Reducing Makespans of DAG Scheduling through Interleaving Overlapping Resource Utilization

Yubin Duan*, Ning Wang†, and Jie Wu*
*Department of Computer and Information Sciences, Temple University, Philadelphia, PA, USA
†Department of Computer Science, Rowan University, Glassboro, NJ, USA
Email: yubin.duan@temple.edu, wangn@rowan.edu, jiewu@temple.edu

***Abstract*—As data center clusters need to process quintillion bytes of data per day, it becomes a critical problem that efficiently scheduling jobs to improve resource utilization. However, the data analysis job usually contains multiple stages with dependent relationships, which brings challenges for scheduling. Those stages are modeled as Directed Acyclic Graphs (DAGs) and the general DAG scheduling problem is NP-hard. In this paper, we notice that in some parallel computing frameworks such as Spark, the execution of each stage could be divided into multiple phases that use different resources. We observe that interleaving different resources in a pipelined manner could improve resource utilization. Based on this observation, we propose to minimize the job makespan by exploiting resource pipeline. We first theoretically analyze the scheduling for perfectly parallel stages. In this case, our scheduling problem is equivalent to a DAG shop problem which is NP-hard. A contention-free scheduler is proposed and its approximation properties are analyzed. Stages of real-world jobs are usually not perfectly parallel. For general jobs, a reinforcement learning (RL) based scheduler is proposed to adaptively adjust the resource contention. We evaluate our contention-free and RL-based schedulers on a Spark cluster deployed on the Amazon EC2. Experiments on real-world and synthetic datasets show our RL-based scheduler can improve the CPU and network utilization by 33.0% and 29.7%, respectively.**

***Index Terms*—data center clusters, DAG scheduling, makespan minimization, pipelines**

I. Introduction

With the rapid growth of data volume in the big data industry, reducing the makespan of data analysis becomes more and more critical. Over quintillion bytes of data are generated every day from Internet of Things devices. However, obtaining the optimal schedule of jobs in polynomial time is challenging. Big data analysis jobs usually consist of multiple stages with dependencies. The dependency in a job are usually modeled by a directed acyclic graph (DAG) as shown in Fig. 1 and the general DAG scheduling problem is known as NP-hard. In some parallel computing frameworks such as Spark, the execution of each stage could be divided into multiple phases that use different resources as shown in Fig. 1. Those stages could be processed in a pipeline. Exploiting the pipeline could improve the resource utilization but also brings challenges for DAG job scheduling.

We have observed that several stages competing for a resource would enlarge the makespan of a job. Specifically, if we launch multiple stages at the same time as the default scheduler, the intense contention on one resource could reduce the utilization of other resources. We use stages 1 and 2 of the DAG shown in Fig. 1 as an example. Let q_i and q_i' denote the data fetching and data processing phase of stage i, respectively. The time of result writing is negligible since the sizes of results are small and are written to the local disk. If the scheduler starts q_1 and q_2 simultaneously as shown in Fig. 2(a), both q_1 and q_2 take longer time to finish compared with executing them individually. The longer execution time of q_1 and q_2 further delays the starting of q_1' and q_2', which eventually leads to a larger time cost of finishing stages 1 and 2. Interleaving the resources in a pipelined manner could reduce the makespan.

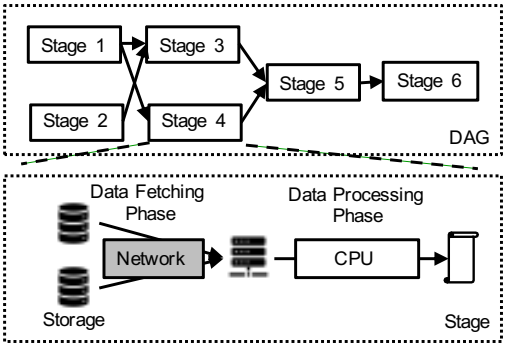
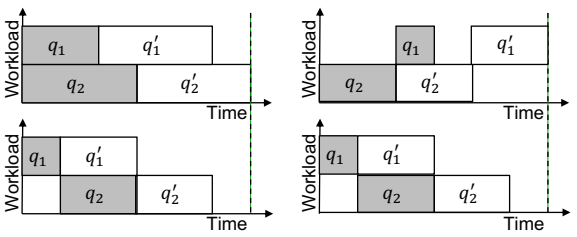


Fig. 1. Illustrations of DAGs, stages and phases.

Besides the resource contention, the execution order of stages also impacts the makespan of a job. It can be shown on the same example. As shown in Fig. 2(b), if the scheduler starts q_2 before q_1 and avoids the resource contention, the makespan of executing stages 1 and 2 is longer compared with that of starting q_1 first. The longer execution time needed by q_2 reduces the utilization of the computational resource.

The motivation example shows the benefits brought by the resource pipeline. However, existing researches, such as [1]–[6], pay little attention to this aspect. In this paper, we investigate the stage scheduling problem for a DAG-style job to minimize the job makespan. We focus on reducing resource contention in the stage execution. We theoretically analyze the scheduling for perfectly parallel stages whose speedups are proportional to the number of resources allocated to them. We show that the optimal schedule for those stages is contention-free, and convert the scheduling problem into a DAG shop

This research was supported in part by NSF grants CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, CNS1651947, and CNS 1564128.



(a) The resource contention would increase the makespan

(b) A good execution order would reduce the makespan

Fig. 2. The challenges in scheduling DAG jobs.

problem which is NP-hard. A contention-free scheduler is proposed and its approximation properties are analyzed.

We also notice that stages are usually not perfectly parallel in real-world workloads. The contention-free schedule is no longer suitable for general stages. Allocating all resources to a stage is a waste since the stage cannot make full use of so many resources. A reasonable level of resource contention is needed. We propose a reinforcement learning (RL) based scheduler to adaptively adjust the starting time of each stage and the percentage of resources allocated to it to control the contention level. The RL-based scheduler frequently takes the available resources and unprocessed stages as the input state and adaptively updates the schedule for the remaining stages.

We evaluate our schedulers on a Spark cluster deployed on the Amazon Elastic Compute Cloud. Our evaluation uses both a real-world dataset from Alibaba and a synthetic dataset.

