Accelerating DAG-Style Job Execution via Optimizing Resource Pipeline Scheduling

Yubin Duan[1](段钰斌), *Student Member, IEEE*, Ning Wang[2](王宁), *Member, IEEE*, Jie Wu[1*](吴杰), *Fellow, IEEE*

[1] *Department of Computer and Information Sciences, Temple University, Philadelphia, 19122, USA*

[2] *Department of Computer Science, Rowan University, Glassboro, 08028, USA*

E-mail: yubin.duan@temple.edu; wangn@rowan.edu; jiewu@temple.edu

Abstract The volume of information that needs to be processed in big data clusters increases rapidly nowadays. It is critical to execute the data analysis in a time-efficient manner. However, simply adding more computation resources may not speed up the data analysis significantly. The data analysis jobs usually consist of multiple stages which are organized as a directed acyclic graph (DAG). The precedence relationships between stages cause scheduling challenges. General DAG scheduling is a well-known NP-hard problem. Moreover, we observe that in some parallel computing frameworks such as Spark, execution of a stage in DAG contains multiple phases that use different resources. We notice that carefully arranging the execution of those resources in pipeline can reduce their idle time and improve the average resource utilization. Therefore, we propose a resource pipeline scheme with the objective of minimizing the job makespan. For perfectly parallel stages, we propose a contention-free scheduler with detailed theoretical analysis. Moreover, we extend the contention-free scheduler for three-phase stages, considering the computation phase of some stages can be partitioned. Additionally, we are aware that job stages in real-world applications are usually not perfectly parallel. We need to frequently adjust the parallelism levels during the DAG execution. Considering reinforcement learning (RL) techniques can adjust the scheduling policy on the fly, we investigate a scheduler based on RL for online arrival jobs. The RL-based scheduler can adjust the resource contention adaptively. We evaluate both contention-free and RL-based schedulers on a Spark cluster. In the evaluation, a real-world cluster trace dataset is used to simulate different DAG styles. Evaluation results show that our pipelined scheme can significantly improve CPU and network utilization.

Keywords data center clusters, directed acyclic graph scheduling, makespan minimization, pipelines

1 Introduction

Nowadays, the data volume in the big data industry grow rapidly. It becomes more and more critical to efficiently reduce the makespan of data analysis jobs. 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 is 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

Regular Paper

A preliminary version was published in the Proceedings of IEEE MASS 2020.

This research was supported in part by NSF grants CNS 2128378, CNS 2107014, CNS 1824440, CNS 1828363, CNS 1757533, CNS 1629746, and CNS 1651947.

*Corresponding Author

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.

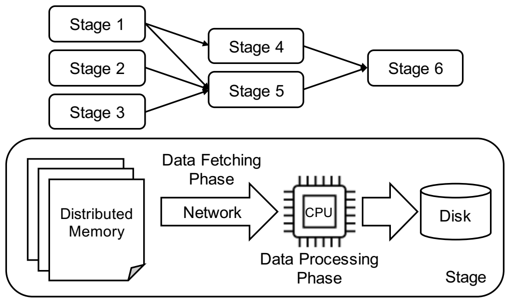


Fig.1. Illustrations of DAGs, stages and phases.

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, 2, 3, 4, 5, 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 problem which is NP-hard. A contention-free scheduler is proposed and its approximation properties are analyzed.

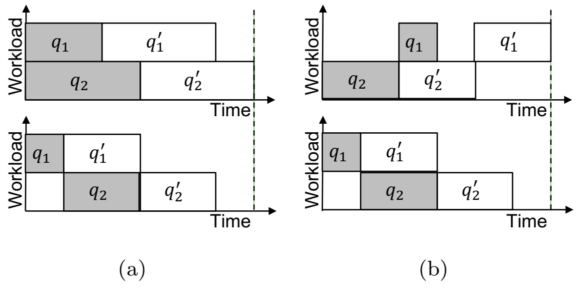


Fig.2. The challenges in scheduling DAG jobs. (a) The resource contention would increase the makespan. (b) A good execution order would reduce the makespan.

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.

This paper is an extended version of the conference paper[7] published in IEEE MASS 2020. Compared with the conference version, we add the discussion for scheduling three-phase stages. Specifically, the computation phases of some stages contain multiple operations, which means we can further divide the computation phase into two parts. Dividing a computation phase into smaller parts can help our scheduler improve resource utilization. The divided phases have shorter lengths. It is more likely to exploit those parts to fulfill the idle time blocks. In the extreme case, if we can divide the computation phase into infinite parts, our non-preemptive scheduling problem becomes a preemptive scheduling task. Resource idle blocks can be easily filled in the preemptive scheduling, while inevitable overhead would be introduced for storing and recovering computation status. Therefore, we still consider the non-preemptive scheduling problem and try to further improve the resource utilization by dividing the computation phase into multiple parts. Moreover, we notice that the number of operations in the computation phase of a Spark stage rarely exceeds two. Therefore, we investigate the scheduling problem in which the computation phase is divided into two parts. Counting the communication phase, we extend our contention-free scheduler for three-phase 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. In our analysis, we convert our problem into a DAG shop problem. A contention-free scheduler is proposed. Its approximation properties are analyzed.
- We notice that the computation phases of some stages contains two operations and can be partitioned accordingly. We further extend our contention-free scheduler for those three-phase stages.
- 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.

