ACCEPTED FROM OPEN CALL

A Survey of Coflow Scheduling Schemes for Data Center Networks

Shuo Wang, Jiao Zhang, Tao Huang, Jiang Liu, Tian Pan, and Yunjie Liu

ABSTRACT

Cluster computing applications, such as MapReduce and Spark, have been widely deployed in data centers to support commercial applications and scientific research. These applications often involve a collection of parallel flows generated by two groups of machines, and the slowest flow will determine the completion of applications. However, existing network-level optimizations are agnostic to the special communication pattern of the applications. Most of them only focus on improving the completion time of an individual flow instead of a collection of parallel flows.

The recently proposed coflow abstraction exactly expresses the requirements of cluster computing applications and creates new opportunities to reduce the completion time of jobs in cluster computing. Due to the wide variations in coflow characteristics, coflow scheduling faces great challenges in data center networks. Much work has been proposed to solve one or some of the various challenges. Therefore, in this article, we survey the latest development in coflow scheduling for data center networks. We hope that this article will help readers quickly understand the causes of each problem and learn about current research progress, so as to serve as a guide and motivation toward further research in this area.

INTRODUCTION

In today's data centers, cluster computing has become widely popular due to its high computing performance and low prices to deal with the increasing data process and analysis demands. The benefits of cluster computing lie in the ability to simultaneously run complex computations on a group of independent machines linked together by high-speed networks. This distributed computing model enables faster, more reliable and more scalable computing than traditional single-machine computing.

Cluster computing applications have many unique features. For example, most of the cluster computing applications, such as MapReduce and Spark, are frameworks that provide special programming models where the user creates jobs to pass input data through a set of operators. These jobs usually have multiple successive computation stages, and a succeeding computation stage can start only after all the intermediate data generated by the previous stage are in place. Generally, *the intermediate data transfer involves a collection of flows between two groups of machines, and the collection of flows is defined as a coflow* [1]. For example, a typical coflow is the shuffle between the mappers and reducers in MapReduce.

Recent studies have shown that intermediate data transmission accounts for more than 50 percent of jobs' completion times [2]. Although a large body of network-level optimizations has emerged to improve flow completion time (FCT) or fairness [3], they are agnostic to the job-specific communication requirements. This mismatch could hurt the performance of cluster computing applications, because the job cannot finish data transmission and enter the next stage until all its flows are completed. Hence, to improve job completion time, the optimization mechanism should schedule flows at the coflow level rather than at the individual flow level.

Managing all the coflows between the machine groups for a large number of jobs is commonly known as the coflow scheduling problem. The key part of coflow scheduling in data center networks is an assignment problem, where network bandwidth and host capacities need to be allocated to competing coflows and individual flows in a coflow. The collective objectives of the coflow scheduling problem are to minimize the average completion time for all coflows, meet the coflow deadlines, and obtain maximum utility defined by users.

The coflow scheduling problem is very challenging in data center networks due to the wide variations in coflow characteristics such as total size, the number of parallel flows, and the size of individual flows [3]. Even worse, in some cases, coflow characteristics might be unknown prior [4]. For example, the data is usually transferred as soon as it is generated in cluster computing applications like Spark, making it hard to know the size of each flow.

Therefore, several works have been proposed to solve one of the various challenges. Previous literature, such as [5], exhaustively introduces the topology architectures, transport schemes, load balancing schemes and other technologies to reduce the latency of data center networks. Thus, in this article, we omit these details about data centers and focus on coflow scheduling.

Due to the wide variations in coflow characteristics, coflow scheduling faces great challenges in data center networks. Much work has been proposed to solve one or some of the various challenges. Therefore, in this article, the authors survey the latest development in coflow scheduling for data center networks.

The authors are with Beijing University of Posts and Telecommunications, Beijing Advanced Innovation Center for Future Internet Technology, and Beijing Laboratory of Advanced Information Networks. Jiao Zhang is the corresponding author.

Digital Object Identifier: 10.1109/MCOM.2017.1700267

We introduce the background and causes and present a classification of the existing proposals. We describe the main motivations and features of such proposals and discuss the challenges and opportunities. Note that, to concisely introduce the coflow scheduling problem, we only show typical proposals and intentionally skip some state-of-the-art such as Seagull[6].

