Computer Science and Information Systems 22(3):839–858 https://doi.org/10.2298/CSIS240831018Z

Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy in Spark

Yanhao Zhang, Congyang Wang, Xin He* , Junyang Yu , Rui Zhai and Yalin Song

School of Software, Henan University
Kaifeng,Henan,China,475000
zhangyanhao@henu.edu.cn
wangcongyang@henu.edu.cn
hexin@henu.edu.cn
jyyu@henu.edu.cn
zr@henu.edu.cn
122648935@qq.com

Abstract. For solving the low CPU and network resource utilization in the task scheduler process of the Spark and Flink computing frameworks, this paper proposes a Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy (DRTS). This algorithm can schedule parallel tasks in a pipelined fashion, effectively improving the system resource utilization and shortening the job completion times. Firstly, based on historical data of task completion times, we stagger the execution of tasks within the stage with the longest completion time. This helps optimize the utilization of system resources and ensures the smooth completion of the entire pipeline job. Secondly, the execution tasks are categorized into CPU-intensive and non-CPU-intensive phases, which include network I/O and disk I/O operations. During the non-CPU-intensive phase where tasks involve data fetch, parallel tasks are scheduled at suitable intervals to mitigate resource contention and minimize job completion time. Finally, we implemented DRTS on Spark 2.4.0 and conducted experiments to evaluate its performance. The results show that compared to DelayStage, DRTS reduces job execution time by 3.18% to 6.48% and improves CPU and network utilization of the cluster by 6.33% and 7.02%, respectively.

Keywords: Job execution time, delay-aware, Spark, task scheduler.

1. Introduction

With the rapid development of the Internet, the size of data has increased dramatically, creating a greater demand for real-time data processing. Big data analytics has aroused the interest of scholars because of its ability to deal with large data sets. Many open-source parallel processing frameworks, such as MapReduce[26] , Hadoop[8], Storm[3] to the later Spark[2] and Flink[1], have been developed to handle large data volumes. These frameworks have evolved through various stages, including the Map-Reduce model, the DAG model, the streaming model, and the real-time model.

These computing frameworks break a job into multiple tasks and assign them to work nodes for large-scale data processing. Task scheduling efficiency is a major bottleneck that affects the framework's performance[10].

* Corresponding author

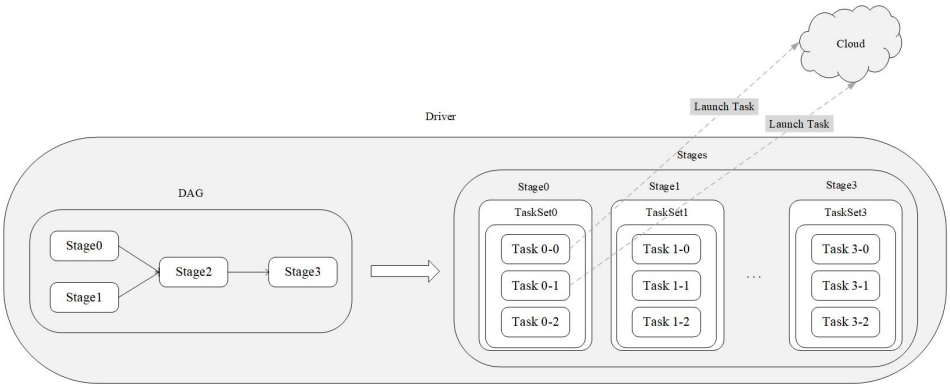

Fig. 1. From DAG scheduling to task scheduling

Moreover, as data centers continue to expand in scale, their energy consumption issues have become increasingly prominent. Research indicates that the energy consumption of cloud data centers accounts for a significant portion of overall operational costs[19]. Adopting efficient task scheduling algorithms can not only enhance computational performance but also significantly reduce energy consumption[21][20]. In this context, optimizing Spark's task scheduling not only improves computational efficiency to meet the demands of real-time data processing but also reduces data center energy consumption through more rational resource management.

In Spark, tasks are executed in parallel, but the scheduler on the driver side sends batches of tasks serially to the target machine based on a priority selection algorithm. The target machine then performs network I/O to fetch the data and carry out the computation, as shown in Fig. 1. This process results in severe CPU and network contention due to the simultaneous submission of parallel tasks. Until these tasks are completed, subsequent tasks that depend on their results cannot be scheduled. This contention leads to unbalanced resource utilization, reduces efficiency, and prolongs task completion time.

