MXDAG: A Hybrid Abstraction for Emerging Applications

Weitao Wang, Sushovan Das, Xinyu Crystal Wu, Zhuang Wang, Ang Chen, T. S. Eugene Ng
Rice University

Abstract

Emerging distributed applications, such as microservices, machine learning, big data analysis, consist of both compute and network tasks. DAG-based abstraction primarily targets compute tasks and has no explicit network-level scheduling. In contrast, Coflow abstraction collectively schedules network flows among compute tasks but lacks the end-to-end view of the application DAG. Because of the dependencies and interactions between these two types of tasks, it is sub-optimal to only consider one of them. We argue that co-scheduling of both compute and network tasks can help applications towards the globally optimal end-to-end performance. However, none of the existing abstractions can provide fine-grained information for co-scheduling. We propose MXDAG, an abstraction to treat both compute and network tasks explicitly. It can capture the dependencies and interactions of both compute and network tasks leading to improved application performance.

ACM Reference Format:
Weitao Wang, Sushovan Das, Xinyu Crystal Wu, Zhuang Wang, Ang Chen, T. S. Eugene Ng. 2021. MXDAG: A Hybrid Abstraction for Emerging Applications. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21), November 10-12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3484266.3487384

1 Introduction

Today's compute systems host a variety of distributed applications ranging from microservices, MapReduce, to distributed DNN training and database systems [23, 34, 38, 39]. Due to the distributed nature of such applications, the network can easily become the bottleneck [26, 32, 46]. Better scheduling decisions are key to application performance and resource utilization. Task-level DAGs (directed acyclic graphs) are a common abstraction for scheduling [3, 20, 33, 43], but many such systems do not explicitly consider network resources. Network-aware DAG scheduling [1, 16, 17] does take network bandwidths into account, but in a resource packing model; it does not explicitly schedule network communications.

Another group of work focuses on explicit network scheduling and job placement where the primary objective is to localize most of the traffic flows between tasks and balance the network utilization across the system [2, 5, 8, 11, 21, 37, 47]. Such frameworks have more fine-grained information about the network I/O, but lack tight integration between network flows and application-level requirements. The coflow abstraction [6] tries to bridge this gap by jointly considering collections of network flows among multiple compute stages, which enables application-aware network scheduling to some extent [7, 9]. However, coflows do not consider a global view of the application DAG. As we will discuss later, defining a coflow inside an asymmetric DAG can be ambiguous. Finally, coflow treats the internal flows based on an all-or-nothing principle, which can obscure the critical path information and harm application performance.

Therefore, *co-scheduling* of both compute and network tasks is necessary to improve application performance and resource utilization. Fundamentally, this explicitly considers all kinds of dependencies from the application (compute-network, compute-compute, network-network) in a more fine-grained manner. This enables better critical path analysis and scheduling strategies, e.g., by overlapping communication with computation, chunking up data for pipelining, and preempting and ordering parallel flows.

We ideally want a co-scheduler that optimally schedules the application given the fine-grained information regarding both the compute and network tasks with an end-to-end view. Unfortunately, none of the existing abstractions can encode such fine-grained information effectively. This inherent gap between the ideal co-scheduling requirements and existing abstractions motivates us to propose a more general and fundamentally different abstraction, called MXDAG. It abstracts both the compute and network tasks in a DAG as explicit nodes with annotations. The edges capture task/network dependencies. MXDAG can potentially address several challenges related to co-scheduling system design. First, by decoupling compute and network tasks, MXDAG enables the co-scheduler to treat them uniquely as they have fundamentally different behaviors. Compute tasks are easily isolated among

HotNets '21, November 10-12, 2021, Virtual Event, UK

ACM ISBN 978-1-4503-9087-3/21/11…$15.00
https://doi.org/10.1145/3484266.3487384

CPU/GPU cores but the performance is less predictable. Network tasks are more predictable given the data size and network bandwidth are known, but they cannot be isolated as easily. Second, MXDAG enables the co-scheduler to consider the heterogeneity in both compute (CPU, GPU) and network resources [22, 29, 30]. Finally, with a global view, MXDAG enables the co-scheduler to carefully analyze the impact of pipelining between all kinds of tasks for better decisions.

2 Motivation

2.1 Previous DAGs Lack Explicit Network Scheduling

DAG-based abstraction is widely used to analyze and optimize parallel jobs. It captures data flows between computational tasks and indicates corresponding dependencies, which plays a significant role in resource sharing and task scheduling. However, most of the existing DAG-based scheduling frameworks, including Spark, Flink, Dryad, and Tez [3, 20, 33, 43], mainly focus on the host-level computational tasks and implicitly treat the network requirements as parts of the computational tasks. Several recent network-aware DAG-based schedulers [1, 16, 17] start to consider the bandwidth resource. Nevertheless, they only consider the bandwidth when packing different resources while no explicit flow-level resource scheduling information is included. Thus, these DAG abstractions usually use the same type of edges, without thoroughly distinguishing between logical dependencies and real data transmissions, leading to inefficient scheduling results.