The contributions of the paper are summarized as follows:

- We investigate the stage scheduling problem for jobs with DAG structures. Especially, we propose to minimize the job makespan by reducing resource contentions.
- We theoretically analyze the scheduling for perfectly parallel stages which converts our problem into a DAG shop problem. A contention-free scheduler is proposed. Its approximation properties are analyzed.
- We also consider the scheduling for general stages, which makes the problem more practical. We investigate a RL-based scheduler which can adaptively adjust the scheduling policy from experiences.
- Experiments on both synthetic and real-world datasets show our scheduler could efficiently improve the resource utilization and reduce the job makespan.

II. Models

A. Overview of the Spark Job and Stage

In the *Apache Spark* framework, a *job* usually consists of a set of *stages* with dependencies. The execution of the job is sliced into the processing of stages. Because of the data flow in each job, some stages cannot be processed until intermediate results are generated by some other stages. The inter-dependencies among stages in a job are usually represented by a *Directed Acyclic Graph* (DAG).

A stage is a physical unit of execution and contains a set of parallel tasks. The Spark scheduler could allocate multiple executors to a stage and process it in parallel. The procedure of a stage execution is illustrated in Fig. 3. The procedure can be divided into two phases: the *data fetching phase* and the *data processing phase*. In the data fetching phase, worker machines shuffle read the data partitions which are distributed among different nodes of the cluster. The data fetching phase is network I/O intensive. In the data processing phase, executors in worker machines run the task functions on the data partitions they fetched, and write the result on their local disks. The data processing phase is computation intense.

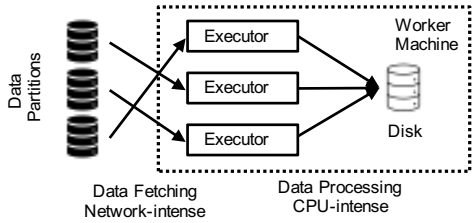


Fig. 3. Procedures of executing a stage on a Spark cluster.

The Spark scheduler controls when a stage starts and how many executors to use. We focus on designing a scheduler which could reduce the job makespan.

B. Notations

Before we formulate our stage scheduling problem, we first introduce the notations we used. Let $G = (S, E)$ denote the DAG. W.l.o.g., we assume there is only one job in our model. Since we consider the stage-level scheduling, there is no need to distinguish different jobs. Scheduling a batch of DAG-style jobs could be treated as scheduling a special DAG-style job which consists of multiple separate DAGs. The vertex set S of the graph G represents the set of stages of the job. Specifically, $S = \{s_1, s_2, \ldots, s_n\}$, where n is the number of stages in the job. The directed edge set E of the graph represents the dependency relations of stages in S. An edge from s_i to s_j is denoted by an ordered pair $(s_i, s_j) \in E$. It means that the stage s_j cannot start until the stage s_i is finished. We divide the execution of a stage into a data fetching phase and a data processing phase. Let q_i and q_i' denote the data fetching phase and data processing phase of a stage $s_i \in S$, respectively. q_i' cannot start until q_i is finished.

The time consumption of executing a stage is affected by the stage size and the amount of resources allocated to it. The stage size is quantified by the overall size of data partitions that are processed in the stage. We use d_i to denote the overall data size of stage s_i. Stages can be processed in multiple worker machines in parallel. If multiple stages are running, we assume both of the executor resource and the bandwidth resource are *equally allocated* to those stages. We use p_i to denote the *parallelism level* of each stage s_i, i.e., the number of executors assigned to the stage. The bandwidth allocated to the stage s_i is denoted as b_i. Let l_i and l_i' denote the length or duration of phases q_i and q_i', respectively. Then, l_i and l_i' can be formulated as a function of the data size d_i, the parallelism level p_i and the bandwidth b_i. Formally,

$l_i = f_i(d_i, b_i)$ and $l'_i = f'_i(d_i, p_i)$. The explicit expression of f and f' depends on different types of DAG stages. For *perfectly parallel stages*, f_i and f'_i are linear functions. Specifically, $f_i(d_i, b_i) \propto d_i/b_i$ and $f'_i(d_i, p_i) \propto d_i/p_i$. For general DAG stages, they are non-linear functions. We use t_i and t'_i to denote the start time and completion time of the stage s_i. We consider the non-preemptive scheduling. Hence, $t'_i = t_i + l_i + l'_i = t_i + f_i(d_i, b_i) + f'_i(d_i, p_i)$, i.e., the completion time of a stage is determined by the start time t_i, data partition size d_i, the parallelism level p_i, and the bandwidth b_i.

For each stage, its parallelism level and bandwidth are correlated with the starting time of itself and all other stages. Instead of simultaneously adjusting all three factors, the scheduler can control the values of p_i and b_i by setting the start time for all stages, since we assume the available resources are *equally* allocated to stages running in parallel. This assumption is practical and reduces the solution space for our problem. Formally, let $O(t)$ denote the set of stages that are running in parallel at time t. Then, $O(t)$ can be calculated by counting stages whose processing intervals $[t_i, t'_i]$ contain t. Formally, $O(t) = \{s_i \in S | t \in [t_i, t'_i]\}$. We use P and B to denote the total number of executors and the overall bandwidth in the cluster, respectively. Then, the computation and network resources allocated to each stage $s_i \in S$ are $p_i = P/|O(t_i)|$ and $b_i = B/|O(t_i)|$, where $|O(t_i)|$ is the set cardinality.

We also notice that it is not necessary to allocate too many executors to a stage. For general DAG stages, the length of the data processing phases $l'_i = f'_i(d_i, p_i)$ can hardly be further reduced when its parallelism level p_i exceeds a certain threshold. Details are explained in Section IV.

The makespan of executing a stage is denoted as τ. Formally, $\tau = \max_{s_i \in S}(t'_i) - \min_{s_i \in S}(t_i)$. Its value is determined by the scheduling policy. Specifically, Let $\mathcal{P}$ denote the scheduling policy. It consists of a vector of start times and a vector of parallelism levels for all stages. The makespan τ can be reduced by wisely adjust the policy $\mathcal{P}$.