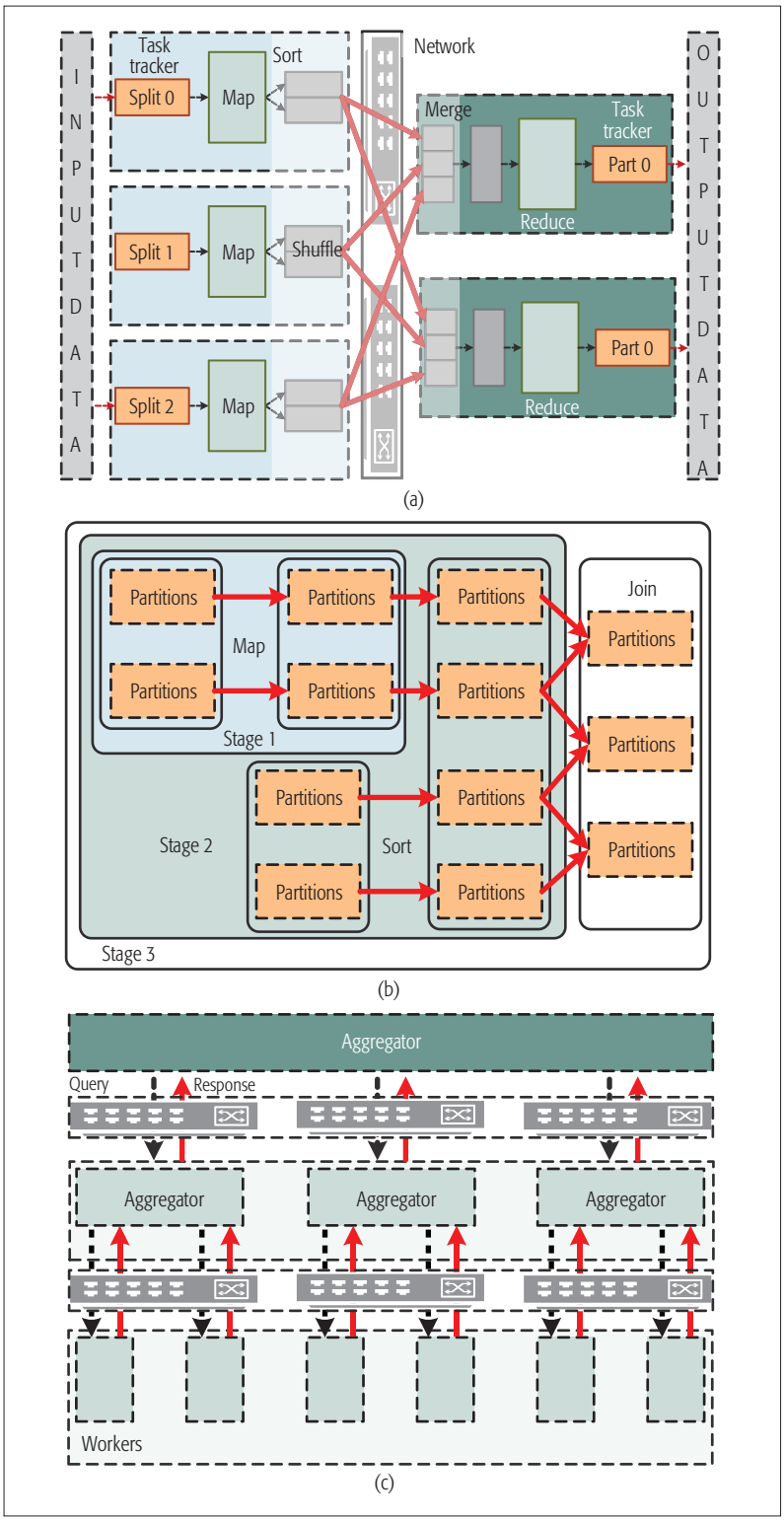

Figure 1. Three typical cluster computing applications. Boxes with dash outlines are machines, and red solid arrows are data flows: a) hadoop architecture; b) spark architecture; c) web search query process.

Cluster Computing Applications

In this section, we introduce three typical cluster computing applications. These applications are widely deployed and their communication patterns reveal some common characteristics for cluster computing applications.

MapReduce

MapReduce is a programming model designed by Google for processing big data sets on a group of machines. It has been implemented in Apache Hadoop. As shown in Fig. 1a, MapReduce usually processes data in four steps. First, to process massive data concurrently, the input is divided into small size files called 'splits' (e.g., 64 MB) and the Hadoop framework assigns one 'split' to each mapper running on distributed machines. In the second Map phase, the mapper processes 'split' and transform input records (key/value pairs) in 'split' into intermediate records that are sorted by keys. Then, in the shuffle phase, the output from all the Mappers are grouped by the key and each reducer obtains all records with the same key. Note that the Hadoop framework will create parallel flows between each mapper and each reducer in networks to transfer sorted records. Indeed, the job cannot begin its fourth phase until the shuffle has finished. Thus, one of the objects of coflow scheduling is to minimize the transmission time for these flows. In the last phase, the Reduce applies user-defined functions on sorted records to generate the final results.