The remainder of the paper is organized as follows. Section 2 introduces the model of DAG-style jobs and formulates our stage scheduling problem. Section 3 discusses the perfectly parallel stage scheduling in which the speedup of a stage is proportional to the percentage of resources allocated to it. A contention-free scheduler is proposed to schedule those stages. Section 4 extends the contention-free scheduler for three-phase stages, considering the computation phases of some stages can be further spitted. Section 5 investigates a more practical case in which stages have non-linear speedups when running in parallel and introduces a RL-based scheduler. Section 6 explains our experiment settings and results. Section 7 reviews related work. Finally, Section 8 concludes the paper.

2 Models

2.1 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 DAG.

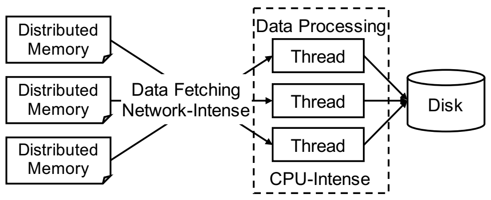


Fig.3. The procedure of executing a stage on a Spark cluster.

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 the stage 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, machines in the cluster would shuffle and 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.

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.

2.2 Notations

Before we formulate our stage scheduling problem, we first introduce the notations we use. 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, $S = \{s_1, s_2, \dots, s_n\}$, where n is the number of the stages in the job. The directed edge set E of the graph represents the dependency relations of the 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, the 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 four 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 the 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 5.

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 Λ 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 adjusting the policy Λ.

2.3 Problem Formulation

In this paper, we aim to design a scheduler which can 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:

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

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

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

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

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

(1) shows our objective of minimizing the makespan of the job execution. (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. (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. (4) is the bandwidth constraint. (5) is the schedule constraint. Each stage $s_i \in S$ should be scheduled and processed by some executors.

2.4 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 brings 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 3. 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.

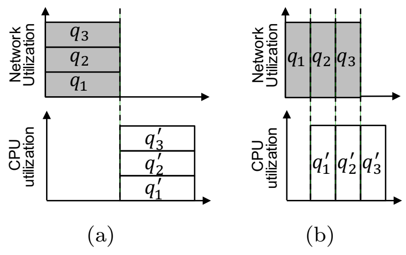


Fig.4. A motivation of scheduling ideal stages in a pipelined manner. (a) Splitting resources. (b) Contention-free

3 Scheduling for Perfectly Parallel Stages

In this section, we investigate an ideal case where the speedup of parallel execution is linear to the number of working machines. A contention-free scheduler is proposed and its approximate ratio is discussed theoretically. The contention-free scheduler provides us useful insights and inspirations for our extended scheduler introduced in Section 4.

3.1 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 is 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 brings 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 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-hardness 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 [8] 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 can 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, these 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 these 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.

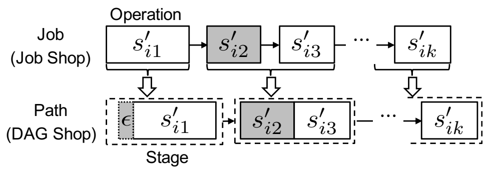


Fig. 5. Converting operations in the job shop to stages in the DAG shop.

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 processed 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 [8], our problem is also NP-hard. ■

3.2 Contention-free Scheduling

Inspired by the list scheduling approach[9], we propose a contention-free scheduling algorithm for perfectly parallel stages. In the list scheduling, one task is processed at a time and each task acquires all resources during its execution. The scheduler assigns priorities to tasks to determine the execution sequence. In our paper, the contention-free scheduling means both data fetching and data 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 data 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 [10] 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 makespan of all feasible sequences and comparing them cannot be done in linear time. Johnson's rule is a method of scheduling flow shop problems. Without the precedence constraints, it can optimally solve our stage scheduling problem.

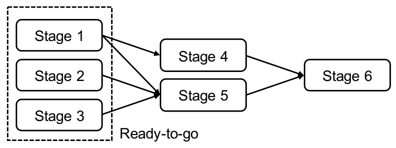


Fig.6. The illustration of the ready-to-go stages.

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 ready-to-go stage set contains s_1, s_2, and s_3. After stages s_1, s_2, and s_3 are processed, s_4 and s_5 become 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.