Although tasks in Spark are executed in parallel, the scheduler on the Driver side calculates task priorities based on a selection algorithm and sends tasks serially in batches to target machines. The target machines then retrieve data through network I/O and perform computations, as illustrated in Fig. 1. However, existing studies[22] have pointed out that this scheduling method may lead to severe CPU and network resource contention. In this process, concurrently submitted parallel tasks compete for limited computing and network resources, resulting in decreased resource utilization. For example, experimental results in[13][7] show that under conditions of high concurrent task submissions, task completion times are significantly extended. Furthermore, until all tasks in these parallel stages are completed, subsequent stage tasks that depend on their results will not be scheduled. This not only leads to unbalanced resource utilization but may also cause system bottlenecks[4]. Therefore, considering resource contention factors in scheduling strategies is crucial for improving system performance and resource utilization efficiency.

In Spark's default scheduling strategy, tasks from different stages are almost all executed in parallel. These tasks compete for network resources and process data while keeping CPU resources idle, or they keep bandwidth or disk idle while contending for CPU

resources. Resource utilization fluctuates drastically between extremes of overutilization and underutilization, resulting in inefficient use.

We conducted trace sampling of the average CPU utilization and network throughput across multiple machines, as shown in Fig.2. The results indicate that the CPU and network resources of most machines are not fully utilized; the average CPU utilization and network bandwidth utilization consistently remain at low levels. Therefore, we infer that when resource contention occurs, using Spark's default scheduling strategy leads to low resource utilization during job execution in the cluster.

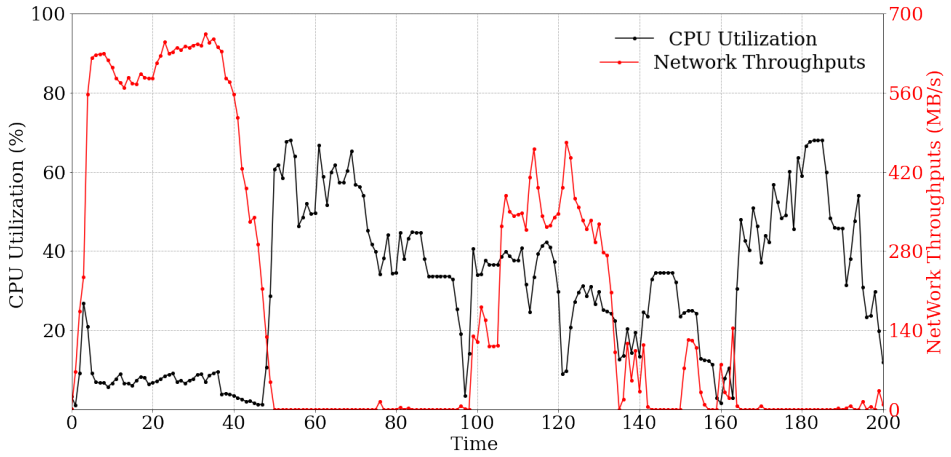

Fig. 2. The utilization of CPU and Network Throughputs

Nowadays, many scholars are focused on job scheduling [27] [11], stage scheduling [5][16], and task scheduling [17][18] to minimize job completion time and improve cluster performance. However, most of these scheduling strategies are coarse-grained and only optimize the overall execution of tasks. They do not consider that all tasks in the task set will be submitted serially to different target machines and start executing simultaneously, leading to peak resource contention.

To address this gap and further enhance cluster resource utilization while reducing job completion time, this paper approaches the problem from the perspective of task scheduling. By incorporating the scheduling priorities among parallel tasks, we analyze the various factors that affect resource utilization. By greedily selecting the tasks with the longest execution times within nodes, we determine when tasks should be scheduled.

We can illustrate this situation with a typical parallel computing task. When a task includes multiple sets of parallel parent tasks and one set of child tasks, Spark's default scheduling will relay and send parallel tasks serially. This can lead to severe network and CPU contention within the cluster during certain periods. However, the application's completion time is only related to the completion time of the longest stage. The simultaneous execution of other parallel parent task sets actually competes for resources with the longest stage, thereby affecting the final completion time.

To address this, this paper designs a Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy (DRTS). DRTS is applicable to most parallel distributed computing frameworks. Without affecting the completion of the next stage in the pipeline, DRTS delays the execution of different task sets according to the characteristics of the cluster machines they are sent to, minimizing peak resource contention and achieving resource-efficient interleaved. Integrated into the distributed framework Spark, DRTS divides the execution state of tasks into two stages: non-CPU-intensive and CPU-intensive stages.Based on the above analysis, our scheduling strategy aims to schedule tasks on different nodes at optimal times. When an execution task utilizes network resources for data fetching in the non-CPU-intensive phase, we schedule parallel tasks to execute alongside it, staggering the utilization of the cluster's resources and reducing resource competition. Therefore, the contributions of this paper can be summarized as follows:

(1) On the basis of fully considering whether there is resource contention in the cluster, this paper proposes a Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy (DRTS). This algorithm prioritizes the scheduling of jobs with long execution times according to the obtained relationship between task execution time, execution machine, and stage. Additionally, it performs interleaved execution of long and short tasks for the tasks in the stage with the longest execution time.