Spark

Apache Spark is another popular open-source big data processing frameworks. Compared to Hadoop, it allows programmers to develop complex, multi-stage data pipelines using directed acyclic graph (DAG) pattern. DAG is a graph to clearly show how the data set is processed in each stage and which stage the data set should be sent to. For example, in Fig. 1b, a job has three different stages, and each stage runs user-defined functions (e.g., map, sort, join) to process the data set. The boundaries of the stages are the data transmission. For instance, stage 3 requires the computation results of stage 1 and stage 2. Therefore, the outputs of stage 1 and stage 2 are transferred to stage 3. As a result, stages have dependencies with each other. The coflow scheduler needs to handle the stage dependencies with the DAG and optimally schedule shuffle flows between stages.

Web Applications

The partition/aggregation workflow is an important communication pattern in large-scale web applications, such as Google search engine, home feeds in Facebook and Yahoo. As illustrated in Fig. 1c, one query from a user relies on computing results from many workers, thus each query task is broken into small pieces and assigned to the workers in the lower layer at each level. Then the responses from multiple workers are aggregated and sent to the upper layer to generate the final query result. Web applications are different from data-intensive computing applications, because each query task has strict deadlines. If the response time is too long, the user may refresh the page, and then the response is

dropped by the web applications, causing wasted bandwidth. Therefore, another important objective for coflow scheduling is to guarantee coflows finishing before deadlines.

Coflow Model

The performance of the application is related to all of the flows in cluster computing applications. Unfortunately, existing network abstractions lack the notion of a collection of flows and can not express the application-level semantics and requirements. Therefore, the authors in [1] first proposed the notion of coflow to capture the semantics of a collection of flows. In this section, we introduce the coflow abstraction and its associated characteristics and optimization objects.

Characteristics

A coflow is defined as a collection of parallel flows that have independent input and output. A Coflow k can be denoted as $C^{(k)}$, and it has a set of flows $\{d_{i,j}^{(k)}\}$, where $d_{i,j}^{(k)}$ denotes a flow from source i sends $d_{i,j}^{(k)}$ size data to destination j. For example, the collection of flows in the shuffle process in Fig. 1a is a typical coflow. The source and destination of each flow in the coflow may be different or overlapped, and each flow can start from a different time with the different size of data. Thus, the structure of coflows is too complex that its characteristics cannot be described using the notion of individual flows. Indeed, we use the following attributes to better describe the characteristics of a coflow [3]:

- *Size* is the sum of all flows'sizes in a coflow. The size can represent the total traffic of a coflow.
- *Length* is the size of the largest flow in a coflow. Generally, coflows can be classified as the short coflow and the long coflow based on their size (e.g., *length* ≤ 5MB is a short coflow).
- *Width* is the number of flows in a coflow. Similarly, coflows can be classified as the narrow coflow and the wide coflow based on their width (e.g., *width* ≤ 50 is a narrow coflow).
- *Start time* is the minimum start time of flows in a coflow.
- *End time* is the maximum end time of flows in a coflow.
- *Coflow Completion Time* (*CCT*) is the difference between the end time and the start time of a coflow.

Structure

By leveraging the coflow abstraction, the communication and requirements of cluster computing applications can be effectively represented. For example, the left side of Fig. 2 shows the coflow abstraction of the shuffle process in the Fig. 1a. The shuffle process is simplified to one coflow between two groups of machines that can clearly represent which flows belong to the job and should be optimized.

The shuffle has the simplest structure because it only has one stage. In multi-stage jobs, the structure of a job is more complex. Generally, one multi-stage job contains more than one coflow, and these coflows may depend on each other.

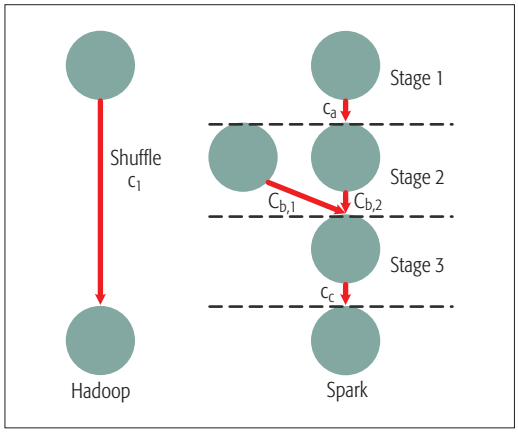

Figure 2. Using coflow abstraction represents the communication stages of cluster computing applications. Cycles denote a group of machines, and red solid arrows are coflows.