For example in Fig. 1(a), host A needs to send two flows to host B and C. A network-aware scheduler would fairly share the bandwidth resources and schedule the tasks as in Fig. 1(b), flow 1 and flow 3 will share the NIC bandwidth and thus extend the completion time. As a result, job X can only complete at time T1. Instead, a network-computation co-scheduler would schedule in a globally optimal way by prioritizing flow 1 over flow 3 as Fig. 1(c), so that both the flows enjoy the full bandwidth. Therefore, the task on C could complete at time T2, much earlier than the previous case.

2.2 Coflow Abstraction Lacks Global View

The Coflow abstraction [6] was proposed a decade ago and is widely used by many network schedulers to optimize resource sharing [7, 9, 24]. Such abstraction jointly considers parallel flows between two subsets of hosts having a common objective [6] and also contains the information about the communication pattern e.g., broadcast (one-to-many), aggregation (many-to-one), shuffle (many-to-many), etc. However, coflow abstraction has two fundamental limitations:

Coflow abstraction implicitly assumes symmetry in the DAG, which leads to definitional ambiguity when abstracting asymmetric DAG. Asymmetric DAG is common in emerging applications [4, 25, 35, 41]. From those, we primarily observe two sources of asymmetry: 1) The asymmetry can arise from the heterogeneity in computation time across the nodes, as

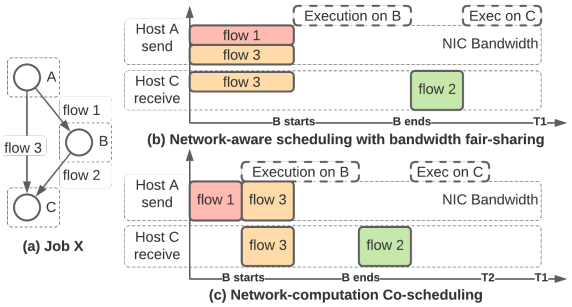

Figure 1: Comparison between network-aware scheduling and network-compute co-scheduling.

shown in Fig. 2(a). The computation times for tasks on host B and C can be unequal (t_1 and t_2 respectively) due to the heterogeneity in the underlining hardware (GPU, CPU, etc.) or the different task sizes; 2) The DAG can also have an asymmetric topology as shown in Fig. 2(b) (adopted from [4]). From which, three different coflow abstractions might be derived as Fig. 2(b1,b2,b3). In Fig. 2(b1), we consider two coflows i.e., broadcast from node C (f_3 and f_4) and aggregation at node F (f_5 and f_6). In Fig. 2(b2), aggregation at node E (f_2 and f_4) is considered to be an alternative coflow. In Fig. 2(b3), all flows between nodes {B,C} and {D,E} are considered to be one coflow (f_2, f_3 and f_4). Despite having several options for defining a coflow, the application programmer must commit to a specific definition while writing the application. It cannot be modified during runtime. Most importantly it is difficult to predict, how a specific coflow definition would impact the application performance.

Without a global view of the DAG, the coflow abstraction could lead to inefficient scheduling. By enforcing all the flows in a coflow to end at the same time, coflow can possibly obscure the critical path information of the DAG, which may lead to sub-optimal performance during scheduling, as shown in Fig. 2(c) and (d). For the DAG with asymmetric computation time, the optimal scheduling without coflow in Fig. 2(c) treats each flow individually and allows each flow to avoid sharing the NIC bandwidth resources smartly. While with the coflow abstraction, coflow $\{f_1, f_2\}$ and $\{f_3, f_4\}$ have to share the NIC bandwidth at the same time and enlarge the end-to-end completion time. For the DAG with asymmetric topology, The optimal solution delays the start time for f_4 and avoids NIC bandwidth sharing at host C (source). And as a cascading effect, f_5 and f_6 also do not share the NIC bandwidth at host F (destination). Whereas, the three different solutions with coflow abstraction all have sub-optimal scheduling. In Fig. 2(d), the coflow abstraction b1 forces the coflows $\{f_3, f_4\}$ and $\{f_5, f_6\}$ to share the NIC bandwidth on hosts C and F respectively, so that the execution on D will be postponed. Meanwhile, the coflow abstractions b2 and b3 also force the scheduler to schedule f_2 and f_4 together as one coflow, leading to NIC bandwidth sharing at host E.