(2)For other parallel stages, the task execution time is calculated (including CPU-intensive and non-CPU-intensive stage times), and they are scheduled at the appropriate times. Continuous negative feedback is applied based on the task completion effectiveness, resulting in significant improvements in practical applications.

(3)Implemented a prototype model of DRTS on Spark 2.4.0 and conducted several experiments to evaluate its performance. The experiments show that DRTS enhances resource utilization and reduces job completion time.

2. Related Work

In big data processing frameworks, task scheduling is a critical step to ensure efficient resource utilization and rapid job completion. Spark's task scheduling module primarily consists of the DAGScheduler and TaskScheduler. These two components are responsible for partitioning user-submitted computational tasks into different stages according to a Directed Acyclic Graph (DAG) and assigning the computational tasks within these stages to different nodes in the cluster for parallel computation. Moreover, based on the various transformations and actions of RDDs, Spark enables users to implement strategies using complex topologies without significantly increasing the learning cost. However, because tasks are executed in parallel, this can lead to frequent usage of system resources during certain periods while they remain relatively idle at other times, thereby causing resource contention issues.

In addition to traditional scheduling strategies such as First-In-First-Out (FIFO) or Fair Scheduling (FAIR), which employ techniques like delay scheduling [24] to improve cluster performance, many studies have focused on addressing resource contention issues. For example, Xu et al. [23] proposed and developed the middleware Dagon, which, by considering and analyzing the dependency structure of jobs and heterogeneous resources, enables reasonable task allocation. They designed a sensitivity-aware task scheduling mechanism to prevent Executors from waiting for location-insensitive tasks for long pe-

riods and implemented cache eviction and prefetch strategies based on the priority of stages. Pan [15]proposed a task scheduling strategy for heterogeneous storage clusters that classifies tasks based on data locality and storage type. This approach redefines the priorities of different types of tasks according to storage device speeds and data locality to reduce task execution time. Lu et al. [14] argued that different stages have varying performance and resource requirements for different tasks, which could lead to longer overall task completion times. As a result, they proposed a task scheduling algorithm based on a greedy strategy, which balances job distribution across nodes to efficiently complete job scheduling tasks in heterogeneous clusters.

In contrast to the aforementioned works, DRTS performs task-level scheduling with finer granularity of control. DRTS assigns different priorities to tasks allocated to each node, ensuring that they are scheduled at the appropriate times, thus minimizing task completion times. Recently, Shao et al. [16] designed DelayStage, which arranges the execution of stages in a pipelined manner to minimize the completion time of parallel stages and maximize the performance of resource interleaving. However, this scheduling algorithm is coarse-grained, and simply delaying the submission time of stages still results in contention for CPU and network resources, affecting cluster execution efficiency. Duan et al. [5] argued that adding more computational resources may not significantly improve data processing speed and proposed a resource pipeline scheme aimed at minimizing job completion time. They also investigated an online scheduling algorithm based on reinforcement learning, which can adaptively adjust to resource contention. However, the reinforcement learning-based scheduler is currently only applicable to the Spark processing framework, and its compatibility with other data processing frameworks has not yet been determined.

In addition, many studies focus on stage-level scheduling strategies, while DRTS operates at the task level. DRTS addresses resource contention issues that arise from subtle time gaps in stage scheduling by performing fine-grained task execution analysis and using appropriate algorithms to schedule tasks at the optimal time, thereby interleaving system resource usage. For example, in [17], it was proposed to invoke new network-intensive tasks during non-network stages, executing two tasks in a pipelined manner by sharing the same CPU core. In [12], the design of Symbiosis, an online scheduler, allows for predicting resource utilization before task initiation and refilling compute-intensive tasks when launching network-intensive ones. In contrast, DRTS breaks tasks into CPU-intensive and network-intensive phases and achieves interleaved resource utilization by appropriately delaying task execution.