The authors in [4] show that there are two types of dependencies. The first type is *Starts-After*, which means a coflow cannot start until its parent coflow has finished. For example, on the right side of Fig. 2, because the input of the second stage is the output of the first stage, coflow $C_{b,2}$ has to wait until coflow C_a has finished. The second type of dependency is *Finishes-Before*, which means a coflow can coexist with another coflow but it cannot finish until another coflow has finished. This usually happens in multi-stage jobs without barriers that a stage can start as soon as some input in the previous stage is available.

Objective

The performance of a job is determined by all its flows. Therefore, one objective of the coflow scheduling problem is to minimize the CCT of a coflow. The coflow scheduler needs to assign proper rates for individual flows and decide when to start the data transmission of each flow, which is called *intra-coflow scheduling*. Further, when there are several coflows competing for network bandwidth, the objective becomes to minimize the average CCT by preempting coflows (e.g., [3, 4, 7, 8]). The scheduling mechanisms need to manage the order and priorities of coflows, which is called *inter-coflow scheduling*. On the other hand, some applications such as web search usually have a deadline, and the coflow needs to finish before the deadline. Thus, another important objective of coflow scheduling is to guarantee that coflows can meet their deadlines and reduce the rate of missed deadlines.

To optimize the above objectives, the coflow scheduler needs to know the information of the coflows, such as the size and length of a coflow. The coflow information is easy to obtain by modifying cluster computing applications in private data centers. However, in a public cloud or sharing cluster, it is impossible to modify applications to obtain the coflow information. Therefore, based on the availability of coflow information, existing coflow scheduling mechanisms can be broadly classified as information-aware coflow scheduling and information-agnostic coflow scheduling, as shown in Fig. 3. Information-aware coflow scheduling

The performance of a job is determined by all its flows. Therefore, one objective of the coflow scheduling problem is to minimize the CCT of a coflow. The coflow scheduler needs to assign proper rates for individual flows and decide when to start the data transmission of each flow, which is called intra-coflow scheduling.

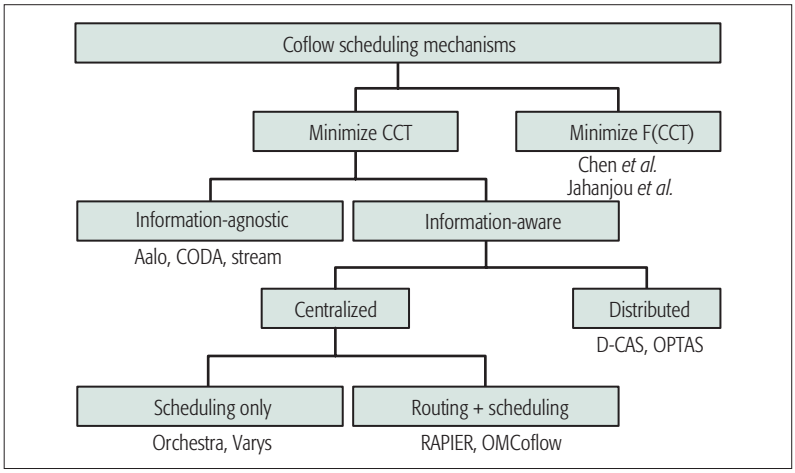

Figure 3. Classification of surveyed schemes.

schemes assume all the information of coflows is available, while the information-agnostic schemes try to schedule coflows when some important information is missing.

Information-Aware Coflow Scheduling

According to the architecture of the coflow scheduling mechanisms, we divide information-aware coflow scheduling mechanisms into two classes. The first class is centralized schemes, which leverage a centralized scheduler to control all the coflows in the network. This type of scheme usually has better performance since they have global information. The second class is distributed schemes, which leverage distributed protocols running on each machine to schedule coflows. This type of scheme achieves low overhead and better scalability.

Centralized

Scheduling Only: Orchestra [2] aims at optimizing *suffle* and *broadcast* communication patterns. Orchestra mainly comprises three components: the inter-transfer controller (ITC), which implements first-in-first-out (FIFO) and priority scheduling policies to schedule inter-coflows; the cornet broadcast transfer controller (TC), which achieves intra-coflow scheduling to manage *broadcast* coflows; and the weighted shuffle scheduling TC, which also achieves intra-coflow scheduling to manage shuffle coflows by assigning flow rates. The flow rates allocation is based on the weighted fair sharing algorithm according to flows' data size.