C. Problem Formulation

In this paper, we aim to design a scheduler which could interleave the usage of different types of resources such that the makespan of executing a job is minimized. Specifically, the resource contention can be reduced by wisely adjusting the start time t_i and the parallelism level p_i for each stage $s_i \in S$. We formulate our scheduling problem as follows:

$$\min \quad \tau, \tag{1}$$

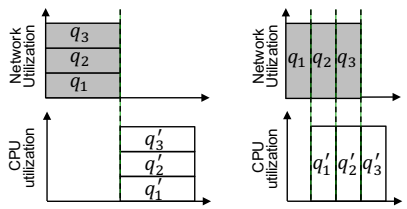
$$s.t. \quad t'_i \le t_j, \forall (s_i, s_j) \in E, \tag{2}$$

$$\sum\nolimits_{s_i \in O(t)} p_i \le P, \forall t > 0, \tag{3}$$

$$\sum\nolimits_{s_i \in O(t)} b_i \le B, \forall t > 0, \tag{4}$$

$$t_i \ge 0, \forall s_i \in S. \tag{5}$$

Eq. (1) shows our objective of minimizing the makespan of the job execution. Eq. (2) is the precedence constraint. If there is an edge $(s_i, s_j) \in E$, then the start time of the stage s_j cannot be earlier than the completion time of the stage s_i.



(a) Splitting resources (b) Contention-free

Fig. 4. A motivation of scheduling ideal stages in a pipelined manner.

Eq. (3) is the computation resource constraint, where $O(t) = \{s_i \in S | t \in [t_i, t'_i]\}$ is the set of stages processing in parallel at time t. Eq. (4) is the bandwidth constraint. Eq. (5) is the schedule constraint. Each stage $s_i \in S$ should be scheduled and processed by some executors.

D. Problem Hardness

Finding the optimal solution for our stage scheduling problem is hard. The DAG structure of the job as well as the complex relation between stage lengths and the contention level bring challenges to our optimization problem. We find that our problem is NP-hard even in an ideal case where the DAG consists of all perfectly parallel stages whose time consumption functions $f_i(d_i, b_i)$ and $f'_i(d_i, p_i)$ are linear and have closed-form expressions . The proof is shown in Section III. For more general cases where f_i and f'_i are non-linear w.r.t. b_i and p_i, the problem becomes even harder.

III. Scheduling for Perfectly Parallel Stages

A. Contention of Perfectly Parallel Stages

We first investigate the scheduling for perfectly parallel stages. Those stages have some useful properties which could help to reduce the complexity of the scheduling problem. Specifically, there is no need to set a parallelism limitation for a perfectly parallel stage. The formulations of $l_i = f_i(d_i, b_i) \propto d_i/b_i$ and $l'_i = f'_i(d_i, p_i) \propto d_i/p_i$ show that the speedup of those phases are proportional to the units of resources allocated to them. Therefore, we could simply assign all computational resources to a stage . Then, the scheduling problem becomes to determine the start time for all stages.

In addition, simultaneously running multiple perfectly parallel stages bring no benefits. It might even enlarge the job makespan. Specifically, the execution time of perfectly parallel stages merely depends on resource utilization. Simultaneously running multiple stages cannot further improve the utilization since running one stage already can make full use of all resources. Splitting resources to multiple stages may enlarge the completion time of some phases, and it delays the start of the following phases. Fig. 4 shows a straightforward example. If we split the network resource to simultaneously execute q_1, q_2, and q_3 as shown in Fig. 4(a), the start of phases q'_1, q'_2, and q'_3 would be delayed. It reduces the utilization of computation resources. If we assign all resources to one stage at a time as shown in Fig. 4(b), the makespan of executing stages s_1, s_2 ,and s_3 can be reduced. Therefore, we

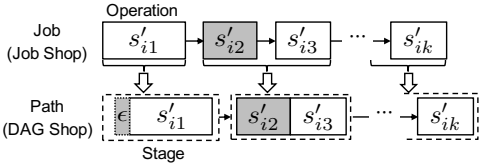


Fig. 5. Converting operations in the job shop to stages in the DAG shop.

can schedule perfectly parallel stages in a pipelined manner. It reduces the searching space of finding the optimal t_i. We only need to determine an execution sequence for those stages. Based on the sequence, the scheduler starts a stage right after its previous stage is finished.

Although the useful properties of perfectly parallel stages reduce the solution space, our scheduling problem is still NP-hard. The NP-haedness is shown in Theorem 1.

***Theorem** 1:* Our scheduling problem for perfectly parallel stages is NP-hard .

Proof. Any instance $\mathcal{J}'$ of the *job shop problem* [7] with two machines can be converted into an instance $\mathcal{J}$ of our stage scheduling problem with all perfectly parallel stages in polynomial time. The solution of $\mathcal{J}$ also could be transformed into the solution of $\mathcal{J}'$ in polynomial time. Specifically, an instance $\mathcal{J}'$ can be stated as follows: We are given n jobs. Job i has a sequence of k_i operations which must be processed in this order. Operations can be divided into two types. Each type of operation must be processed on a specific machine, and each machine can process one operation at one time. The objective is to minimize the makespan of n jobs. A job in job shop problem is shown in Fig. 5. s'_{ij} denotes the j-th operation on job i. Note that if two adjacent operations belong to the same type, those operations could be merged. Therefore, we can assume adjacent operations are different in our proof. As shown in Fig. 5, we could convert the job into a path in DAG by inserting dummy operations and treating operations in jobs as phases in stages. For example, after inserting an ϵ-length operation before s'_{i1}, we can treat those two operations as two phases in a stage. An job shop instance $\mathcal{J}'$ can contain multiple jobs. Each job could be converted into a DAG path in polynomial time. Then, a common ancestor with two ϵ-length operations is added before paths. If we treat $\epsilon = 0$, the instance $\mathcal{J}'$ is converted into an instance $\mathcal{J}$ of our stage scheduling problem with perfectly parallel stages.

Stages in $\mathcal{J}$ are perfectly parallel, which makes an optimal solution of $\mathcal{J}$ optimal for $\mathcal{J}'$. The solution to $\mathcal{J}$ contains start time t_i and parallelism level p_i for each stage. For perfectly parallel stages, $p_i = P$ and it is determined. The start time t_i could be converted into the processing sequence by sorting. There may exist resource contentions in the optimal schedule of $\mathcal{J}$. But we can always find an equivalent schedule that has the same makespan and no resource contention. Specifically, for any two stages s_i and s_j running in parallel and competing for a resource, we could always delay the stage with the larger start time without affecting the makespan, since their overall workload is certain. W.l.o.g., we assume $t_j > t_i$ and the processing of q_i and q_j are overlapped. Then, we could delay the start of q_j such it starts after the completion of q_i. Because the overlapped sizes of q_i and q_j as well as the amount of resource B are fixed, the time needed to finish those phases would not change, no matter they are processes simultaneously or separately. Hence, the value of t'_j remains and the execution of the following stages would not be affected.