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
```

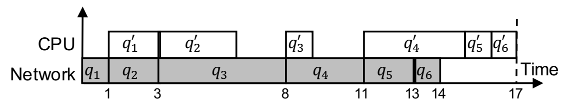
The detailed procedures of our algorithm are illustrated in Algorithm 1. Lines 1-2 calculate the phase lengths for all stages and initialize the schedule list. In lines 3-7, we iteratively schedule a set of ready-to-go stages. Line 4 finds the ready-to-go stages that have no income edges in G. Lines 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 length of each phase used in our example is shown in Table 1. The detailed steps of our scheduling algorithm are shown as follows. In the first iteration, s_1, s_2, and s_3 are ready-to-go stages. According to the definition, s_1 and s_2 is computation-heavy and s_3 is shuffle-heavy. For s_1 and s_2, we sort them based on their communication phase length. In our example, $l_1 < l_2$, and s_1 is placed before s_2. s_3 is placed after s_1 and s_2 since it is shuffle-heavy. Then, they are removed from the DAG and s_4, s_5 become ready-to-go. s_4 has a longer data processing phase and should be processed before s_5. After s_4 and s_5 are scheduled, s_6 has no predecessors and is concatenated 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). With our scheduling, the makespan of the input DAG is 17.

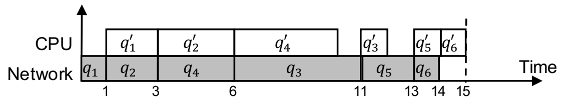
Table 1. Lengths of Computation and Communication Phases

	s_1	s_2	s_3	s_4	s_5	s_6
l_i	1	2	5	3	2	1
l'_i	2	3	1	4	1	1

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 schedule s_1, s_2, and s_3 before s_4 and s_5 in Fig.6. It introduces a precedence restrictions $s_3 \rightarrow s_4$, which is not necessary. Adding those restrictions could let the scheduler miss the optimal solution. As shown in Fig.7(b), the optimal schedule is $(s_1, s_2, s_4, s_3, s_5, s_6)$. Notably, the stage s_4 is processed before s_3.



(a)



(b)

Fig. 7. An example of our scheduling algorithm. (a) Our contention-free schedule (s_1,s_2,s_3,s_4,s_5,s_6). (b) The optimal schedule (s_1,s_2,s_4,s_3,s_5,s_6).

Our contention-free scheduler is 2-approximate for perfectly parallel stage scheduling. It is 3/2-approximate if the data fetching and the 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 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. ∎

4 Scheduling for Three-phase Stages

Besides scheduling for stages with two phases, we further investigate a contention-free scheduler for three-phase stages. Specifically, we notice that some stages in a Spark job may contain more than one operation. For example, the ShuffleMapStage contains two operations — map and filter. In theory, we can split the computation phase into multiple parts and extend our pipeline scheme accordingly. As shown in the previous subsection, the computation phase of a stage may be blocked until its precedent communication phase is finished. The size of the idle block may not be large enough to insert a computation phase from another stage. It causes an inevitable idle time of computation resources and reduces the overall resource utilization. If we can break the computation phase into multiple segments, the smaller parts are more likely to be allocated to those idle blocks. Ideally, if we can divide the computation phase into infinite segments like preemptive scheduling, the idle slots with arbitrary lengths can be easily fulfilled. However, preemptive scheduling would introduce additional overhead for storing and recovering the computation status. Therefore, we consider dividing the computation phase instead of following the preemptive scheduling approach. Moreover, different types of computation parts usually have different speedup ratios when executing in parallel, which provides additional scheduling opportunities. Motivated by the observations, we investigate the scheduling problem for stages with three phases, i.e., the original computation phase is further divided into two parts. We can hardly find a realistic application in which a stage contains more than two types of operations. Therefore, the more general case where the computation phase is divided into more than two parts is not discussed.

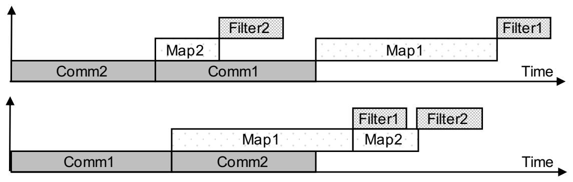


Fig.8. Scheduling for three phase tasks.