Built on Orchestra, Varys [3] can more efficiently schedule coflows with various structures. Varys proposes the smallest-effective-bottleneck-first (SEBF) heuristic to address inter-coflow scheduling and designs the minimum-allocation-for-desired-duration (MADD) algorithm to address intra-coflow scheduling. Specifically, Varys abstracts the network as a single big-switch connected to all the machines. First, for each coflow, SEBF calculates the minimum time (minimum CCT) to transfer the coflow's remaining data without violating the bandwidth limitation. Then, the coflow with the minimum CCT is scheduled first and allocated the maximum bandwidth. Finally, given the minimum CCT, the MADD algorithm sets the rates of each flow to the flow size divided by the minimum CCT. This enables MADD to allocate the least amount of bandwidth to complete a coflow in the minimum possible time.

Routing and Scheduling: Rapier [9] aims at reducing the average CCT by seamlessly integrating routing and scheduling, because the data center networks often leverage tree-based topologies. These topologies are not equivalent to a big-switch, and there are many paths between machines, considering that routing is significant to avoid link congestion. Rapier assumes all the information about coflows and network topologies can be easily known, and leverages the information to obtain optimal routing and scheduling policies. Specifically, Rapier first considers that there is only one coflow and formulates the one coflow scheduling problem as an optimization problem with bandwidth constraints. By iterating coflows and solving the optimization problem, the CCT, routing and scheduling policy of each coflow is obtained. Then, Rapier proposes a shortest job first (SJF) based heuristics algorithm that prefers to allocate more bandwidth to coflows with minimal CCT. However, the heuristics-based inter-coflow scheduling algorithm in Rapier may cause flows to be frequently rerouted. OMCoflow[10] addresses these problems by proposing the OneCoflow and OMCoflow algorithms. The main idea of these two algorithms is to only rout flows once by using convex programming and rounding.

Distributed

Although the centralized coflow scheduling schemes achieve good performance for large coflows, they generally omit tiny coflows (e.g., size ≤ 1 MB) and cannot handle these coflows due to large centralized control overhead. For example, Varys batches control messages at about a 100 millisecond interval. This interval is too large for tiny coflows and can significantly increase the CCT of tiny coflows. Thus, the tiny coflows are generally treated as background traffic by Varys and Rapier.

To reduce control overhead, the authors in [8] propose the distributed coflow scheduling system Optas for optimizing the CCT of tiny coflows. Optas monitors system calls and backlogged bytes in the send-buffer to identify tiny coflows. After identifying the tiny coflows, all the tiny coflows are sorted according to their start time in ascending order and scheduled in FIFO manner. To further reduce the CCT and avoid head-of-line blocking, tiny coflows are assigned higher priority than the other coflows.

D-CAS [11] is a decentralized solution scheduling all types of coflows. D-CAS proves the coflow scheduling is equivalent to the concurrent open shop with the objective of minimizing the average completion times. Thus, based on the algorithms for solving concurrent open shop, D-CAS proposes the SOA-II algorithm to schedule coflows based on priority. Indeed, D-CAS implements a decentralized negotiation mechanism to negotiate the priorities of coflows calculated by the SOA-II. In addition, because the negotiation mechanism takes a few RTTs to take effect, it has large overheads for tiny coflows. Hence, D-CAS directly gives tiny coflows the highest priority and schedules them following FIFO.

Information-Agnostic Coflow Scheduling

Aalo [4] is the first scheme that tries to schedule coflows without knowing the characteristics of the coflows. It assumes the size of flows in coflows cannot be known prior. Aalo aims to approximate the behavior of the Least-Attained Service (LAS) scheduling discipline and thus proposes the Coflow-Aware Least-Attained Service (CLAS) algorithm. This algorithm assigns each coflow a priority based on sent bytes. Specifically, there are several priority queues, and each queue has a predefined threshold. Initially, a coflow is placed in the highest priority queue. When the sent bytes of the coflow exceed the queue's threshold, the coflow is removed to the lower priority queue. As a result, smaller coflows obtain higher priorities than larger coflows.

However, Aalo still needs to modify cluster computing applications to obtain coflow information. This is unrealistic in public cloud or shared environments. Therefore, CODA [12] is proposed to automatically collect coflow information and schedule coflows without changing cluster computing applications. CODA mainly has two parts: coflow identification and error-tolerant scheduling. The first part identifies flows belong to a specific coflow by leveraging popular unsupervised clustering technology in machine learning. Because clustering has inevitable identification errors, CODA designed an error-tolerant scheduling algorithm to reduce the impact of misidentifications. Like Aalo, this algorithm also leverages priority queues and adjusts the coflow priorities based on sent bytes.