Above all, the instance $\mathcal{J}$ and $\mathcal{J}'$ are equivalent. Considering the job shop problem with two machines is NP-hard [7], our problem is also NP-hard. ■

Algorithm 1 Contention-free Scheduling Algorithm

Input: The DAG $G=(S,E)$, available resources (B,P)
Output: The scheduling for DAG stages in S

1: Evaluate phase lengths $l_i=f_i(d_i,B), l'_i=f'_i(d_i,P), \forall s_i \in S$
2: Initialize the schedule list $L \leftarrow \emptyset$
3: **while** S is not empty **do**
4: Ready-to-go stage set $S' \leftarrow \{s_i \in S | (s_j,s_i) \notin E, \forall s_j \in S\}$
5: Shuffle-heavy stage set $S_1 \leftarrow \{s_i \in S' | l_i > l'_i\}$. Computation-heavy stage set $S_2 \leftarrow S' \setminus S_1$
6: $L_2 \leftarrow$ Sort $s_i \in S_2$ for ascending order of l_i. $L_1 \leftarrow$ Sort $s_i \in S_1$ for descending order of l'_i. $L \leftarrow L||L_2||L_1$
7: Update $S \leftarrow S \setminus S'$. Remove corresponding edges in E
8: **return** L as the schedule list

B. Contention-free Scheduling

We propose a *contention-free* scheduling algorithm for perfectly parallel stages. The contention-free scheduling means both data fetching and processing phases of each stage $s_i \in S$ can acquire all cluster resources, i.e., $b_i = B$ and $p_i = P$. The motivation of using a contention-free scheduler is that splitting resources to run multiple stages concurrently cannot reduce the makespan, but may even increase it. Besides, as shown in the proof of Theorem 1, any optimal schedule could be converted into an equivalent contention-free schedule. In our contention-free scheduler, stages are executed in a pipelined manner to interleave the resource usage.

To generate the contention-free scheduling, we need to determine the processing sequence of stages. The DAG structure (precedence constraints of stages) and two different types of phases make it challenging to find the optimal sequence. The DAG structure gives partial order relations among all stages. We need to extract a feasible total order relation when building the sequence. Besides, the lengths of phases vary with stages. Some stages are shuffle-heavy and have longer data fetching phases than the processing phases, while some other stages are computation-heavy. Scheduling those stages without precedence constraints is not trivial. Dealing with those factors at the same time is NP-hard and we treat them separately.

We borrow ideas from the topological sort and Johnson's rule [8] to design our scheduling algorithm. The topological sort can find feasible execution sequences of stages in the DAGs. However, the number of feasible sequences is exponential. Calculating makespans of all feasible sequences and comparing them cannot be done in linear time. Johnson's

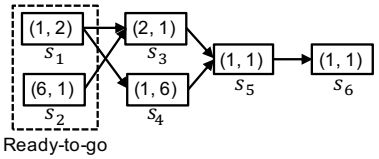


Fig. 6. The illustration of the ready-to-go stages.

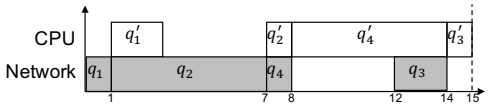
rule is a method of scheduling flow shop problems. Without the precedence constraints, it can optimally solve our stage scheduling problem.

Intuitively, our contention-free scheduling algorithm iteratively uses Johnson's rule on a set of ready-to-go stages until all stages are scheduled. A ready-to-go stage is a stage whose predecessors are scheduled. Use the DAG in Fig. 6 as an example, the initial set ready-to-go stages contains s_1 and s_2. After stages s_1 and s_2 are finished scheduled, s_3 and s_4 becomes ready-to-go. In each iteration, our algorithm schedules all ready-to-go stages in the set, removes those stages form the DAG, and determines the next set of ready-to-go stages. It stops when all stages are processed and removed from the DAG.

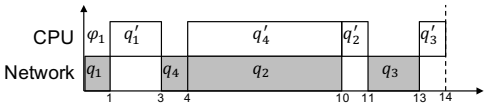
The detailed procedures of our algorithm are illustrated in Alg. 1. Lines 1-2 calculate the phase lengths for all stages and initialize the schedule list. In lines 3-7, we iteratively schedule a set of ready-to-go stages. Line 4 finds the ready-to-go stages that have no income edges in G. Lines 5-6 apply Johnson's rule. The stages are divided into a shuffle-heavy group S_1 and a computation-heavy group S_2. Stages in the computation-heavy group S_2 have shorter data fetching phases. For $s_i \in S_2$, we prefer to process the stage with the shortest data fetching phase l_i first. For shuffle-heavy stages, we process the stage with the shortest data processing phase l'_i last. Then, we concatenate the sorted stages in S_2 and S_1 to the list, and the computation-heavy stages in S_2 are concatenate before S_1. The concatenation is represented by $||$. Line 7 updates the graph for the next iteration. Line 8 returns the result.

We use the DAG shown in Fig. 6 as a go-through example. The tuple (l_i, l'_i) associated with each stage indicates the lengths of its data fetching and data processing phase. In the first iteration, s_1 and s_2 are ready-to-go stages. s_1 is computation-heavy and is scheduled before s_2 that is shuffle-heavy. Then, they are removed from the DAG and s_3, s_4 becomes ready-to-go. s_3 has a longer data processing phase and is scheduled before s_4. After s_3 and s_4 are scheduled, s_5 has no predecessors and is concatenated to the schedule list. Eventually, s_6 is appended to the schedule list. Based on the sequence of stages in the list and the principle of contention-free, the start time of each stage can be easily derived. Our schedule of the first four stages is illustrated in Fig. 7(a). Stages 5 and 6 are not shown since their schedule is fixed according to their dependent relationships.

Although ready-to-go stages are optimally scheduled in each iteration, the final schedule for all stages might be suboptimal. This is because we manually set precedence restrictions for stages among different ready-to-go groups. For example, we



(a) Our contention-free schedule (s_1,s_2,s_3,s_4)



(b) The optimal schedule (s_1,s_4,s_2,s_3)

Fig. 7. An example of our scheduling algorithm.

schedule s_1 and s_2 before s_3 and s_4 the of the DAG in Fig. 6. It introduces a precedence restrictions $s_2 \rightarrow s_4$, which is not necessary. Adding those restrictions could let the scheduler miss the optimal solution. The optimal schedule is shown in Fig. 7(b), where s_4 is processed before s_2.