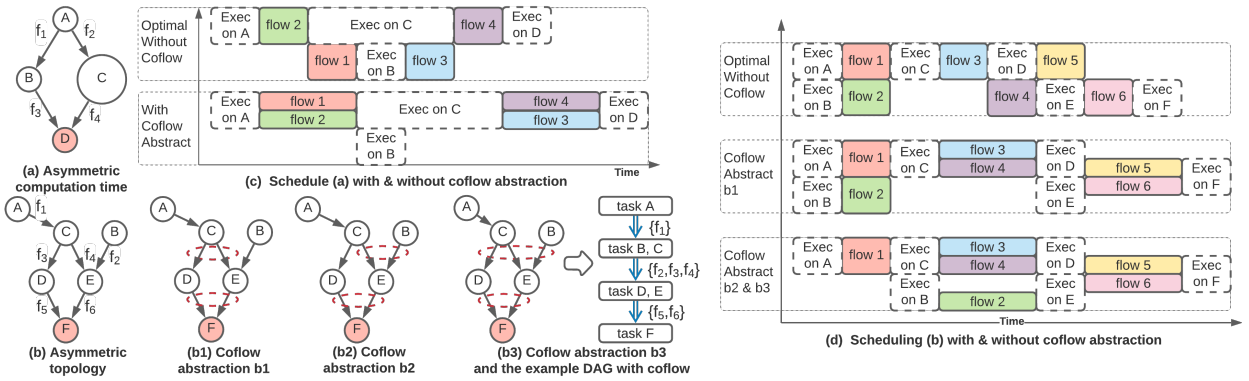

Figure 2: (a) DAG with asymmetric computation times; (b) DAG with asymmetric topology from Wukong [4]; (b1,b2,b3) Three potential coflow abstractions of DAG in (b), and flows grouped with dashed ellipse are considered as a coflow; (c) An optimal schedule without coflow for DAG in (a) and the schedule with $\{f_1, f_2\}$ and $\{f_3, f_4\}$ as coflows; (d) An optimal schedule without coflow abstraction for DAG in (b) and schedules with different coflow abstraction.

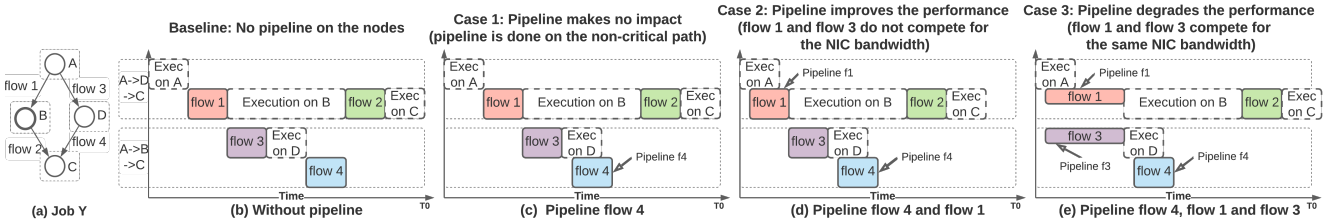

Figure 3: Different pipelineablity choices make different impacts on application performance

2.3 Both Abstractions Lack Pipelineability Analysis

Pipelining is a promising strategy to improve the performance of distributed applications. By chunking up the data flows, not only the storage usage on the host can be reduced, but also the overlap between communication with computation can be maximized. There are two common kinds of applications that could be optimally scheduled with efficient pipelining. On one hand, map-reduce jobs could significantly reduce the job completion time by pipelining the execution of the map and reduce tasks [10]. On the other hand, distributed deep learning, especially the gradient aggregation part, can benefit a lot from pipelining the push and pull operations, thereby significantly reducing the communication time and accelerating the overall training [32].

However, none of the existing DAG-based and coflow-based abstractions fully consider the pipelineability in their scheduling strategies [6, 17]. Caerus [44] does consider pipelineability and provides a step dependency model to capture the pipeline information. Nevertheless, it only profiles the pipelineability on the computational tasks, without any network-level pipelining analysis. Therefore, such network-oblivious pipelining could lead to sub-optimal scheduling decisions. We analyze several situations where pipelineablity has different impacts, using a four-node DAG with A->B->C as the critical path, as shown in Fig. 3(a). Fig. 3(b) displays the execution timeline of the baseline situation where pipelines are not allowed anywhere. We then illustrate three different scenarios with different pipelineablity choices as follows to provide insights into pipelining impacts. With these insights, we could observe that a better scheduler should allow network operators to choose whether to use pipeline or not and which tasks need to be pipelined at runtime.