Stream [13] seeks to be readily deployed in data centers by using a decentralized protocol without hardware modifications. This protocol leverages many-to-one and many-to-many coflow patterns to exchange information between machines. The coflow scheduling policy of Stream emulates the conditional Shortest Job Fist (CSJF) heuristic algorithm that is similar to the key idea of Aalo and CODA. However, compared to Aalo and CODA, it improves scalability and can handle the tiny coflows.

Optimize Other Objectives

Previous coflow scheduling schemes mainly focus on reducing the average CCT of all coflows. However, different coflows have different degrees of sensitivity to CCT. For example, recall that coflows in web applications usually have strict deadlines and thus are more sensitive to CCT than coflows in Hadoop.

Chen *et al.* [14] assume coflows have respective utility functions and try to optimize the utility of coflows. In addition to optimizing the utility of coflows, they try to achieve optimal max-min fairness between coflows, which is needed in sharing cloud. Since achieving optimal utility with max-min fairness is NP-hard, Chen *et al.* first solve the subproblem that maximizes the worst utility among all the coflows. After solving the subproblem, the variables of the coflow that achieve the optimal worst utility can be removed from the original problem. As a result, the original problem is solved by repeatedly solving the subproblem and removing variables.

Jahanjou *et al.* [15] assume each coflow has a weight and try to minimize the weighted CCT of coflows. They consider two different models for coflows: circuit-based coflows (a flow is a connection request) and packet-based coflows (a flow is a packet). Further, they design approximation algorithms for scheduling coflows with or without given flow paths by using the two models. If the flow paths are not given, their first approximation algorithms can schedule coflows over general network topologies.

Challenges and Broader Perspective

In this section, as shown in Table 1, we compare coflow scheduling mechanisms based on their main features.

Challenges of Coflow Scheduling

Objectives: Users usually have different demands for networks, thus the coflow scheduling problem may have various optimization objectives, such as minimizing CCT, maximizing utilization of networks, meeting deadlines and guaranteeing Quality of Service(QoS). However, Table 1 shows that existing mechanisms mainly focus on reducing the average CCT or weighted CCT of coflows, and they are unable to optimize other objectives. For example, one of the most important objectives is providing differentiated service among coflows. Generally, web search services have deadlines, while data analysis jobs have no deadlines. Although Jahanjou *et al.* and Chen *et al.* give high weight to time-sensitive coflows, they are not giving explicit priority to coflows and cannot guarantee that time-sensitive coflows are always scheduled in the highest priority. Thus, how to assign priorities and schedule coflows according to priorities are still unsolved.

Scalability: Due to large overheads, centralized proposals, such as Orchestra, Varys and RAPIER, are unable to handle tiny coflows. On the other hand, although distributed schemes like Optas, D-CAS and Stream can handle tiny coflows, they have sub-optimal performance since they cannot obtain global coflow information in a timely manner. Thus, the challenge is to improve scalability while still obtaining optimal performance. Jointly leveraging distributed and centralized schemes may address the problem and become a promising research direction. We may design an algorithm to identify tiny coflows and decide the optimal number of tiny coflows handled by distributed schemes to increase scalability without sacrificing performance.

Deployment: Considering that more and more cluster computing applications are moving to the cloud, it will be significant to explore whether coflow scheduling mechanisms can be easily deployed in cloud environments. The first challenge of deployment is how to obtain coflow information without application modifications. As we have discussed, CODA has already taken this into consideration. However, CODA heavily relies on historical coflow information to identify coflows, and if workloads change, it may have worse performance. The second challenge is how to schedule coflows without application and hardware modifications. To address this problem, Stream implements its algorithms in the stack of operating systems to avoid modifying the cluster

There are many different optimization objectives for coflow scheduling, and it is hard to design coflow scheduling mechanisms for different objectives. By using machine learning, the computer can learn various optimization objectives and then generate management policies accordingly.