Our contention-free scheduler is 2-approximate for perfectly parallel stage scheduling. It is $3/2$-approximate if data fetching and data processing phases of all stages have a unit length. The 2-approximation ratio is trivial. The insight is that our scheduler would not leave both resources idle. Formally, let τ^* denote the optimal makespan. Then, we have $\tau^* \geq \max\{\sum_{s_i} l_i, \sum_{s_i} l'_i\}$ since even if the optimal scheduler could perfectly pipeline all phases, it cannot compress the essential computation or communication time consumption. Our contention-free scheduler would not leave both resources idle. Therefore, our makespan $\tau \leq (\sum_{s_i} l_i + \sum_{s_i} l'_i) \leq 2\max\{\sum_{s_i} l_i, \sum_{s_i} l'_i\} \leq 2\tau^*$. Theorem 2 shows the $3/2$-approximation ratio for the unit-length case.

***Theorem** 2:* Our contention-free scheduler is $3/2$-approximate if $l_i = l'_i = c, \forall s_i \in S$, where c is a constant.

Proof. The key property used in the proof is that the total resource idle time of our schedule would not exceed the optimal makespan τ^*. Let $\Phi = \{\varphi_1, \ldots, \varphi_k, \ldots\}$ denote the set of idle slots in the scheduling. For example, in Fig. 7(b), φ_1 represents the CPU idle time before processing q'_1. For each $\varphi_k \in \Phi$, we can find a corresponding stage phase ν_k which is in execution during φ_k, since two types of resource would not be idle simultaneously. In Fig. 7(b), $\nu_1 = q_1$. We will show that for $\varphi_k \in \Phi$ $(k \neq 1, k \neq |\Phi|)$, their corresponding ν_k cannot be pipelined when $l_i = l'_i = c, \forall s_i \in S$. For any two adjacent φ_k and φ_{k+1}, we have $\nu_k \prec \nu_{k+1}$ meaning there is a partial order relation between ν_k and ν_{k+1}. If it is not the case, ν_k and ν_{k+1} should run simultaneously by shifting ν_{k+1} ahead to occupy φ_k. Therefore, there is a chain $\nu_2 \prec \nu_3 \prec \cdots \prec \nu_{|\Phi|-1}$. Similar to the concept of the critical path, the makespan of this chain cannot be reduced, even in the optimal schedule. It means that the idles of φ_k for $2 \leq k \leq |\Phi| - 1$ cannot be avoided even in the optimal schedule. The head φ_1 and tail $\varphi_{|\Phi|}$ cannot be avoided either. Therefore, in the optimal schedule, its total idle time is greater or equal to $\sum_{k=1}^{|\Phi|} \varphi_k$. Besides, the optimal makespan τ^* must be greater or equal to its total idle time. Therefore, $\tau^* \geq \sum_{k=1}^{|\Phi|} \varphi_k$.

We notice that $\tau = \frac{1}{2}(\sum_{s_i} l_i + \sum_{s_i} l'_i + \sum_{k=1}^{|\Phi|} \varphi_k)$. We have

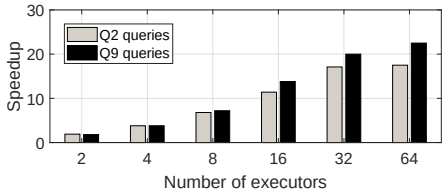


Fig. 8. Speedups of two different jobs on the Spark cluster.

shown that $(\sum_{s_i} l_i + \sum_{s_i} l'_i) \leq 2\max\{\sum_{s_i} l_i, \sum_{s_i} l'_i\} \leq 2\tau^*$ and $\sum_{k=1}^{|\Phi|} \varphi_k \leq \tau^*$. Hence, $\tau \leq \frac{1}{2}(2\tau^* + \tau^*) = \frac{3}{2}\tau^*$. The $3/2$-approximation ratio holds. ■

IV. Scheduling for General Stages

When scheduling general stages, contention-free scheduling is no longer optimal. For general DAG stages, their speedups are no longer proportional to the units of resources allocated to them, especially for the data processing phase. When the number of executors allocated to a stage exceeds a threshold, adding more executors to the stage would barely reduce its execution time any further. For those stages, allocating all executors to a ready-to-go stage as the contention-free scheduler is a waste. The scheduler should set an upper bound on the parallelism level of each stage. If the limitations are properly set, simultaneously executing multiple stages and controlling their competition within a reasonable level could improve resource utilization and reduce the makespan. In this section, we first show the non-linear speedup of general DAG stages. Then, we introduce a reinforcement learning (RL) based scheduler for general stage scheduling. Finally, we discuss the training details of the RL agent.

A. Speedup of General Jobs

We first test the speedup of general DAG stages on the Spark server. Fig. 8 shows the speedup of two different jobs from the TPC-H dataset[1]. From the figure, we can observe the non-linear speedup and the parallelism level threshold. For example, for the Q2 job, allocating more than 32 executors would barely further improve the speedup or even might reduce it. Therefore, it is a waste that allocating more than 32 executors and the scheduler should set a parallelism limitation m_i. When the available executors are more than the limitation, the scheduler should allocate the extra executors to other stages. The contention-free scheduler is no longer optimal.

It is difficult to theoretically model the speedup in practice. Although the Amdahl's law [9] shows a speedup model, the percentages of sequential parts in each DAG stage is not clear. According to the Amdahl's law, the execution time of sequential parts is fixed, and the speedup of the parallel part is proportional to the number of executors. However, determining the percentage of sequential parts is hard to implement in practice. Therefore, it is not reasonable to determine a fixed parallelism level for all DAG stages. In contrast, we adapt a RL based scheduler to adaptively adjust the parallelism level for different DAG stages and maintain the resource contention at a reasonable level.