Hu et al. [10] pointed out that traditional scheduling strategies do not consider job size and designed a Shortest Job First (SJF) scheduling algorithm to avoid large jobs from blocking smaller ones. Zhang et al. [23] proposed a task scheduling method in heterogeneous server environments, based on data affinity, to minimize the maximum task completion time. He et al. [9] introduced a network-aware scheduling method, SDN, to eliminate communication barriers between the cluster computing platform and the underlying network. Zhang et al. [25] optimized scheduling using hierarchical algorithms and node scheduling algorithms, incorporating dynamic factors such as task runtime and CPU utilization on work nodes. Fu [6] utilized a bipartite graph model to propose an optimal location-aware task scheduling algorithm to reduce execution time delays and network congestion caused by cross-node data transfers.

Although a significant amount of research has been devoted to optimizing Spark's task scheduling strategies, most of the work has focused on the stage level, and some limitations remain: the details at the stage level are not fully utilized, leading to continued resource contention; dynamic resource adjustment is not performed, resulting in imbalanced resource utilization during high-concurrency task submissions; and some scheduling algorithms based on specific technologies (such as reinforcement learning) are only applicable to certain frameworks, lacking broad applicability.

To address these limitations, this paper proposes a resource-interleaved task scheduling algorithm (DRTS). DRTS overcomes the shortcomings of existing stage-level scheduling strategies by employing fine-grained task-level scheduling and dynamic resource management. It provides more efficient resource utilization and shorter job completion times, significantly improving the overall performance of Spark clusters.

3. Dual-phase Task Scheduling Strategy for the Spark Platform

When task scheduling is not properly managed, CPU and network resources are not fully utilized, leading to longer task execution times. To improve the resource utilization and job execution efficiency of the Spark platform, it is essential to address the resource contention issue.

This paper designs and implements a resource-interleaved task scheduling strategy (DRTS). The approach first greedily selects stages with the longest execution times for scheduling, in order to avoid delays in job completion. Then, by analyzing the logs generated during the job execution, detailed information for each task corresponding to the nodes is extracted. For the stage with the longest execution time, tasks are executed alternately in a way that switches between long and short tasks within the node. For the remaining stages, appropriate algorithms are used to ensure that tasks are executed at the optimal time, with continuous feedback to adjust the scheduling timing. This aims to interleave node resource usage and minimize job completion time.

3.1. Task Scheduling Optimization Strategy based on Resource Interleaving

Each task submitted by the user forms an RDD in a DAG. If an RDD requires a shuffle during the transformation process, the DAG is divided into different stages. Due to the shuffle, these stages cannot be computed in parallel, as the subsequent stages depend on the results of the preceding stages. Therefore, we divide the DAG at the boundaries of parallel stages and represent this with the symbol φ_m.

As shown in Fig. 3, the downstream shuffle stages in this splitting path are not always connected to the stages in the next splitting path. To avoid resource contention, it is necessary to appropriately delay the start time of such tasks, which will be explained in detail in Alg. 2.

In a Spark job, the Spark framework prioritizes nodes with high data locality to execute tasks, aiming to minimize data transfer overhead and enhance overall performance. While the specific scheduling of tasks is not fully determined before job execution, the Spark scheduler dynamically assigns and schedules tasks based on real-time cluster state and task demands. By analyzing log information generated during job execution, detailed task information is extracted from each node, including execution time, data volume, etc.,

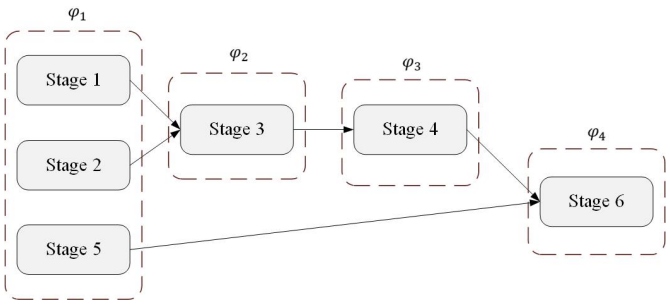

Fig. 3. Illustration of the separation of parallel phases in DAG style. Based on topological ordering, we separate the parallel stages, denoted by φ_m. The first set of parallel phases contains Stage 1, Stage 2, and Stage 5, so they can be placed in one path, i.e.,φ_1.. Stage 3 is used as the second execution path, i.e., φ_2. Stage 4's execution path is φ_3, and Stage 6's execution path is φ_4

and the scheduling node is recorded to ensure subsequent tasks are scheduled to the same node. In the case of φ_1, tasks within it are scheduled at optimal times, thereby staggering resource usage. The completion time of the longest stage in φ_1 determines the start time of φ_2. However, if stages within φ_1 are executed simultaneously, it can lead to significant resource contention, affecting execution efficiency. As shown in Fig. 4. (left), when tasks assigned to nodes execute simultaneously, it may result in severe resource contention.