An illustration of three-phase stages is shown in Fig.8. The figure compares the makespan of two different scheduling sequences. In the example, there are two stages s_1 and s_2 with no precedence restrictions. Each stage has three phases: communication, map, and filter. These phases have to be executed in sequence. The number i labeled after each phase indicates that the phase belongs to stage s_i. If we execute s_2 first and then s_1, the total makespan is suboptimal as shown in Fig.8. To improve the resource utilization and reduce the makespan, the optimal schedule for this example is to process s_1 and then s_2. It is necessary to carefully decide the processing schedule of three-phase stages if we hope to minimize the makespan. Fig.8 shows that the execution of communication, map-, and filter-phase in the same stage is not overlapped. It is worth noting that even each stage has three phases as shown in Fig.8, there are still two types of resources. Map and filter operations are executed using computation resources.

The extended scheduling problem for three-phase stages is NP-hard. Similar to the proof of Theorem 1, we can show the NP-hardness of the extended problem by a reduction from the job shop problem with three machines, which is NP-hard. The only difference in the proof is that we need to group three operations in a job shop scheduling into a stage. Three-phase stages provide more scheduling opportunities. Explicitly exploring all possible operation sequences cannot be completed in polynomial time. Therefore, we investigate another heuristic for the scheduling problem for three-phase stages as an extension of our contention-free scheduler.

To formulate the scheduling problem for three-phase stages, we need to extend our notations. For stage s_i, we use q_i, q'_i, and q''_i to denote its three phases, respectively. Among them, q_i and q'_i are computation phases after partition, and q''_i represents the communication phase. Notably, the meaning of q'_i in three-phase stages is different from that in regular two-phase stages. In two-phase stages, q'_i represents the communication phase, while it stands for the second computation phase in three-phase stages. We update the meaning of q'_i for three-phase stages to emphasize the processing sequence of phases, i.e., q_i is processed before q'_i which is executed before q''_i. Let l_i, l'_i, and l''_i denote the length of q_i, q'_i, and q''_i, respectively. We still use t_i and t'_i to denote the start time and the completion time of the stage s_i. The difference that the formulation of t'_i is changed to $t'_i = t_i + l_i + l'_i + l''_i$. The duration of the whole DAG-style job is still denoted as τ, which is the time length between the start of the first stage and the completion of the last stage. For the extended problem, our objective is still to minimize the makespan τ of the whole DAG-style job.

We cannot directly apply Johnson's rule to solve the three-phase stage scheduling problem. The major challenge is that we can no longer cluster computation or communication stages based on relative lengths of computation and communication phases. Following the idea of Johnson's rule, we investigate an approach that adaptively merges the middle phase q'_i with q_i or q''_i. Specifically, the insight behind Johnson's rule is to greedily minimize the resource idle time. In two-phase stage scheduling, the computation resources are blocked by communication. Hence, for communication-heavy stages, Johnson's rule assigns higher priority to the stages with shorter computation phases, which aims to start communication phases as soon as possible. For computation-heavy stages, Johnson's rule chooses to execute stages with longer communication phases to maximize the overlap between usages of CPU and network resources. Following similar design principles, we focus on relative lengths of the first and last phases in a stage. If the length of the first phase q_i is shorter than the length of the last phase q''_i, then we merge the middle phase q'_i with q_i. For a set of such stages, we sort them for ascending order of $l_i + l'_i$. The reason is that the bottleneck phase should start as early as possible. For other stages where $l_i >= l''_i$, we merge q'_i with q''_i and sort them in descending order of $l'_i + l''_i$. In this way, we adaptively merge the middle phases for three-phase stages. Our contention-free scheduler is extended accordingly. Note that merging the middle phases is only used to determine the scheduling of phases. It is a symbolic step. In real execution, there is no phase merging and each phase is processed separately.

Algorithm 2 Extended Scheduling Algorithm for Three-phase Stages