[1]Available online: http://www.tpc.org/tpch/

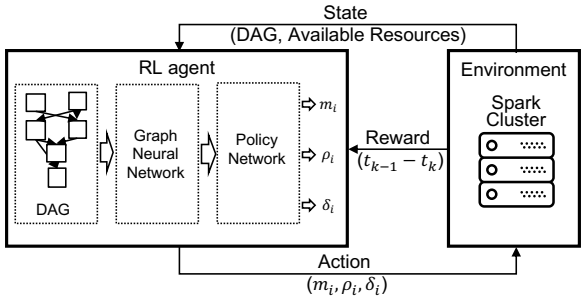


Fig. 9. The reinforcement learning framework for general stage scheduling.

B. A RL-based Scheduler for General Stage Scheduling

We adapt the RL framework in [10] to generate schedules for general stages. Different from the RL agent in [10], we consider to control the resource contention by adjusting the start time t_i of each stage s_i. The RL framework is shown in Fig. 9. It consists of a RL agent and the environment. The agent observes the state from the environment and generate the schedule as an action. The environment is the Spark engine running on the data center cluster.

We first show our design of the action space. The scheduler needs to determine the parallelism level limitation m_i and the starting time t_i for each stage $s_i \in S$. The possible values of m_i are discrete and bounded by the total number of executors P. Formally, $m_i \in \{1, 2, \ldots, P\}$. We can use a neural network with softmax layers to calculate the probability of choosing each potential value in $\{1, 2, \ldots, P\}$. Determining the value of t_i is more challenging since its value is continuous and there is no fixed upper bound of its possible value. Searching the optimal value for $t_i \in \mathbb{R}^+$ without closed-form formulations is intractable. Therefore, we discretize t_i by exploiting ideas of list scheduling and delay scheduling [11].

In the list scheduling, stages are ordered by assigning with priorities. During execution, ready-to-go stages are repeatedly selected based on their priorities when there are available resources. Based on this idea, we let the RL agent set discrete priorities to stages instead of directly learning their start time. We use ρ_i to denote the priority of s_i. However, simply setting priorities is not sufficient to control the resource contention. When the available executors are sufficient to execute multiple stages, those stages should not start simultaneously. Otherwise, similar to the motivation example shown in Fig. 2(a), it would cause the network resource contention and enlarge the finish time of those stages.

Unlike computational resources, it is not convenient to set bandwidth limitations. Inspired by the delay scheduling [11], we interleave the usage of network resources by delaying the start of some stages. The RL agent needs to learn the length of delay time δ_i for each stage s_i. After a stage s_i is selected by the scheduler based on its priority, a timer with length δ_i is associated with the stage. The stage s_i would not start until its timer is out. To reduce the action space, we set an

upper bound for each δ_i and discretize its value. The delay length should not exceed the longest stage length $l_{\max}$ that the scheduler has seen so far. The $l_{\max}$ is sliced into Δ pieces, where Δ is a hyperparameter. Formally, $\delta_i \in \{0, l_{\max}/\Delta, 2 \cdot l_{\max}/\Delta, \ldots, l_{\max}\}$. Then, to determine the delay length, the RL agent only needs to choose a value from $\{0, 1, 2, \ldots, \Delta\}$ by using neural networks with softmax layers.

To adaptively adjust the schedule, we frequently invoke the RL agent when there are available resources and unprocessed stages. Specifically, we call the RL agent at following trigger events: a stage starts and there are unused resources; a stage completes and releases its resource. At each trigger event, the RL agent would update the (m_i, ρ_i, δ_i) for each unprocessed stage s_i. Then, from all ready-to-go stages, one is selected based on their priorities. After its timer expires, the selected stage is allocated with executors whose number would not exceed its parallelism limitation. The remaining challenge is how to encode the DAG and the dependent relationships indicated by the DAG.

To capture dependent relationships, the Graph Neural Network (GNN) [12] is used to encode the DAG. GNN encodes the dependencies by aggregating DAG information from children to parent nodes along the DAG edges. By aggregating stage information along paths in DAG, GNN could convert the DAG into a fixed-length feature vector. Along with the features describing the system workload, i.e., the resource utilization information, the state used by our RL agent is formed.

Given a state, the goal of the RL agent is to generate an action that could maximize the expected future reward (or minimize the expected future penalty). We use r_k to denote the reward of its k-th action. r_k is quantified by the negation of the time interval length between the $k-1$-th and the k-th action. Let t_k denote the wall-clock time at the k-th action. Then, $r_k = -(t_k - t_{k-1})$. The negation is used to show that the term $(t_k - t_{k-1})$ is actually a penalty. With this formulation, the expected future penalty is $\mathbb{E}[\sum_k (t_k - t_{k-1})] = \mathbb{E}[t_T - t_{k-1}]$, where t_T it the time of the last action. The $\mathbb{E}[t_T - t_{k-1}]$ shows the expected time consumption for executing the remaining stages. Therefore, minimizing this penalty function could help to reduce the makespan.

V. Related Work

Based on different schedule granularity, existing DAG schedulers could be divided into three major groups: job-level schedulers, stage-level schedulers, and task-level schedulers. The job-level schedulers arrange the sequence of job execution and the typical objective is to reduce the job response time. Besides the classical FIFO or Fair scheduling, Hu *et. al.* [13] propose to use multiple level priority queues to schedule the jobs without knowing their sizes in advance. The stage-level schedulers consider the execution of stages, including the parallelism level, resource allocation, and dependence relations of stages. Mao *et. al.* [10] follow a reinforcement learning approach to determine the parallelism level and priority of each stage. For resource allocation, Grandl *et. al.* [14] propose to greedily match the stage resource demands with available resources. They further define the concept of troublesome stages in [5]. Troublesome stages would be considered first on the resource plane. Our paper focus on stage-level scheduling. Different from existing schedulers, we notice that the interleave usage of resources could help reduce job makespan and improve resource utilization. [15] and [16] also propose to interleave resources. To improve resource utilization, [17] and [18] discuss solutions for private datacenters. [19] further considers the public dataset. Different from them, we develop a scheduler based on reinforcement learning to adaptively interleave resources for general DAG stages. Each stage in a DAG consists of a set of parallel tasks. A task scheduler such as Monotasks [20] considers fine-grained parallelization of tasks. However, it needs to modify the Spark API while our scheduler could be easily implemented on Spark.