We constructed an analytical model to simulate the scheduling of parallel tasks in a DAG job, determining the optimal timing for submitting parallel tasks within the job. The primary objective is to develop a scheduler that facilitates the scheduling of parallel tasks at optimal times to interleave various types of resources, thereby enhancing resource utilization and minimizing job completion time.

3.2. Task Time Statistics Strategy Based on Data Fetching and Processing

To address task resource contention, scheduling tasks at optimal times can minimize completion time by efficiently managing CPU resource requirements across different phases of the cluster. A group of Spark jobs comprises parallel computation stages, represented as $S = \{S_1, S_1, ...S_n\}$. The parallel computation phases are submitted first when there are sufficient computational resources, and each parallel phase must wait until all parallel phases have completed their computations before submitting the next phase. specifically, a stage is divided into individual tasks based on parallelism. Subsequently, the DAGScheduler submits these tasks to the TaskScheduler. We use $T_{StageId\#TaskId}$ to denote the tasks within each stage, indicating the stage number and the task number within that stage. For each task, the execution process involves several stages: initially, it requires significant network resources to fetch data; subsequently, the fetched data is processed, requiring high CPU usage; ffnally, the processed result data is written to disk. We classify the execution of a task into a non-CPU-intensive phase, denoted as $F_{i\#j}$, and a CPU-intensive phase, denoted as $P_{i\#j}$. To elucidate the execution time of tasks on the worker node W during the non-CPU-intensive phase, this paper delves into the detailed process of segmenting each task. Specifically, the non-CPU-intensive phase primarily involves

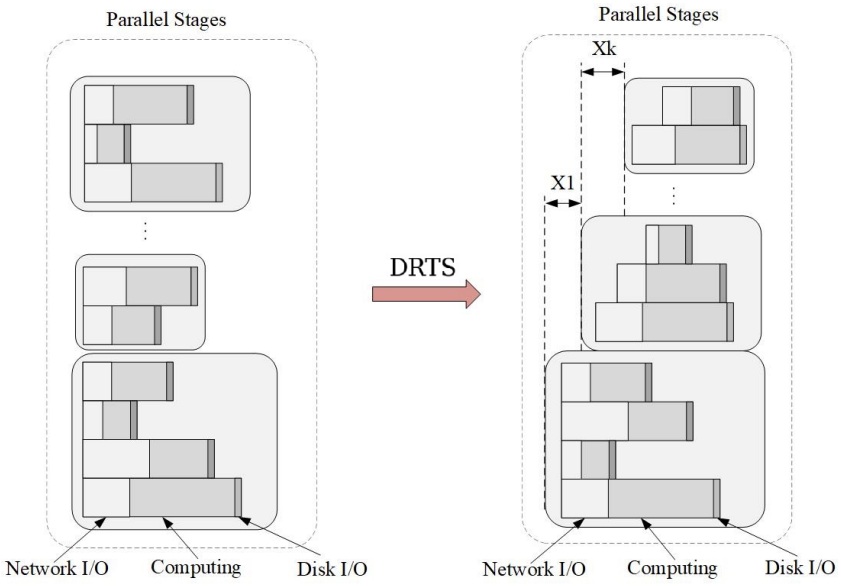

Fig. 4. Three-stage scheduling optimization strategy

data transfer, encompassing data reading and writing. Hence, the execution time of a task during the non-CPU-intensive phase on worker node W can be expressed as follows:

$$FT'_{i\#j} = T_{sr}^{i\#j} + T_{sw}^{i\#j} \tag{1}$$

The first term of Eq.(1) $T_{sr}^{i\#j}$, occurs when a task needs to get data from other nodes or file systems. The second term, $T_{sw}^{i\#j}$,occurs when a task stores the resulting data to disk after completing the computation.

Further, $T_{sr}^{i\#j}$ is calculated as:

$$T_{sr}^{i\#j} = \frac{D_r^{i\#j}}{BW_r^{i\#j}} \tag{2}$$

The process of writing data to disk involves some computation and I/O operations. However, if the data is stored only in memory, it doesn't impose a significant demand on CPU usage. Therefore, disk I/O is categorized as a non-CPU-intensive stage.