Case 1: Pipelining on the non-critical path makes no impact on the application performance. As shown in Fig. 3(c), pipelining flow 4 on node D will not affect the length of the critical path since node D does not belong to it. Therefore, the execution on C will be the same as the baseline case and there are no changes to the overall application performance.

Case 2: Pipelining on the critical path can improve the application performance. As shown in Fig. 3(d), besides pipelining flow 4 as the previous case, flow 1 on node A is also chosen to be pipelined. Since flow 3 still starts after flow 1 is completed, these two flows will not overlap and enjoy the full NIC bandwidth of node A. Therefore, such pipelining strategies will shorten the critical path length (A->B->C), causing the execution on node C to start earlier than in the baseline.

Case 3: Pipelining on the critical path can degrade the application performance. As shown in Fig. 3(e), allowing to pipeline flow 3 along with flow 1 and flow 4, will increase the critical path length. In this case, flow 1 and flow 3 start at the same time and take twice the time to finish as they share the same NIC bandwidth of node A. Therefore, the length of the critical path (A->B->C) becomes longer that causes the execution on node C to start later than in the baseline.

3 MXDAG

To address the above drawbacks of the existing solutions, we introduce the MXDAG abstraction. The construction of MXDAG can still rely on existing solutions to get the necessary information for different kinds of applications. On one hand, there are bare-metal applications like distributed deep learning and distributed matrix computation, where all the necessary information (e.g., CPU/GPU cores, data size, NIC bandwidth) can be provided explicitly before execution. On the other hand, for the applications running on Spark, Hadoop, Dryad etc., such information is not known *a priori*, because the physical placement is decided by system schedulers (e.g., YARN) at runtime. For these cases, flow-level details can be estimated from historical placement information [18, 31, 44] and the execution time of a compute task on specific hardware can be estimated by measurement-based job profiling [36, 40, 42, 45].

3.1 Definition

MXTasks are the nodes $\{v_1, v_2, ..., v_k\}$ in the MXDAG G, and each MXTask represents either a task running on a host using CPU/GPU/accelerator or a flow in the network with a single sender and receiver. Note that all the MXTasks are physical processes or flows, rather than logical tasks which usually contain multiple physical tasks on multiple machines. Each MXTask is a procedure that receives an input and gives an output after being processed by a certain amount of resources. To include more quantitative information, each MXTask has two additional fields: 1) MXTask size $Size(v_i)$ represents the completion time of an MXTask with the maximum resource assigned (computation or bandwidth), which has similar meaning with the concept of task durations in Decima [27] and Graphene [17]. Size information can be used to estimate the completion time when only partial resources are assigned to the task; 2) MXTask unit $Unit(v_i)$ represents the size of the smallest unit when being executed under the pipeline. Note that for MXTasks that cannot be executed in a pipeline, its unit size is equal to its task size.

MXDAG is a directed graph $G = (V, E)$ composed of MXTasks $V = \{v_S, v_1, v_2, ..., v_k, v_E\}$ and dependencies represented as $E = \{e_1, e_2, ..., e_i\}$. v_S and v_E are the dummy start and end MXTasks in an MXDAG. An edge from v_i to v_j indicates that task v_j cannot start before v_i starts. MXDAG serves as an abstraction for a complete application or an individual function within an application, the latter being very common in serverless environments. For instance, the MXDAGs for job X and job Y are shown in Fig. 4(a) and (b). Different from existing DAG-based systems, MXDAG elevates the network flows to the same level as the computational tasks on the hosts. Therefore, MXDAG can provide detailed information as well as the importance of each network flow, figuring out the relative priorities and achieving better scheduling strategies.

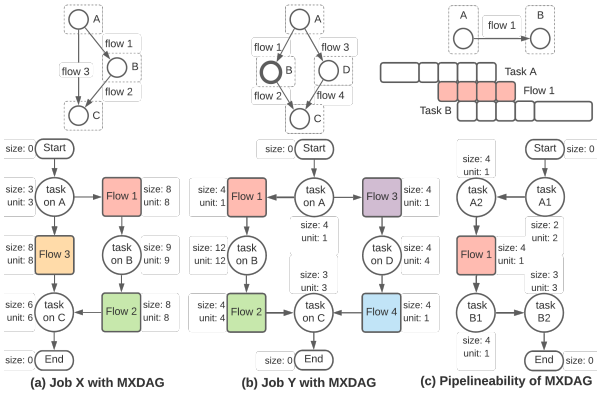

Figure 4: (a, b) MXDAGs for jobs X and Y; (c) MXDAG with only partial parts of tasks A and B being pipelined.