```
Input: The DAG $G=(S,E)$
Output: The scheduling for DAG stages in $S$
1: Initialize the schedule list $L \leftarrow \emptyset$
2: while $S$ is not empty do
3:     Ready-to-go stage set $S' \leftarrow \{s_i \in S | (s_j, s_i) \notin E, \forall s_j \in S\}$
4:     $S_1 \leftarrow \{s_i \in S' | l_i > l''_i\}$, $S_2 \leftarrow \{s_i \in S' | l_i \leq l''_i\}$
5:     $L_1 \leftarrow$ Sort $s_i \in S_1$ for descending order of $l'_i + l''_i$
6:     $L_2 \leftarrow$ Sort $s_i \in S_2$ for ascending order of $l_i + l'_i$
7:     $L \leftarrow L || L_2 || L_1$
8:     $S \leftarrow S \setminus S'$
9: return $L$
```

The procedures of the extended algorithm are shown in Algorithm 2. We first initialize the scheduling list L as an empty set in line 1. Then, we iteratively schedule ready-to-go stages in the following loop. The set S' that contains current ready-to-go stages is updated in line 3. Line 4 splits set S' according to the relative lengths of the first and last phases of stages in the set. Lines 5 and 6 adaptively merge the middle phases and sort the merged stages based on Johnson's rule. For stages in S_1 where $l_i > l''_i$, they are sort decreasingly based on the value $l'_i + l''_i$ in line 5. For stages in S_2 where $l_i \leq l''_i$, line 6 sorts them increasingly based on the value of $l_i + l'_i$. Line 7 updates the schedule L by appending lists L_2 and L_1 after L. L_1 is concatenated after L_2. Line 8 updates the DAG by removing the stages that have been scheduled. Finally, L is returned as the final schedule for the three-phase stages.

Our major observations for designing the contention-free scheduler can be summarized as follows. Firstly, we notice that there are no benefits to simultaneously run multiple perfectly parallel stages. Therefore, we propose to allocate all resources to one stage at a time. Then, following the list scheduling approach, our contention-free scheduler determines the processing priority of each stage before executing the DAG-style job. To set processing priorities, we apply Johnson's rule on two-phase stages. To deal with the precedence constraints in the DAG, we define the ready-to-go stage set inspired by the topological sort. Combining these ideas, our contention-free scheduler is 3/2-approximate for stages with equal-length phases.

5 Scheduling for General Stages

From the analysis in previous sections, we notice that the parallelism level is critical in scheduling. In ideal cases, the parallelism level should be set as large as possible. However, in real-world applications, allocating more computation resources does not lead to a linear increase in the speedup ratio. It means that we need to adaptively adjust the parallelism level of each stage for general DAGs.

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 these stages, allocating all executors to a ready-to-go stage as the contention-free scheduler is a waste. The scheduler should adaptively adjust the parallelism level of each stage. The parallelism level can be controlled by setting 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. RL is a useful tool to adaptively adjust the scheduling policy on the fly. In this section, we first show the non-linear speedup of general DAG stages. Then, we introduce a RL-based scheduler for general stage scheduling and present the design details of the RL agent.

5.1 Speedup of General Jobs

We first test the speedup of general DAG stages on the Spark server. Fig.9 shows the speedup of two different jobs from the TPC-H dataset[1]. From Fig.9, 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.

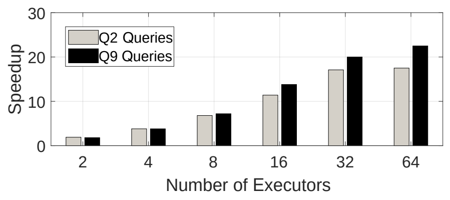


Fig.9. Speedups of two different jobs on the Spark cluster.

It is difficult to theoretically model the speedup in practice. Although the Amdahl's law [11] shows a speedup model, the percentage of the 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.

5.2 A RL-based Scheduler for General Stage Scheduling

We adapt the RL framework in [12] to generate schedules for general stages. Different from the RL agent in [12], we consider to control the resource contention by adjusting the start time t_i of each stage s_i. The framework of the RL-based scheduler is shown in Fig.10. It consists of a RL agent and the environment. The agent observes the state from the environment and generates the schedule as an action. The environment is the Spark engine running on the data center cluster.

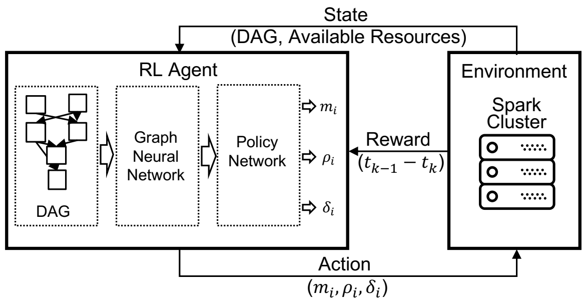


Fig.10. The RL framework for general stage scheduling.

The RL-based scheduler mainly consider three aspects to improve the resource utilization and optimize the makespan. Specifically, the action space of the RL agent contains three dimensions. Because of the non-linear speedup ratios of general stages, we need to adaptively adjust the parallelism level for general DAG stages. We use parameter m_i to denote the parallelism level limitation for stage s_i. Our RL-based scheduler also follows the list scheduling approach. We use parameter ρ_i to denote the priority of s_i. The priority is used to adjust the scheduling sequence of DAG stages. Moreover, we need to reduce the communication contention to minimize the makespan. As shown in [13], delay scheduling can reduce the communication contention level by interleaving the usage of network resources. Therefore, we propose to insert delay time