Therefore, there is a formula for $T_{sw}^{i\#j}$:

$$T_{sw}^{i\#j} = \frac{N_w^{i\#j} * B_w^{i\#j}}{BW_w^{i\#j}} + T_{dw} \tag{3}$$

$$FT_{i\#j} = \frac{D_r^{i\#j}}{BW_r^{i\#j}} + \frac{N_w^{i\#j} * B_w^{i\#j}}{BW_w^{i\#j}} + T_{dw} \tag{4}$$

Generally, the duration of writing intermediate data to disk is not extensive. For simplicity, we omit the disk I/O time when calculating the non-CPU-intensive time. Thus, we have:

$$FT_{i\#j} = \frac{D_r^{i\#j}}{BW_r^{i\#j}} + \frac{N_w^{i\#j} * B_w^{i\#j}}{BW_w^{i\#j}} \tag{5}$$

In a task, the computation time of a non-CPU intensive phase can be used to estimate the computation time of a CPU intensive phase. Given Spark's log messages provide records for the entire task completion time, noted as $T_t^{i\#j}$, once we compute the non-CPU-intensive time for the jth task in the ith stage, we can subtract this from the total task execution time to obtain the CPU-intensive time:

$$PT_{i\#j} = T_t^{i\#j} - FT_{i\#j} \tag{6}$$

We calculated the time consumption of tasks in both CPU-intensive and non-CPU-intensive phases by combining the online and offline methods described above.

4. Algorithm Implementation of Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy

In this section, we present a Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy (DRTS). The aim of DRTS is to stagger the utilization of CPU and non-CPU resources on the worker node, thereby reducing resource contention and minimizing task completion time. Initially, we compute the time set during which a task performs data fetching and processing. Subsequently, we greedily determine the optimal scheduling time for the task using the resource polling task scheduling algorithm.

4.1. Dual-stage Task Time Estimation Based Algorithm

To facilitate finer task scheduling, DRTS splits a task's execution phase into two stages: the CPU-intensive phase and the non-CPU-intensive phase. During the non-CPU-intensive phase, tasks undertake data fetching operations, which consume significant network resources and disk I/O. Conversely, the CPU-intensive phase involves extensive computation. The time intervals during which the task resides in the non-CPU-intensive and CPU-intensive phases during execution are determined by Alg. 1. The primary execution steps are divided into the following segments:

(1) We initialize two empty maps, each with NodeId as the key. The value associated with each NodeId is a map with StageId as the key and task fetch or process time as the value. These maps represent the data fetching phase (X_fetch) and data processing phase ($X\_process$) of tasks corresponding to different stages assigned to different nodes.

(2) The execution times of the data fetching phase and data processing phase during task execution, obtained from Eq. (5) and Eq. (6), are added to the result set.

(3) By traversing the collection of tasks in stage and considering the execution order and branching, the data fetching time and data processing time of different tasks are calculated and recorded in the collections X_fetch and $X\_process$.

Algorithm 1 Phase-based Task Time Estimation

Input: The total amount of data for the i#jth task $D_r^{i\#j}$ network bandwidth BW_w , the time taken to execute the completed task $T_t^{i\#j}$, the collection of execution information for various stages obtained from historical data, denoted as set S.

Output: Time maps X_fetch,$X\_process$

Initialize: $X\_fetch = \{\emptyset\}$, $X\_process = \{\emptyset\}$
for all S_i in S **do**
  for all $T_{i\#j}$ in S_i **do**
    Calculate the task data fetching time $FT_{i\#j} \leftarrow eq.(4) and eq.(9)$
    $X\_fetch.add\,(FT_{i\#j})$
    Calculate the task data processing time $PT_{i\#j} \leftarrow eq.(10)$
    $X\_process.add\,(PT_{i\#j})$
  end for
end for
return $FT_{i\#j}, PT_{i\#j}$

In Spark's default task scheduling mechanism, when the number of partitions exceeds the number of tasks running concurrently, the vCPU tasks are executed simultaneously. Only when the current task completes execution and there are available CPU cores, can the next task commence. Task execution demands various resources including CPU and network, and this demand is dynamic. Hence, task execution is divided into data fetching and data processing phases. Running other tasks at suitable times during task execution can mitigate resource contention and enhance resource utilization.

4.2. Task Scheduling Algorithm Based on Maximizing Resource Interleaving

According to the detailed information of the corresponding tasks in each node, the tasks in each node are first grouped according to different stages and sorted according to the execution time; then, since the stage with the longest execution time affects the completion time of the whole pipeline, it is prioritized and staggered according to the length of the completion time of the tasks in the historical data; finally, for other parallel stages, due to the existence of deadline, it is only necessary to ensure the completion of its execution before the completion of the longest stage, so its tasks can be scheduled at the right time to stagger the utilization of cluster resources.