Pipelineability. To include the pipelineability of MXTasks, pipelineable MXTask divides its input and output into minimum units. Namely, once an input unit is received, that MXTask can start processing and immediately give an output unit as shown in Fig. 5. For the network MXTasks, as long as the output of the predecessor CPU task can be given in units, the pipelineability can be enabled instead of waiting until all outputs are ready (e.g., sending serialized objects like hash tables). While for the computational tasks, we rely on the existing pipeline analysis works, like the per-step dependency model in Caerus [44], to profile the pipelineability in the computational tasks. For the computational tasks with both pipelineable part and sequential-execution-only part, two MXTasks will be derived as the task A and B in Fig. 4(c).

3.2 Properties of MXDAG

Firstly, we will introduce several notations and properties of MXDAG that are useful for the following discussions. *Path* in the MXDAG denotes a finite sequence of edges that join a sequence of MXTasks with a head task (node) and a tail task. *Copath* denotes a group of paths with the same head node and tail node, as the path A->f_1->B->f_2->C and A->f_3->C in job X of Fig. 4(a). The *Path Length*, representing the end-to-end computation time for a path, is calculated recursively in an MXDAG: 1) divide a path into Copaths and normal paths without Copath, and its length is treated as the sum of normal path lengths and Copath lengths; 2) For a Copath, its length is equal to the length of its longest path; 3) For a normal path, its length can be calculated as the sum of the pipelineable-only paths and sequential-only paths. The length of a sequential-only path $P_{seq} = \{v_0, v_1, ..., v_m\}$ and a pipelineable-only path $P_{pipe} = \{v_0, v_1, ..., v_n\}$ can be calculated as below (Given the resource assigned to each MXTask v_i as $Rsrc(v_i)$):

$$Len(P_{seq}) = \sum_{i=0}^{m} \frac{Size(v_i)}{Rsrc(v_i)} \tag{1}$$

$$Len(P_{pipe}) = \sum_{i=0}^{n} \frac{Unit(v_i)}{Rsrc(v_i)} + \max_i\{\frac{Size(v_i)}{Rsrc(v_i)}\} - \max_i\{\frac{Unit(v_i)}{Rsrc(v_i)}\} \tag{2}$$

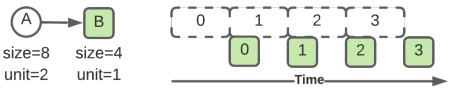

Figure 5: Example pipeline for two pipelineable MX-Tasks with different task sizes and task unit sizes.

Equation (2) implies that the length of a pipelineable-only path is dominated by the pipelineable task with the longest execution time as shown in Fig. 5. Also, the maximum throughput of the flow can be restricted by the CPU processing speed when the pipeline is used.

Another important property is that the paths within any Copath have the same barriers so that every Copath has a critical path. With these barriers, only all of the paths within that Copath have finished execution, or given the first unit of result in a pipeline, that tail node can start the execution. We define the path with the longest length in a Copath as its critical path, then the length of the critical path determined the overall execution time of a Copath.

4 Using MXDAG

MXDAG provides essential network task information to more precisely reveal the end-to-end critical path, with which better scheduling decisions become possible. Applications whose critical paths contain many network tasks or large network transfers, and applications where paths have similar lengths but some can become bottlenecks due to network transfers, would benefit most from MXDAG.

A richer abstraction raises concerns with respect to scheduling latency. For many applications, MXDAG only adds a modest number of nodes for data transfers in the DAGs. In the case of an all-to-all shuffle traffic pattern, there could be a sizeable number of network task nodes. To control scheduling latency, we plan to investigate several approaches: 1) Prune unimportant paths to focus on the most plausible critical paths, since the critical path determines the end-to-end completion time for the job; 2) Solve the full scheduling problem optimally only when necessary, and incrementally optimize scheduling at runtime based on the resource usage changes and the remaining tasks as in [14]; 3) Use a data-driven approach with the help of machine learning, inspired by [27], which could potentially give a good scheduling plan within a short time. 4) Reuse scheduling solutions for recurring applications [44] so that the overhead introduced by scheduling only affects the first iteration. We leave the concrete scheduling algorithms as future work and focus on formulating a more powerful abstraction, the MXDAG, in this paper.

4.1 Schedule a Single MXDAG

Single MXDAG scheduling aims to minimize the job completion time (JCT) considering both computational and network tasks. The collective objective of all the paths in the MXDAG can be expressed as:

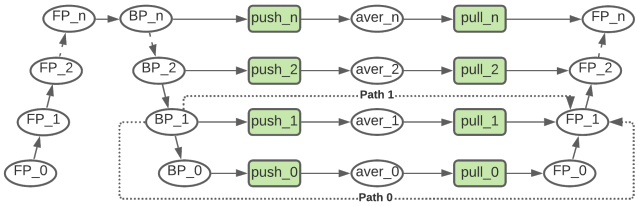

Figure 6: MXDAG for Distributed Machine Learning

$$\min \max_{P \in \mathcal{P}} \{Len(P)\}$$
$$\text{where } \mathcal{P} = \{P | Head(P) = v_S, Tail(P) = v_E\} \quad (3)$$

To achieve the above objective, we will use MXDAG to analyze the inherent dependencies and resource sharing between the MXTasks. Since the optimal scheduling for MXDAG is an NP-hard problem [12, 17, 28], we will give several principles to guide the scheduling and inspire new heuristics, leaving the detailed algorithm as future work.