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

before each stage. The RL agent uses parameter δ_i to adjust the delay time of each stage s_i.

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[13].

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, these 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 these stages.

Unlike computational resources, it is not convenient to set bandwidth limitations. Inspired by the delay scheduling [13], 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) [14] 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 wallclock 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 is 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.

Our major observations for scheduling general stages are summarized as follows. Firstly, we notice that we should set a parallelism limitation for general stages since they have non-linear speedup ratios. Also, we need to adaptively adjust the parallelism level of each stage on the fly. To achieve these, we adapt a RL-based scheduler to schedule general DAG stages.

6 Experiment

6.1 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 clusters. 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 five stages.

Before the experiment, we first illustrate the percentage of parallel stages in the Alibaba dataset. Fig.11 shows the cumulative distribution function (CDF) of the Alibaba dataset. Fig.11(a) shows the distribution of the number of stages in a job. In total, the dataset contains 2,775,025 jobs. From Fig.11(a), we can find that most of these jobs have more multiple stages. More than 80% of these jobs have more than one stage. Besides, we use topological sort to analyze 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.11(b) shows the distribution of stage duration that executes in the production clusters. It shows the distribution of stage sizes to some extent.

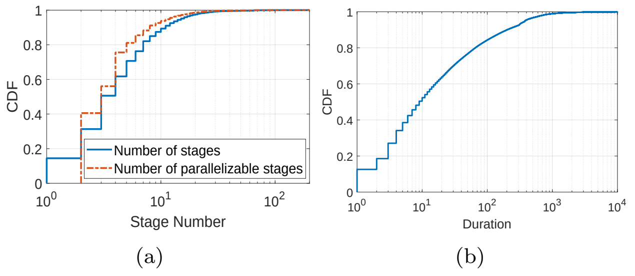


Fig.11. Cumulative distribution function of the Alibaba dataset. (a) The stage number distribution. (b) The stage duration distribution.

6.2 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 four 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 [15] 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 [16]. On our local PC, each training iteration takes about five seconds on average. Considering the initial policy of the RL agent is ran-

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

domly 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.

6.3 Experiment Result

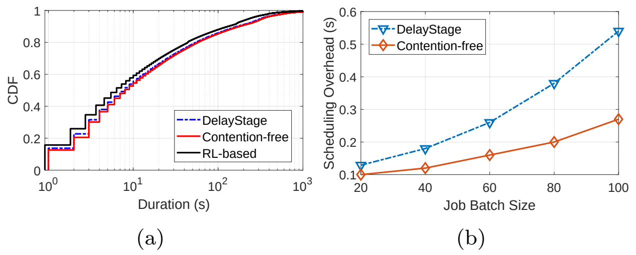


Fig.12. Performance evaluation on the Alibaba dataset. (a) The stage duration distribution. (b) Comparison on the overhead.

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 [17]. 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.12(a). Lines on the left have better performance. From Fig.12(a), 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 the 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(b) shows the comparison of the scheduling overheads. 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 Fig.12(b), we can find that the contention-free scheduler has smaller overheads and the contention-free scheduler 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.

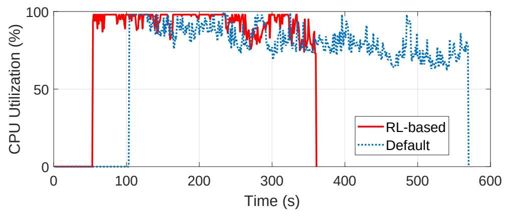


Fig.13. The CPU utilization of a worker node.

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.13. From Fig.13, 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. These factors make the RL-based scheduler finish the job batch in 361s. It is much faster than the default Spark scheduler which needs 560s to finish the job batch.

The resource utilization achieved by different schedulers is compared in Table 2. In the comparison, we use a job batch from the Alibaba dataset and record the average CPU and the network resource utilization of a worker node. Compared with the previous experiment, the workload is increased and the dependency relationships among stages become more complex. The utilization is shown in Table 2. From Fig.13, we can find that the RL-based scheduler could improve both CPU and network utilization. Compared with the default Fuxi scheduler used in the Alibaba clusters, the RL-based scheduler can improve the CPU utilization by 33.0% and improve the network utilization by 29.7%, respectively. It also outperforms the DelayStage scheduler.

Table 2. The Average Resource Utilization

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

7 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 first-in-first-out (FIFO) or Fair scheduling, Hu *et. al.*[18] proposed 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.*[12] followed a RL approach to determine the parallelism level and priority of each stage. For resource allocation, Grandl *et. al.*[2] proposed to greedily match the stage resource demands with available resources. They further defined 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. [19] and [17] also proposed to interleave resources. To improve resource utilization, [20, 21, 22] discussed solutions for private datacenters. [23] further considered the public dataset. Different from them, we develop a scheduler based on RL 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[24] considered 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 [25, 26] usually focused on simple cases. The state-of-the-art theoretical result is given in [25]. However, we cannot directly apply these theoretical results to our problem since we also consider the precedence relation between two phases in each stage. Scheduling these phases is also no trivial, and it can be viewed as a shop scheduling problem [8]. It has been proven that the job shop problem is hard to approximate [27]. Shmoys *et. al.*[28, 29] showed several RNC-approximation algorithms for shop scheduling. Although their algorithms are polynomial-time in theory, they are inefficient. Zheng *et. al.*[30] considered 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.