The data fetching time $FT_{i\#j}$, data processing time $PT_{i\#j}$ of $Task_{i\#j}$ is obtained from Algorithm 1. During task execution, a large amount of network resources are needed in the data fetching phase, while a large amount of memory is needed for computation in the data processing phase. The goal of DRTS is to mitigate resource contention when executing tasks in the CPU-intensive phase, during which the cluster requires substantial network resources and disk I/O. In accordance with Spark's default task scheduling strategy, multiple tasks will be executed in parallel, and multiple tasks simultaneously multiple tasks compete for resources at the same time. At this point, Algorithm 2 schedules parallel tasks at optimal times, leveraging staggered time intervals to utilize cluster resources efficiently and reduce resource contention.

To achieve the objective of minimizing the job execution time, the following algorithm is designed. The algorithm can be divided into the following steps:

Algorithm 2 Delay-Aware Resource-Efficient Interleaved Task Scheduling Strategy

Input: Time maps X_{fetch}, X_{process} and the initial set of parallel stages S, the downstream shuffle S_i of S_k.

Output: Path and delay time set X.

Initialize: $X \leftarrow \{\}$ and the set of execution path P according to the job's DAG.
Sort the parallel S in descending order
for all S_i in S **do**
    Sort $T_{i\#j}$ in S_i in descending order, and group by target_node
    X.put $(i,$ new List$())$
end for
for all S_i in S **do**
    for all $different\_node\_task$ in S_i **do**
        for all $T_{i\#j}$ in all $different\_node$ **do**
            if isLongestStage(node.StageId) **then**
                Target machines execute tasks in an interleaved manner of long and short tasks firstly
            else
                Base_Time $= \left(S_{S_i} - S_{S_{i+1}}\right)/2$
                if $S_k \notin \varphi_m$ **then**
                    Base_Time $= (S_{S_k} - \max(S_{\text{parent}_{S_k}}))/2$
                    $S_i = S_i +$ Base_Time
                    for all $\hat{x_k} \in [0, T_{i\#j} - T_{i\#(j+1)}]$ **do**
                        if $PT_{i\#j} < T_{i\#(j+1)}$ **then**
                            $\hat{x_k} \leftarrow T_{i\#j} - T_{i\#(j+1)}$
                            $\hat{x_k} \leftarrow \hat{x_k} - \triangle x$
                        else
                            $\hat{x_k} \leftarrow FT_{i\#j}$
                            $\hat{x_k} \leftarrow \hat{x_k} \pm \triangle x$
                        end if
                        $x_k \leftarrow \hat{x_k} +$ Base_Time
                        X.get(i).add$(T_{i\#j}, x_k)$
                    end for
                end if
            end if
        end for
    end for
end for

(1) According to the parallelism of stage, it is partitioned according to different paths, and the partitioned paths are φ_m. Based on the allocation of parallelizable stage in each node, traversal is conducted to obtain the set of tasks in the node belonging to different partitioned paths.

(2) When the downstream shuffle of $S_i \notin \varphi_m$, the scheduling time of S_i is adjusted accordingly. This adjustment involves finding the downstream shuffle of S_i, denoted as S_k, by analyzing the DAG in the log file. Consequently, the delayed scheduling time interval of S_i is determined to be [0, S_{S_k} -max($S_{parent_{S_k}}$)]. where S_{S_k} refers to the start time of S_k, and max($S_{parent_{S_k}}$) refers to the maximal completion time in the parent stage of S_k as shown in Figure 3,and Stage5 belongs to this situation.

(3) Since the stage with the longest execution time significantly impacts the completion time of the entire job, it is prioritized for scheduling. Therefore, it is necessary to first sort φ_m in descending order.

(4) $Task_{i\#j}$ are grouped according to i in each target node according to the above results. They are then arranged in both descending and ascending time orders to enable interleaved execution.

(5) For a stage, if it is the longest among several parallel stages, each node prioritizes the execution of tasks assigned to that stage. Tasks of that stage are then executed according to the alternation of long and short time tasks.

(6) For the other stages that can be parallelized, since their overall execution time is not long, the start time of the execution of the back-ordered stages has little to do with the completion time of those stages. The adjusted completion time must not exceed the completion time of the first stage after sequencing. Therefore, the task execution of subsequent stages is initially delayed by a certain period of time, with the delay time initially set as the middle value of $S_i - S_{i+1}$.

(7) The execution of tasks corresponding to each stage on each node is performed in the order in (3), and the execution of tasks is performed in the order of the results in (4), and the execution of tasks that are at the back of the ordering is postponed at an appropriate time. But the delay time of the subsequent task must be within [0, $S_i - S_{i+1}$-$T_{i\#j}$]. The reason is that when the Task is delayed for too long, it may lead to a marginal overall completion time of the task, which is contrary to the DRTS policy of minimizing the job completion time.