Principle 1: Prioritize the critical path over non-critical paths within any Copath, without letting the non-critical paths have a longer completion time than the critical path.

If the different paths within a Copath share some resources, like the NIC bandwidth or the CPU cores, delaying the resource allocation for the non-critical paths or allocating fewer resources to the non-critical paths could help shrink the critical path completion time. By ensuring the non-critical paths have a shorter or equal completion time with the critical path, over-optimization can be avoided. Notably, though the pipeline can be used to shrink the delay between two tasks, it also enforces the resources to be occupied right after the precedent task begins processing, which may contend with the tasks on the critical path. Thus, even for pipelineable MX-Tasks, the pipelines will only be applied when they can shrink the overall execution time.

Example Case: Distributed Deep Learning. We take a widely-used and increasing-important application as an example, the data-parallel distributed deep learning. The communication overhead of synchronizing the parameters on different machines is significant to the data-parallel distributed learning workloads, but transmitting the parameter in layers can shrink the overall completion time. To explain the idea of layer-wise parameter synchronization, Fig. 6 shows the MXDAG for that process: the parameters of each layer are generated after the back-propagation (BP) process on the GPU, and the synchronized parameters will be used by the forward-propagation (FP) process in the next iteration. For neural networks with multiple layers, the FP processes are executed from the lower layer to the higher layer, and the BP processes are executed in a reverse manner.

Take path 0 and path 1 for example, the MXTasks $push_i$ and $pull_i (i \in \{0, 1\})$ share the same bandwidth resources over the network. Consider the Copath between BP_1 and FP_1, if path 0 is the critical path, then all the bandwidth resource

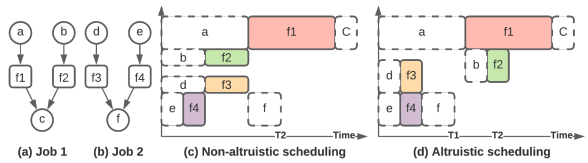

Figure 7: Schedule multiple Map-Reduce Jobs. MXTasks b and d share the same computational resource and MXTasks f_2 and f_3 share the same NIC bandwidth resource.

should be allocated to path 0 to achieve the shortest completion time. Whereas, if the path 1 is the critical path, as long as the FP_0 finishes earlier than the $pull_1$, there is no strict ordering for the resource allocation between path 0 and path 1. (Note that strictly prioritizing the path 0 is optimal scheduling within the above solution space.) ByteScheduler [32] rearranged the tensor transmission order—i.e., strictly prioritize the parameters in the lower layers over those in the upper layers, which accelerates the training process. Our analysis over MXDAG exactly echoes their solution.

4.2 Schedule Multiple MXDAGs

Besides minimizing the JCT, scheduling multiple MXDAGs has more objectives, such as meeting the deadline of each job and ensuring fairness among all the MXDAGs. Since the key of multiple MXDAG scheduling is also resource sharing, here we give our second principle to guide the resource allocation over multiple MXDAGs.

Principle 2: Let each MXDAG be altruistic by delaying its resource allocation on non-critical paths to benefit other MXDAG's critical path, without increasing its own JCT.

Once the resource allocation for the critical path has been determined inside the MXDAG, the overall JCT is certainly bounded with the execution time of the critical path. Thus, as long as other MXTasks on the non-critical path finish earlier than the critical path, the shortest JCT is preserved. With this idea, we allow the scheduler to delay the resource allocation for those non-critical MXTasks, since the resources saved during those waiting times, can be allocated to other application's critical MXTasks for a shorter JCT.

Example Case: MapReduce Applications. For the example MapReduce jobs in Fig. 7, MXTasks a and f_1 have longer computation time than MXTask b and f_2. While the MXTask d and f_3 from job 2 share the same computational / bandwidth resource with b and f_2 from job 1.

Without altruistic scheduling, task d and b, f_2 and f_3 will share the resource lead to a longer completion time for job 2 in Fig. 7(c). However in Fig. 7(d), with principle 2, though job 1 cannot benefit itself from delaying the resource allocation for b and f_2, its altruistic behavior helps job 2 to finish earlier from T2 to T1 by shrinking the critical path in the job 2's MXDAG. This scheduling plan is also compatible with another job-scheduling project's solution—CARBYNE [16].