The core challenge of designing a stage-level scheduler is brought by the precedence constraints in DAGs. Existing theoretical analyses [21], [22] usually focus on simple cases. The state-of-the-art theoretical result is given in [21]. However, we cannot directly apply those theoretical results to our problem since we also consider the precedence relation between two phases in each stage. Scheduling those phases is also no trivial, and it can be viewed as a shop scheduling problem [7]. It has been proven that the job shop problem is hard to approximate [23]. Shmoys *et. al.* [24], [25] show several RNC-approximation algorithms for shop scheduling. Although their algorithms are polynomial-time in theory, they are inefficient. Zheng *et. al.* [26] consider the shop scheduling problem in the MapReduce framework. There is no DAG structure in their problem formulation. We jointly consider the DAG scheduling and the shop scheduling problems.

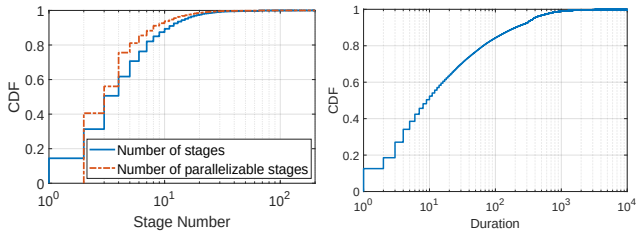
VI. Experiment

A. Dataset

In the experiment, we use the Alibaba trace data v2018[2] to evaluate our contention-free scheduler and the RL-based scheduler. The Alibaba dataset contains job traces sampled from their production cluster. Most of the jobs in the dataset have DAG structures. Besides, we also construct a synthetic dataset. We choose the CosineSimilarity job which is available in Spark MLlib and has 5 stages.

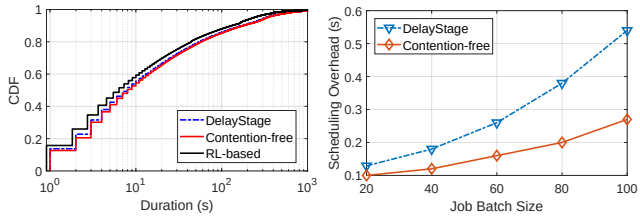
Before the experiment, we first illustrate the percentage of parallel stages in the Alibaba dataset. Fig. 10 shows the statistics of the Alibaba dataset. Fig. 10(a) shows the distribution of the number of stages in a job. In total, the dataset contains 2,775,025 jobs. From the figure, we can find that most of those jobs have more multiple stages. More than 80% of those jobs have more than one stages. Besides, we use topological sort the analysis the number of parallel stages in each job, and find that more than 68% of jobs have parallel stages. It shows the importance of efficiently scheduling parallel stages. Fig. 10(b) shows the distribution of stage duration that executing in their cluster. It shows the distribution of stage sizes to some extent.

[2]Avaiable online: https://github.com/alibaba/clusterdata/blob/master/cluster-trace-v2018/trace_2018.md



(a) The stage number distribution (b) The stage duration distribution

Fig. 10. Statistics of the Alibaba dataset.



(a) The stage duration distribution (b) Comparison on the overhead

Fig. 11. Performance evaluation on the Alibaba dataset.

B. Experiment Setting

We evaluate our contention-free scheduler and the RL-based scheduler on a real Spark cluster and in simulations. The Spark cluster is set up on the Amazon Elastic Compute Cloud (EC2). We use 10 `m4.xlarge` instances. Each instance has 4 Intel Xeon E5-2676 vCPU cores, 32GB RAM, 750MB maximum bandwidth. When setting up the Spark clsuter, the default parameter configuration is kept for simplicity.

Besides using the EC2 cluster, we use a local PC to train our RL-agent. The local Pc has an Intel i7-8700 CPU, a 32GB RAM, and a single Nvidia GTX 1080 GPU. We use the REINFORCE policy gradient algorithm [27] to train the RL agent, and we subtract the baseline performance from the reward function in each iteration of the parameter updating. Specifically, the baseline is used to reduce the variance of the policy gradient. Details of the explanation on subtracting baselines can be found at [28]. On our local PC, each training iteration takes about 5 seconds on average. Considering the initial policy of the RL agent is randomly generated, its performance is not good enough to handle a heavy workload. Therefore, we first use small job batches to train the RL agent and then gradually enlarge the job batch size. A well-trained RL agent is deployed on the EC2 cluster.

C. Experiment Result

We first compare the stage execution time obtained by different schedulers. We compare our contention-free strategy and RL based scheduler with the DelayStage scheduler in [16]. We execute the same job batch with different schedulers on the Spark cluster and record the duration of each stage. The distribution of the stage duration is shown in Fig. 11(a). Lines on the left have better performance. From the figure, we can find that the stage duration distributions achieved by the contention-free scheduler and the DelayStage scheduler are similar. The DelayStage scheduler slightly outperforms then contention-free scheduler. It is because that perfectly parallel assumption of the contention-free scheduler is strong and can hardly be satisfied in real-world workloads. However, the overhead of the contention-free scheduler is smaller. The RL-based scheduler significantly outperforms other schedulers. It could efficiently reduce the duration of long stages. The main reason is that the RL-based scheduler could set parallelism limitations for stages, which further avoids the waste of the computational resources.

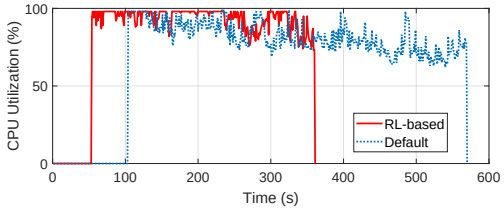


Fig. 12. The CPU utilization of a worker node.

Fig. 11(b) shows the comparison of the scheduling overhead. In this set of experiments, we vary the job batch size (therefore vary the number of stages), and record the time consumption of the contention-free and the DelayStage scheduler. The overhead of the RL-based scheduler is not shown since it frequently updates its schedule during the job execution. From the figure, we can find that the contention-free scheduler has smaller overheads and it is more efficient. The reason is that the contention-free scheduler partitions the DAG into multiple subsets of ready-to-go stages and only need to sort stages in each subset. This partitioning approach reduces the average time complexity of the contention-free scheduler.