There are many other important studies [31, 32, 33, 34, 35, 36] that improves the scheduling algorithms in different evaluation metrics. Grandl *et. al.*[31] focused on the fairness issues in scheduling and discusses the trade-off between fairness and job completion time. Their scheduling can maintain long-term fairness and improve the job completion time by sacrificing the short-time fairness. [32] discussed the job miss ratios. Their scheduler can reduce the deadline miss rate by providing guarantees on job latency. [33] showed the approximation bounds of scheduling algorithms for online arrival jobs. [34] presented a scheduler that minimizes the total weighted response time for online jobs in edge-cloud computing scenarios. [35] analyzed the worst-case makespan of a conditional DAG task under list scheduling. [36] illustrated a learning algorithm for the distributed DAG scheduling problem.

8 Conclusion

In this paper, we considered the stage scheduling problem for DAG-style jobs. We noticed that interleaving resource usage could reduce the makespan and improve cluster resource utilization. Our theoretical analysis for scheduling perfectly parallel stages can be used to calculate approximation ratios for DAG shop scheduling problems. The contention-free scheduler proposed in this paper can be used to schedule perfectly parallel stages. We also noticed that the practical jobs usually have very few perfectly parallel stages, and the contention-free scheduling might waste computational resources. For the general stage scheduling, our RL-based scheduler can dynamically control the resource contention level by adaptively setting parallelism limitations and delaying the start time of some stages. We used the real-world dataset to evaluate our contention-free and RL-based scheduler. The content-free scheduler can achieve a 3/2 approximate ratio with relatively small time complexity. With proper training, the RL-based scheduler can achieve higher resource utilization compared with default and contention-free schedulers.

References

[1] Isard M, Prabhakaran V, Currey J, Wieder U, Talwar K, Goldberg A. Quincy: fair scheduling for distributed computing clusters. In *Proc. the ACM SOSP*, October 2009, pp.261–276.

[2] Grandl R, Ananthanarayanan G, Kandula S, Rao S, Akella A. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review*, 2014, 44(4): 455–466.

[3] Zhang Z, Li C, Tao Y, Yang R, Tang H, Xu J. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. In *Proc. the VLDB Endowment*, September 2014, pp.1393–1404.

[4] Vulimiri A, Curino C, Godfrey P B, Jungblut T, Padhye J, Varghese G. Global analytics in the face of bandwidth and regulatory constraints. In *Proc. the USENIX NSDI*, May 2015, pp.323–336.

[5] Grandl R, Kandula S, Rao S, Akella A, Kulkarni J. GRAPHENE: Packing and dependency-aware scheduling for data-parallel clusters. In *Proc. the USENIX OSDI*, November 2016, pp.81–97.

[6] Hu Z, Li B, Chen C, Ke X. Flowtime: Dynamic scheduling of deadline-aware workflows ad-hoc jobs. In *Proc. the IEEE ICDCS*, July 2018, pp.929–938.

[7] Duan Y, Wang N, Jie W. Reducing Makespans of DAG Scheduling through Interleaving Overlapping Resource Utilization. In *Proc. the IEEE MASS*, December 2020, pp.392–400.

[8] Brucker P. Scheduling algorithms. Springer, 2007.

[9] Wang H, Sinnen O. List-scheduling versus cluster-scheduling. *IEEE Transactions on Parallel, Distributed Systems*, 2018, 29(8): 1736–1749.

[10] Johnson S M. Optimal two-and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1954, 1(1): 61–68.

[11] Amdahl G M. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. the Spring Joint Computer Conference*, April 1967, pp.483–485.

[12] Mao H, Schwarzkopf M, Venkatakrishnan S B, Meng Z, Alizadeh M. Learning scheduling algorithms for data processing clusters. In *Proc. the ACM SIGCOMM*, August 2019, pp.270–288.

[13] Zaharia M, Borthakur D, Sen S J, Elmeleegy K, Shenker S, Stoica I. Delay scheduling: a simple technique for achieving locality, fairness in cluster scheduling. In *Proc. the ACM EuroSys*, April 2010, pp.265–278.

[14] Khalil E, Dai H, Zhang Y, Dilkina B, Song L. Learning combinatorial optimization algorithms over graphs. In *Proc. the NeurIPS*, December 2017, pp.6348–6358.

[15] Williams R J. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992, 8(3-4): 229–256.

[16] Weaver L, Tao N. The optimal reward baseline for gradient-based reinforcement learning. arXiv:1301.2315, 2013. https://arxiv.org/abs/1301.2315, Jan 2013.

[17] Shao W, Xu F, Chen L, Zheng H, Liu F. Stage Delay Scheduling: Speeding up DAG-style Data Analytics Jobs with Resource Interleaving. In *Proc. the IEEE ICPP*, August 2019, pp.1–11.