5 Related Work

On one hand, network-aware DAG schedulers [13, 15, 17] modify the DAG abstraction to treat network bandwidth as a dividable resource and provide greedy heuristics to efficiently pack the tasks. There is no explicit scheduling of network flows due to the lack of flow-level information. On the other hand, explicit network schedulers [7, 9, 11] fundamentally consider Coflow abstraction and perform application-aware network scheduling. Despite scheduling the network flows explicitly, they lack the global view of the application DAG which makes the critical path information elusive. Recent work [19] extends traditional DAG abstraction and glues that with Coflow. Although such extended abstraction provides a slightly better way to capture the compute-network dependencies, the fundamental limitations of both DAG and Coflow abstractions still remain. It does not decouple compute and network tasks explicitly. Also, there is no provision of prioritizing the flows inside a coflow which could potentially benefit the application. In contrast, MXDAG decouples the compute and network tasks, captures dependencies in a more fine-grained manner with an end-to-end application view, and enables explicit co-scheduling. As our abstraction can better characterize applications, it can potentially benefit more recently proposed deep neural network schedulers [27].

6 Concluding Remarks

While this paper has motivated the MXDAG primarily from the perspective of scheduling, there are other use cases to be explored in the future.

What-if analysis on distributed applications. MXDAG can be used to conduct a what-if analysis on the applications, including whether to pipeline compute and network tasks, whether to re-partition work among compute and network tasks, which are not possible with traditional DAG. For instance, an application developer can use the MXDAG of an application to determine whether a revised application design that enables pipelining between previously non-pipelined MXTasks can help shrink the end-to-end completion time.

Monitoring and debugging distributed applications. The estimated task execution time may be different from the actual execution time due to inaccurate data or unexpected events during runtime. By monitoring the progress of each path and the barriers in MXDAG, we can efficiently and accurately identify the unexpected events and the corresponding host straggler or network straggler, while traditional DAG cannot distinguish those two kinds of stragglers. Moreover, operators could leverage the current progress and determine the new critical paths to optimize the scheduling plan at runtime.

We thank our shepherd Aurojit Panda and the anonymous reviewers for their valuable feedback. This research is sponsored by the NSF under CNS-1718980, CNS-1801884, and CNS-1815525.

References

[1] S. Blagodurov, A. Fedorova, E. Vinnik, T. Dwyer, and F. Hermenier. Multi-objective job placement in clusters. In *IEEE SC*, 2015.

[2] P. Bodík et al. Surviving failures in bandwidth-constrained datacenters. In *ACM SIGCOMM*, 2012.

[3] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[4] B. Carver, J. Zhang, A. Wang, A. Anwar, P. Wu, and Y. Cheng. Wukong: a scalable and locality-enhanced framework for serverless parallel computing. In *ACM SoCC*, 2020.

[5] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *ACM SIGCOMM Computer Communication Review*, 2013.

[6] M. Chowdhury and I. Stoica. Coflow: A networking abstraction for cluster applications. In *ACM HotNets*, 2012.

[7] M. Chowdhury and I. Stoica. Efficient coflow scheduling without prior knowledge. *ACM SIGCOMM Computer Communication Review*, 2015.

[8] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica. Managing data transfers in computer clusters with orchestra. *ACM SIGCOMM Computer Communication Review*, 2011.

[9] M. Chowdhury, Y. Zhong, and I. Stoica. Efficient coflow scheduling with varys. In *ACM SIGCOMM*, 2014.

[10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *USENIX NSDI*, 2010.

[11] F. R. Dogar, T. Karagiannis, H. Ballani, and A. Rowstron. Decentralized task-aware scheduling for data center networks. *ACM SIGCOMM Computer Communication Review*, 2014.

[12] U. Feige and C. Scheideler. Improved bounds for acyclic job shop scheduling. *Springer Combinatorica*, pages 361–399, 2002.

[13] F. Giroire, N. Huin, A. Tomassilli, and S. Pérennes. When network matters: Data center scheduling with network tasks. In *IEEE INFOCOM*, 2019.

[14] I. Gog, M. Schwarzkopf, A. Gleave, R. N. Watson, and S. Hand. Firmament: Fast, centralized cluster scheduling at scale. In *USENIX OSDI*, 2016.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 2014.

[16] R. Grandl, M. Chowdhury, A. Akella, and G. Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *USENIX OSDI*, 2016.

[17] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni. Graphene: Packing and dependency-aware scheduling for data-parallel clusters. In *USENIX OSDI*, 2016.