Proposals	Objective	Distributed	Routing	Required information	Inter-coflow	Intra-coflow	Application modification	Scalability	Performance
Orchestra [2]	CCT for (broadcasting or shuffle)	No	No	All coflow information	FIFO	Weighted	Yes	Low	Low
Varys [3]	CCT or deadline	No	No	All information	SEBF	MADD	Yes	Medium	Medium
Rapier [9]	CCT	No	Yes	All coflow, network info.	SJF	Optimization	Yes	Low	High
OMCoflow[10]	CCT	No	Yes	All coflow, network info.	Weighted sharing	Optimization	Yes	Low	High
Optas [8]	CCT for tiny coflow	Yes	No	Coflow structure	FIFO-based	FIFO-based	No	High	Medium
D-CAS [11]	CCT	Yes	No	All coflow information	SJF	SJF	Yes	High	Medium
Aalo [4]	CCT	No	No	Coflow structure	LAS	Fair sharing	Yes	Medium	Medium
CODA [12]	CCT	No	No	None	LAS	LAS	No	Medium	Medium
Stream [13]	CCT	Yes	No	Coflow structure	LAS	LAS	No	High	Medium
Chen *et al.*, [14]	Utility, fairness	No	No	All coflow information	Liner programming	Liner programming	Yes	Low	High
Jahanjou *et al.*, [15]	Weighted CCT	No	Yes	All coflow, network info.	Liner programming	Liner programming	Yes	Low	High

Table 1. Summary of coflow scheduling proposals and their main designs.

computing applications. However, Stream cannot automatically identify coflows, and it still needs to obtain the information from applications. Therefore, the above two problems of deployment are not addressed perfectly, and they are still challenges for coflow scheduling.

Coexisting with Load Balancing: Load balancing aims to equally balance the load at flow-level between multiple links to minimize flow completion time (FCT). Compared to coflow scheduling, its target is individual flows. In data centers, flows usually coexist with coflows. Thus, coflow mechanisms need to coexist with load balancing. However, most coflow schemes assume the network is a big switch and do not consider the impacts of load balancing. Since flows compete for bandwidth with coflows, instead of optimizing the two problems separately, we should consider how to schedule flows and coflows at the same time.

Broader Perspective

Multi-Tenant Data Centers: Multi-tenant data centers allow tenants to run different applications on the same physical resources. Providers need to guarantee resource isolation and sharing of bandwidth among different tenants. While bandwidth isolation is relatively easier, isolation in coflow scheduling is challenging. Basically, it is easy to isolate bandwidth at each switch or link. However, in coflow scheduling, isolation should be realized at coflow-level that coflows of different users weighted share the resources. Existing schemes based on SJF or LAS prefer to give small coflows high priority. As a result, if a user always generates small coflows while another user always generates large coflows, the fairness between the two users cannot be guaranteed. Therefore, coflow scheduling in multi-tenant data centers can be a promising research direction.

Machine Learning: As machine learning evolves, it becomes a promising approach to improve the performance of coflow scheduling. Machine learning technologies, such as deep learning and reinforce learning, allow the computer to learn control policies automatically. As we have discussed before, there are many different optimization objectives for coflow scheduling, and it is hard to design coflow scheduling mechanisms for different objectives. By using machine learning, the computer can learn various optimization objectives and then generate management policies accordingly. Actually, machine learning technologies have been used to learn management policies for allocating CPU and memory resources of computers.

Conclusion

In this work, we presented a survey of coflow scheduling. We provided an overview of the characters of coflow and classified the proposed approaches. We observed that existing proposals focus on minimizing CCT and neglect other objectives. In addition, their major designs are based on the SJF heuristic, which has sigificant

room for improvement. We observed that most schemes need to modify applications to identify coflows and schedule coflows, thus it is a challenge to deploy them in cloud environments. Finally, we described some challenges and made a broader perspective. With the development of multi-tenant data centers, we found isolation at coflow-level may become a promising research direction. In summary, research on coflow scheduling is quite broad and a number of research issues and challenges are ahead.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (NSFC) under Grant No. 61502049, the National High-Tech Research and Development Plan of China (863 Plan) under Grant No. 2015AA016101, the Young Talent Development Program of the CCF, the Young Elite Scientist Sponsorship Program by CAST under Grant No. 2015QNRC001, the Beijing New-star Plan of Science and Technology under Grant No. Z151100000315078, and the 111 project (NO.B17007).

