 2014.

[13] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," in *Advances in Neural Information Processing Systems*, 2014, pp. 19–27.

[14] M. Jeon, S. Venkataraman, J. Qian, A. Phanishayee, W. Xiao, and F. Yang, "Multi-tenant gpu clusters for deep learning workloads: Analysis and implications," Technical report, MSR-TR-2018-13, Tech. Rep., 2018.

[15] J. Jiang, B. Cui, C. Zhang, and L. Yu, "Heterogeneity-aware distributed parameter servers," in *Proc. of the 2017 ACM International Conference on Management of Data*. ACM, 2017, pp. 463–478.

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

[17] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient coflow scheduling with varys," in *Proc. of ACM SIGCOMM*, Chicago, IL, USA, 2014.

[18] Y. Zhao, K. Chen, W. Bai, C. Tian, Y. Geng, Y. Zhang, D. Li, and S. Wang, "Rapier: Integrating routing and scheduling for coflow-aware data center networks," in *Proc. of IEEE INFOCOM*, HK, 2015.

[19] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron, "Decentralized task-aware scheduling for data center networks," in *Proc. of ACM SIGCOMM*, 2014.

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

[21] H. Zhang, L. Chen, B. Yi, K. Chen, M. Chowdhury, and Y. Geng, "Coda: Toward automatically identifying and scheduling coflows in the dark," in *Proc. of ACM SIGCOMM*, 2016.

[22] W. Wang, S. Ma, B. Li, and B. Li, "Coflex: Navigating the fairness-efficiency tradeoff for coflow scheduling," in *Proc. of IEEE INFOCOM*, 2017.

[23] L. Chen, W. Cui, B. Li, and B. Li, "Optimizing coflow completion times with utility max-min fairness," in *Proc. of IEEE INFOCOM*, 2016.

[24] Y. Li, S. H.-C. Jiang, H. Tan, C. Zhang, G. Chen, J. Zhou, and F. Lau, "Efficient online coflow routing and scheduling," in *Proc. of ACM MobiHoc*, 2016.

[25] S. Agarwal, S. Rajakrishnan, A. Narayan, R. Agarwal, D. Shmoys, and A. Vahdat, "Sincronia: near-optimal network design for coflows," in *Proc. of ACM SIGCOMM*, 2018.

[26] B. Tian, C. Tian, H. Dai, and B. Wang, "Scheduling coflows of multi-stage jobs to minimize the total weighted job completion time," in *Proc. of IEEE INFOCOM*, 2018.

[27] J. Gu, M. Chowdhury, K. G. Shin, Y. Zhu, M. Jeon, J. Qian, H. Liu, and C. Guo, "Tiresias: A gpu cluster manager for distributed deep learning," in *Proc. of USENIX NSDI*, 2019.

[28] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu, "Petuum: A new platform for distributed machine learning on big data," in *Proc. of SIGKDD*, 2015.

[29] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng, "Map-reduce for machine learning on multicore," in *Proc. of NIPS*, 2007.

[30] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proc. of USENIX OSDI*, 2016.

[31] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. of USENIX OSDI*, 2014.

[32] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," in *Proc. of the VLDB Endowment*, 2012.

[33] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "Mllib: Machine learning in apache spark," in *Proc. of CoRR*, 2015.

[34] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. of ACM Multimedia*, 2014, pp. 675–678.

[35] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems," *arXiv preprint arXiv:1512.01274*, 2015.

[36] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proc, of NIPS*, 2013.

[37] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc, of NIPS*, 2011.

[38] M. Chowdhury and I. Stoica, "Coflow: A networking abstraction for cluster applications," in *Proc. of ACM Workshop on Hot Topics in Networks*, 2012.

[39] 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 *Proc. of ACM SIGCOMM*, 2009.

[40] 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 *Proc. of ACM SIGCOMM*, 2009.

[41] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout, "Homa: A receiver-driven low-latency transport protocol using network priorities," in *Proc. of ACM SIGCOMM*, 2018.

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

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

[44] B. Kalyanasundaram and K. R. Pruhs, "Minimizing flow time nonclairvoyantly," *Journal of the ACM (JACM)*, vol. 50, no. 4, pp. 551–567, 2003.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[46] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proc. of ACM SIGCOMM*, 2015.

[47] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang, "Information-agnostic flow scheduling for commodity data centers," in *Proc. of USENIX NSDI*, 2015, pp. 455–468.

[48] L. Chen, J. Lingys, K. Chen, and F. Liu, "Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *Proc. of ACM SIGCOMM*, 2018.