[18] 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 *USENIX NSDI*, 2019.

[19] Z. Hu, D. Li, Y. Zhang, D. Guo, and Z. Li. Branch scheduling: Dag-aware scheduling for speeding up data-parallel jobs. In *IEEE/ACM IWQoS*, 2019.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *ACM SIGOPS/EuroSys*, 2007.

[21] V. Jalaparti et al. Network-aware scheduling for data-parallel jobs: Plan when you can. *ACM SIGCOMM*, 2015.

[22] Y. Jiang, Y. Zhu, C. Lan, B. Yi, Y. Cui, and C. Guo. A unified architecture for accelerating distributed dnn training in heterogeneous gpu/cpu clusters. In *USENIX OSDI*, 2020.

[23] 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 *USENIX OSDI*, 2014.

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

[25] P. G. López, A. Arjona, J. Sampé, A. Slominski, and L. Villard. Triggerflow: trigger-based orchestration of serverless workflows. In *ACM DEBS*, 2020.

[26] 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 *ACM SoCC*, 2018.

[27] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh. Learning scheduling algorithms for data processing clusters. In *ACM SIGCOMM*. 2019.

[28] M. Mastrolilli and O. Svensson. (acyclic) job shops are hard to approximate. In *IEEE FOCS*, 2008.

[29] D. Narayanan, K. Santhanam, F. Kazhamiaka, A. Phanishayee, and M. Zaharia. Heterogeneity-aware cluster scheduling policies for deep learning workloads. In *USENIX OSDI*, 2020.

[30] J. H. Park, G. Yun, M. Y. Chang, N. T. Nguyen, S. Lee, J. Choi, S. H. Noh, and Y.-r. Choi. Hetpipe: Enabling large dnn training on (whimpy) heterogeneous gpu clusters through integration of pipelined model parallelism and data parallelism. In *USENIX ATC*, 2020.

[31] Y. Peng, Y. Bao, Y. Chen, C. Wu, and C. Guo. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *ACM EuroSys*, 2018.

[32] 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 *ACM SOSP*, 2019.

[33] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino. Apache tez: A unifying framework for modeling and building data processing applications. In *ACM SIGMOD*, 2015.

[34] A. Sergeev and M. Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint*, 2018.

[35] V. Shankar, K. Krauth, Q. Pu, E. Jonas, S. Venkataraman, I. Stoica, B. Recht, and J. Ragan-Kelley. Numpywren: Serverless linear algebra. *arXiv preprint*, 2018.

[36] O. Sonmez, N. Yigitbasi, A. Iosup, and D. Epema. Trace-based evaluation of job runtime and queue wait time predictions in grids. In *ACM HPDC*, 2009.

[37] X. Tang, H. Wang, X. Ma, N. El-Sayed, J. Zhai, W. Chen, and A. Aboulnaga. Spread-n-share: improving application performance and cluster throughput with resource-aware job placement. In *IEEE SC*, 2019.

[38] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *ACM SoCC*, 2013.

[39] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at google with borg. In *ACM EuroSys*, 2015.

[40] M. Wasi-ur Rahman, N. S. Islam, X. Lu, and D. K. Panda. A comprehensive study of mapreduce over lustre for intermediate data placement and shuffle strategies on hpc clusters. *IEEE TPDS*, 2016.

[41] W. Wu, A. Bouteiller, G. Bosilca, M. Faverge, and J. Dongarra. Hierarchical dag scheduling for hybrid distributed systems. In *IEEE IPDPS*, 2015.

[42] M. R. Wyatt, S. Herbein, T. Gamblin, A. Moody, D. H. Ahn, and M. Taufer. Prionn: Predicting runtime and io using neural networks. In *ACM ICPP*, pages 1–12, 2018.

[43] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, et al. Spark: Cluster computing with working sets. *USENIX HotCloud*, 2010.

[44] H. Zhang, Y. Tang, A. Khandelwal, J. Chen, and I. Stoica. Caerus: Nimble task scheduling for serverless analytics. In *USENIX NSDI*, 2021.

[45] Q. Zhang, M. F. Zhani, Y. Yang, R. Boutaba, and B. Wong. Prism: Fine-grained resource-aware scheduling for mapreduce. *IEEE TCC*, 2015.

[46] Z. Zhang, C. Chang, H. Lin, Y. Wang, R. Arora, and X. Jin. Is network the bottleneck of distributed training? In *ACM SIGCOMM Workshop on Network Meets AI & ML*, 2020.

[47] C. Zimmer, S. Gupta, S. Atchley, S. S. Vazhkudai, and C. Albing. A multi-faceted approach to job placement for improved performance on extreme-scale systems. In *IEEE SC*, 2016.