We then investigate whether our RL-based scheduler could improve resource utilization. We use the synthetic dataset for this experiment since the jobs from the synthetic dataset have relatively simple DAG structures. We compare our RL-based scheduler with the default Spark scheduler. From this experiment, we can have a closer look at the resource utilization and have a better understanding of the RL-based scheduler. The experiment result is shown in Fig. 12. From the figure, we can find that the RL-based scheduler could start using the CPU earlier than the default Spark scheduler. The reason is that the default scheduler starts all ready-to-go stages simultaneously and causes network congestion. The RL-based scheduler could interleave the network resources. It delays the start of some parallel stages and the stage in execution could be allocated with a larger bandwidth. In addition, we also find that the RL-based scheduler could achieve a higher CPU utilization. Specifically, the default scheduler has several time intervals during which the CPU utilization is low, but the RL-based scheduler could keep a high resource utilization. Those factors make the RL-based scheduler finish the job batch in 361s. It is much faster than the default Spark scheduler which needs 560s finish the job batch.

We then execute a larger job batch from the Alibaba dataset and record the average CPU and network resource utilization

TABLE I
THE AVERAGE RESOURCE UTILIZATION

	Default	DelayStage	RL-based
Average CPU utilization	37.9%	46.1%	50.4%
Average Network utilization	43.5%	54.5%	56.4%

of a worker node. Comparing with the previous experiment, the workload is increased and the dependency relationships among stages become more complex. The utilization is shown in Table 1. From the table, we can find that the RL-based scheduler could improve both CPU and network utilization. Compared with the default Fuxi scheduler used in Alibaba trace, the RL-based scheduler can improve the CPU utilization by 33.0% and improve the network utilization by 29.7%. It also outperforms the DelayStage scheduler.

VII. CONCLUSION

In this paper, we consider the stage scheduling problem for DAG-style jobs. We notice that interleaving resource usage could reduce the makespan and improve cluster resource utilization. We theoretically analyze the scheduling for perfectly parallel stages and convert our problem into a DAG shop scheduling problem. For perfectly parallel stages, we propose a contention-free scheduler. We also notice that the practical jobs usually have very few perfectly parallel stages, and the contention-free scheduling might waste computational resources. For general stage scheduling, we control the resource contention level by setting parallelism limitations and delaying the start of some stages. We use a reinforcement learning (RL) based scheduler to adaptively adjust parallelism limitations and the start time. We use the real-world Alibaba trace data to evaluate our contention-free and RL-based scheduler. Experiment results show that the contention-free scheduler it more time-efficient and the RL-based scheduler can achieve higher resource utilization.

REFERENCES

[1] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 261–276.

[2] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 455–466, 2014.

[3] Z. Zhang, C. Li, Y. Tao, R. Yang, H. Tang, and J. Xu, "Fuxi: a fault-tolerant resource management and job scheduling system at internet scale," in *Proceedings of the VLDB Endowment*, vol. 7, no. 13. VLDB Endowment Inc., 2014, pp. 1393–1404.

[4] A. Vulimiri, C. Curino, P. B. Godfrey, T. Jungblut, J. Padhye, and G. Varghese, "Global analytics in the face of bandwidth and regulatory constraints," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, 2015, pp. 323–336.

[5] R. Grandl, S. Kandula, S. Rao, A. Akella, and J. Kulkarni, "GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 81–97.

[6] Z. Hu, B. Li, C. Chen, and X. Ke, "Flowtime: Dynamic scheduling of deadline-aware workflows and ad-hoc jobs," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 929–938.

[7] P. Brucker and P. Brucker, *Scheduling algorithms*. Springer, 2007, vol. 3.

[8] S. M. Johnson, "Optimal two-and three-stage production schedules with setup times included," *Naval research logistics quarterly*, vol. 1, no. 1, pp. 61–68, 1954.

[9] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, 1967, pp. 483–485.

[10] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 2019, pp. 270–288.

[11] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*, 2010, pp. 265–278.

[12] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems*, 2017, pp. 6348–6358.

[13] Z. Hu, B. Li, Z. Qin, and R. S. M. Goh, "Job scheduling without prior information in big data processing systems," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 572–582.

[14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4. ACM, 2014, pp. 455–466.

[15] S. Liu, H. Wang, and B. Li, "Optimizing shuffle in wide-area data analytics," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 560–571.

[16] W. Shao, F. Xu, L. Chen, H. Zheng, and F. Liu, "Stage delay scheduling: Speeding up dag-style data analytics jobs with resource interleaving," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–11.

[17] C. Delimitrou and C. Kozyrakis, "Paragon: Qos-aware scheduling for heterogeneous datacenters," in *ACM SIGPLAN Notices*, vol. 48, no. 4. ACM, 2013, pp. 77–88.

[18] ——, "Quasar: resource-efficient and qos-aware cluster management," in *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1. ACM, 2014, pp. 127–144.

[19] W. Zhang, N. Zheng, Q. Chen, Y. Yang, Z. Song, T. Ma, J. Leng, and M. Guo, "Ursa: Precise capacity planning and fair scheduling based on low-level statistics for public clouds," in *49th International Conference on Parallel Processing - ICPP*, ser. ICPP '20. ACM, 2020.

[20] K. Ousterhout, C. Canel, S. Ratnasamy, and S. Shenker, "Monotasks: Architecting for performance clarity in data analytics frameworks," in *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 184–200.

[21] K. Agrawal, J. Li, K. Lu, and B. Moseley, "Scheduling parallel dag jobs online to minimize average flow time," in *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2016, pp. 176–189.

[22] C. Chekuri, A. Goel, S. Khanna, and A. Kumar, "Multi-processor scheduling to minimize flow time with ε resource augmentation," in *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*. ACM, 2004, pp. 363–372.

[23] M. Mastrolilli and O. Svensson, "(acyclic) job shops are hard to approximate," in *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. IEEE, 2008, pp. 583–592.

[24] D. B. Shmoys, C. Stein, and J. Wein, "Improved approximation algorithms for shop scheduling problems," *SIAM Journal on Computing*, vol. 23, no. 3, pp. 617–632, 1994.

[25] H. Zheng and J. Wu, "Joint scheduling of overlapping mapreduce phases: Pair jobs for optimization," *IEEE Transactions on Services Computing*, 2018.

[26] H. Zheng, Z. Wan, and J. Wu, "Optimizing mapreduce framework through joint scheduling of overlapping phases," in *2016 25th ICCCN*. IEEE, 2016, pp. 1–9.

[27] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[28] L. Weaver and N. Tao, "The optimal reward baseline for gradient-based reinforcement learning," *arXiv preprint arXiv:1301.2315*, 2013.