[49] Y. Peng, Y. Zhu, Y. Chen, Y. Bao, B. Yi, C. Lan, C. Wu, and C. Guo, "A generic communication scheduler for distributed dnn training acceleration," in *Proc. of ACM SOSP*, 2019.

[50] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo, "Optimus: an efficient dynamic resource scheduler for deep learning clusters," in *Proc. of ACM EuroSys*, 2018.

[51] W. Li, X. Yuan, K. Li, H. Qi, and X. Zhou, "Leveraging endpoint flexibility when scheduling coflows across geo-distributed datacenters," in *Proc. of IEEE INFOCOM*, 2018.

[52] W. Li, D. Guo, A. X. Liu, K. Li, H. Qi, S. Guo, A. Munir, and X. Tao, "Coman: managing bandwidth across computing frameworks in multiplexed datacenters," *IEEE transactions on parallel and distributed systems*, vol. 29, no. 5, pp. 1013–1029, 2017.

[53] Z. Qiu, C. Stein, and Y. Zhong, "Minimizing the total weighted completion time of coflows in datacenter networks," in *Proc. of ACM SPAA*, 2015.

[54] S. Ma, J. Jiang, B. Li, and B. Li, "Chronos: Meeting coflow deadlines in data center networks," in *Proc. of IEEE ICC*, 2016.

[55] R. Xu, W. Li, K. Li, X. Zhou, and H. Qi, "Scheduling mix-coflows in datacenter networks," *IEEE Transactions on Network and Service Management*, 2020.

[56] L. Wang, W. Wang, and B. Li, "Utopia: Near-optimal coflow scheduling with isolation guarantee," in *Proc. of IEEE INFOCOM*, 2018.

[57] L. Wang and W. Wang, "Fair coflow scheduling without prior knowledge," in *Proc. of IEEE ICDCS*, 2018.

[58] A. Jayarajan, J. Wei, G. Gibson, A. Fedorova, and G. Pekhimenko, "Priority-based parameter propagation for distributed dnn training," in *Proc. of SysML*, 2019.

[59] Y. Bao, Y. Peng, Y. Chen, and C. Wu, "Preemptive all-reduce scheduling for expediting distributed dnn training," in *Proc. of IEEE INFOCOM*, 2020.

Wenxin Li received the B.E. and Ph.D degrees from the School of Computer Science and Technology, Dalian University of Technology, in 2012 and 2018 respectively. Currently, he is a postdoc researcher in Hong Kong University of Science and Technology. From May 2014 to May 2015, he was a research assistant in National University of Defense Technology. From Oct. 2016 to Sept. 2017, he was a visiting student at the Department of Electrical and Computer Engineering in the University of Toronto. His research interests include datacenter networks and cloud computing.

Sheng Chen received the bachelor's and master's degrees from Dalian Maritime University in 2011 and Dalian University of Technology in 2017, respectively. He is currently pursuing the Ph.D. degree with the College of Intelligence and Computing, Tianjin University, China. His research interests include data center network, edge computing, wireless sensing, and indoor localization.

Keqiu Li received the bachelors and masters degrees from the Department of Applied Mathematics at the Dalian University of Technology in 1994 and 1997, respectively. He received the Ph.D. degree from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology in 2005. He also has two-year postdoctoral experience in the University of Tokyo, Japan. He is currently a professor in the School of Computer Science and Technology, Dalian University of Technology, China. He has published more than 100 technical papers, such as IEEE TPDS, ACM TOIT, and ACM TOMCCAP. He is an Associate Editor of IEEE TPDS and IEEE TC. He is a senior member of IEEE. His research interests include internet technology, data center networks, cloud computing and wireless networks.

Heng Qi was a Lecture at the School of Computer Science and Technology, Dalian University of Technology, China. He got bachelor's degree from Hunan University in 2004 and master's degree from Dalian University of Technology in 2006. He servered as a software engineer in GlobalLogic-3CIS from 2006 to 2008. Then he got his doctorate degree from Dalian University of Technology in 2012. His research interests include computer network, multimedia computing, and mobile cloud computing. He has published more than 20 technical papers in international journals and conferences, including ACM Transactions on Multimedia Computing, Communications and Applications (ACM TOMCCAP) and Pattern Recognition (PR).

Renhai Xu received the B.E. and M.S. degrees from the School of Computer Science and Technology, Dalian University of Technology, China, in 2014 and 2017, respectively. Currently, he is a second-year PhD student in the School of Computer Science and Technology, Tianjin University, China. His research interests include datacenter networks and cloud computing.

Song Zhang received the bachelor's and master's degrees from Hunan University in 2015 and 2019, respectively. He is working toward the PhD degree with the School of Computer Science and Technology, TianJin University. His current research interests include data center networks and cloud computing.