[18] Hu Z, Li B, Qin Z, Goh R S M. Job scheduling without prior information in big data processing systems. In *Proc. the IEEE ICDCS*, June 2017, pp.572–582.

[19] Liu S, Wang H, Li B. Optimizing shuffle in wide-area data analytics. In *Proc. the IEEE ICDCS*, June 2017, pp.560–571.

[20] Delimitrou C, Kozyrakis C. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 2013, 48(4): 77–88.

[21] Vavilapalli V K, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proc. the ACM SoCC*, October 2013, pp.1–5.

[22] Delimitrou C, Kozyrakis C. Quasar: resource-efficient, QoS-aware cluster management. *ACM SIGARCH Computer Architecture News*, 2014, 42(1): 127–144.

[23] Zhang W, Zheng N, Chen Q, Yang Y, Song Z, Ma T, Leng J, Guo M. URSA: Precise Capacity Planning, Fair Scheduling Based on Low-Level Statistics for Public Clouds. In *Proc. the ACM ICPP*, August 2020, pp.1–11.

[24] Ousterhout K, Canel C, Ratnasamy S, Shenker S. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proc. the ACM SOSP*, October 2017, pp.184–200.

[25] Agrawal K, Li J, Lu K, Moseley B. Scheduling parallel DAG jobs online to minimize average flow time. In *Proc. the ACM-SIAM SODA*, January 2016, pp.176–189.

[26] Chekuri C, Goel A, Khanna S, Kumar A. Multi-processor scheduling to minimize flow time with ε resource augmentation. In *Proc. the ACM STOC*, June 2004, pp.363–372.

[27] Mastrolilli M, Svensson O. Acyclic job shops are hard to approximate. In *Proc. the IEEE FOCS*, October 2008, pp.583–592.

[28] Shmoys D B, Stein C, Wein J. Improved approximation algorithms for shop scheduling problems. *SIAM Journal on Computing*, 1994, 23(3): 617–632.

[29] Zheng H, Wu J. Joint Scheduling of Overlapping MapReduce Phases: Pair Jobs for Optimization. *IEEE Transactions on Services Computing*, 2018, : 1–1.

[30] Zheng H, Wan Z, Wu J. Optimizing mapreduce framework through joint scheduling of overlapping phases. In *Proc. the IEEE ICCCN*, August 2016, pp.1–9.

[31] Grandl R, Chowdhury M, Akella A, Ananthanarayanan G. Altruistic scheduling in multi-resource clusters. In *Proc. the USENIX OSDI*, November 2016, pp.65–80.

[32] Ferguson R D, Bodik P, Kandula S, Boutin E, Fonseca R. Jockey: guaranteed job latency in data parallel clusters. In *Proc. the ACM EuroSys*, April 2012, pp.99–112.

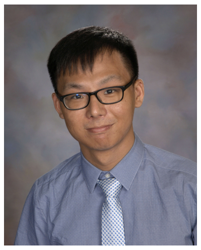
[33] Im S, Kell N, Kulkarni J, Panigrahi D. Tight bounds for online vector scheduling. In *Proc. the IEEE FOCS*, October 2015, pp.525–544.

[34] Tan H, Han Z, Li X-Y, Lau F CM. Online job dispatching, scheduling in edge-clouds. In *Proc. the IEEE INFOCOM*, May 2017, pp.1–9.

[35] Marchetti-Spaccamela A, Megow N, Schlöter J, Skutella M, Stougie L. On the Complexity of Conditional DAG Scheduling in Multiprocessor Systems. In *Proc. the IEEE IPDPS*, May 2020, pp.1061–1070.

[36] Jinhong L, Xijun L, Mingxuan Y, Jianguo Y, Jia Z. Learning to Optimize DAG Scheduling in Heterogeneous Environment. arXiv:2103.06980, 2021. https://arxiv.org/abs/2103.06980, March 2021.

Yubin Duan received his B.S. degree in mathematics and physics from University of Electronic Science and Technology of China, Chengdu, China, in 2017. He is currently a Ph.D. candidate in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania, USA. His current research focuses on scheduling algorithms for distributed systems and parallel computing.



Ning Wang is currently an assistant professor in the Department of Computer Science at Rowan University, Glassboro, NJ. He received his Ph.D. degree in the Department of Computer and Information Sciences at Temple University, Philadelphia, PA, USA, in 2018. He obtained his B.E. degree in School of Physical Electronics at University of Electronic Science and Technology of China, Chengdu, Sichuan, China, in 2013. He currently focuses on communication and computation optimization problems in Internet-of-Things systems and operation optimization in Smart Cities applications. He has published nearly thirty papers in high-impact networking conferences and journals, such as, IEEE ICDCS, IEEE INFOCOM, IEEE/ACM IWQoS, IEEE Transactions on Big Data, Journal of Parallel and Distributed Computing, etc. He has served as a program committee member for top international conferences such as IEEE ICDCS, IEEE WCNC, etc., and reviewers for premier journals such as IEEE TPDS, TWC, TMC. TITS, TOIT, TITS, TSC, etc.

Jie Wu is the Director of the Center for Networked Computing and Laura H. Carnell professor at Temple University. He also serves as the Director of International Affairs at College of Science and Technology. He served as Chair of Department of Computer and Information Sciences from the summer of 2009 to the summer of 2016 and Associate Vice Provost for International Affairs from the fall of 2015 to the summer of 2017. Prior to joining Temple University, he was a program director at the National Science Foundation and was a distinguished professor at Florida Atlantic University. His current research interests include mobile computing and wireless net- works, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Service Computing and the Journal of Parallel and Distributed Computing. Dr. Wu was general co-chair for IEEE MASS 2006, IEEE IPDPS 2008, IEEE ICDCS 2013, ACM MobiHoc 2014, ICPP 2016, and IEEE CNS 2016, as well as program co-chair for IEEE INFOCOM 2011 and CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.