References

[1] M. Chowdhury and I. Stoica, "Coflow: A Networking Abstraction for Cluster Applications," *ACM HotNets*, 2012, pp. 31–36.
[2] M. Chowdhury *et al.*, "Managing Data Transfers in Computer Clusters with Orchestra," *ACM SIGCOMM*, 2011, pp. 98–109.
[3] M. Chowdhury, Y. Zhong, and I. Stoica, "Efficient Coflow Scheduling with Varys," *ACM SIGCOMM*, 2014, pp. 443–54.
[4] M. Chowdhury and I. Stoica, "Efficient Coflow Scheduling without Prior Knowledge," *ACM SIGCOMM*, 2015, pp. 393–406.
[5] R. Rojas-Cessa, Y. Kaymak, and Z. Dong, "Schemes for Fast Transmission of Flows in Data Center Networks," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 3, pp. 1391–1422, 2015.
[6] Z. Fu *et al.*, "Seagull – A Real-Time Coflow Scheduling System," *IEEE CSCloud*, 2015, pp. 540–45.
[7] F. R. Dogar *et al.*, "Decentralized Task-Aware Scheduling for Data Center Networks," *ACM SIGCOMM*, 2014, pp. 431–42.
[8] Z. Li *et al.*, "OPTAS: Decentralized Flow Monitoring and Scheduling for Tiny Tasks," *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[9] Y. Zhao *et al.*, "RAPIER: Integrating Routing and Scheduling for Coflow-aware Data Center Networks," *Proc. IEEE INFOCOM*, 2015.
[10] Y. Li *et al.*, "Efficient Online Coflow Routing and Scheduling," *ACM MobiHoc*, 2016, pp. 160–73.
[11] S. Luo *et al.*, "Towards Practical and Near-Optimal Coflow Scheduling for Data Center Networks," *IEEE TPDS*, vol. 27, no. 11, Nov. 2016, pp. 3366–80.
[12] H. Zhang *et al.*, "CODA: Toward Automatically Identifying and Scheduling Coflows in the Dark," *ACM SIGCOMM*, 2016, pp. 160–73.
[13] H. Susanto, H. Jin, and K. Chen, "Stream: Decentralized Opportunistic Inter-Coflow Scheduling for Datacenter Networks," *IEEE ICNP*, 2016, pp. 1–10.
[14] L. Chen *et al.*, "Optimizing Coflow Completion Times with Utility Max-Min Fairness," *Proc. IEEE INFOCOM*, 2016, pp. 1–9.
[15] H. Jahanjou, E. Kantor, and R. Rajaraman, "Asymptotically Optimal Approximation Algorithms for Coflow Scheduling," *ACM SPAA*, 2017, pp. 45–54.

Biographies

Shuo Wang [S] (shuowang@bupt.edu.cn) is working toward the Ph.D. degree at the State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. He received a B.S. degree in communication engineering from Zhengzhou University, China. His current research interests include data center networks, load balancing, and software-defined networking.

Jiao Zhang [M] (jiaozhang@bupt.edu.cn) is an associate professor in the School of Information and Communication Engineering and the State Key Laboratory of Networking and Switching Technology at Beijing University of Posts and Telecommunications. In July 2014, she received her Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, China. From August 2012 to August 2013, she was a visiting student in the networking group of ICSI, UC Berkeley. In July 2008, she obtained her B.S. degree from the School of Computer Science and Technology at BUPT. Her research interests include traffic management in data center networks, software-defined networking, network function virtualization, future Internet architecture and routing in wireless sensor networks. She has (co)-authored more than 20 international journal and conference papers.

Tao Huang [M] (htao@bupt.edu.cn) received his B.S degree in communication engineering from Nankai University, Tianjin, China, in 2002, and the M.S. and Ph.D. degrees in communication and information systems from Beijing University of Posts and Telecommunications, Beijing, China, in 2004 and 2007, respectively. He is currently a professor at Beijing University of Posts and Telecommunications. His current research interests include network architecture, routing and forwarding, and network virtualization.

Jiang Liu [M] (liujiang@bupt.edu.cn) received his B.S. degree in electronics engineering from Beijing Institute of Technology, China, in 2005, his M.S. degree in communication and information systems from Zhengzhou University, China, in 2009, and his Ph.D. degree from BUPT in 2012. He is currently an associate professor at BUPT. His current research interests include network architecture, network virtualization, software-defined networking, information-centric networking, and platforms for networking research and teaching.

Tian Pan [M] (pan@bupt.edu.cn) received his B.S. degree from Northwestern Polytechnical University, Xi'an, China, in 2009, and the Ph.D. degree from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 2015. He has been a post-doctoral researcher with Beijing University of Posts and Telecommunications since 2015. His research interests include router architecture, network processor architecture, network power efficiency, and software defined networking.

Yunjie Liu (liuyj@bupt.edu.cn) received his B.S degree in technical physics from Peking University, Beijing, China, in 1968. He is the academician of the China Academy of Engineering, the chief of the science and technology committee of China Unicom, and the dean of the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications. His current research interests include next generation networks, network architecture and management.

With the development of multi-tenant data centers, we found isolation at coflow-level may become a promising research direction. In summary, research on coflow scheduling is quite broad and a number of research issues and challenges are ahead.