(8) Ensure that after the execution of the task with the longest stage, the data fetching time $FT_{i\#j}$ in the other tasks is used as a cumulative count of the initial value. This ensures that the delay time of each task is different, and constant feedback is provided to adjust this value.

(9) Divide the delay time of the task into blocks of time Δx (e.g., each block has an interval of 200 ms). When $PT_{i\#j} < T_{i\#(j+1)}$, $\hat{x_k}$ iterates over its upper and lower ranges [0, $T_{i\#j} - T_{i\#(j+1)}$]; When $PT_{i\#j} >= T_{i\#(j+1)}$,$\hat{x_k}$ iterates over its upper and lower ranges [$FT_{i\#j}$, $T_{i\#j} - T_{i\#(j+1)}$]. There is a candidate task scheduling time $\hat{x_k}$ in each iteration . Where $T_{i\#j}$ is the previous task of $T_{i\#j+1}$ at the same target machine.

(10) After continuous feedback and adjustments based on the results of the last execution, DRTS finally determines the optimal value of the delayed task execution time x_k to greedily minimize the execution time of parallel tasks.

DRTS schedules parallel tasks in a pipelined manner, which effectively reduces resource interleaving during task execution and improves the resource utilization of the cluster.

5. Experiments

In this section, a comprehensive evaluation of the DRTS strategy is presented. Three workloads ConnectedComponents, CosineSimilarity and TriangleCount are used as benchmarks to evaluate the performance. The evaluation results are as follows:

(1) Accelerated the workload of the benchmark suite, reducing the average job completion time by 7.51% to 15.64%.

(2) By utilizing cluster resources in a staggered manner, the CPU utilization and network utilization of the cluster are improved by 5.83% and 10.38%, respectively.

5.1. Experiment Setting

Cluster Configurations In this section, to validate the performance of DRTS, we conduct extensive experiments on a Spark 2.4.0-based cluster with one master node and six worker nodes. We evaluated the performance of task scheduling using three representative DAG-style data analysis workloads as benchmarks. The cluster configuration is shown in Tab.1. The deployment method is Spark on Standalone, and Spark's parameters are configured as shown in Tab. 2.

Table 1. Configuration of Cluster

Type	Configuration
Number of Nodes	1 master,6 workers
CPU Number	4
RAM	32G
Hard disk	500G
Environment	Centos7.0, Spark2.4.0, Hadoop 2.6.0
Digital meter	2500W, 10A
JDK	JDK 1.8

Table 2. Configuration of Spark parameters

Type	Configuration
Executor cores	2
Executor memory	4G
Executor number	12

Workload Detail Description In the cluster, we chose three representative Spark benchmark workloads: ConnectedComponents, CosineSimilarity, and TriangleCount, where ConnectedComponents and TriangleCount are from Spark GraphX. There are 5 stages and 11 stages, respectively. CosineSimilarity comes from Spark MLlib and has five stages. Specifically, the experimental ConnectedComponents application has 11GB of synthetic data, the CosineSimilarity application uses 34GB of synthetic data, and TriangleCount uses synthetic data from 1 million users and 20 million followers. The workload specifications are summarized in Tab. 3.

Base Line The scheduling algorithms we compared in these experiments are as follows:
Spark: the default scheduling strategy FIFO in Spark.

Table 3. Workload

Wordload	Specification
ConnectedComponents	11GB synthetic input data
CosineSimilarity	34GB synthetic input data
TriangleCount	1 million users and 20 million followers of synthetic input data

DelayStage[16]: it is a stage delay scheduler for big data parallel computing frameworks (e.g. Spark). It optimizes resource utilization by overlapping cluster resources during parallel phases to minimize completion time. Additionally, it schedules phase execution in a pipelined manner to maximize the performance benefits of resource interleaving.

5.2. Performance Results

In order to validate the performance of DRTS, a series of experiments were conducted on different datasets using the benchmarks in Tab .3. Each benchmark was tested 15 times and the final results were averaged over the tests. This experiment compares the DRTS scheduling strategy with Spark's default scheduling strategies FIFO and DelayStage scheduling strategies.

Job Execution Time We first compare the job execution time of different schedulers. As shown in Fig. 5, we can see that DRTS shortens the job execution time by 8.01% to 14.46% compared to Spark's default scheduling strategy. DRTS is a delayed scheduler operating at the task level across different nodes. When tasks are executed concurrently, it can lead to resource contention issues. By scheduling parallel tasks at the right time, DRTS effectively interleaves the use of cluster resources, thereby enhancing overall resource utilization. Moreover, the task execution is divided into two phases, when the data is being acquired, scheduling a CPU-intensive task at the right time to greedily realize the resource interleaving between tasks can also reduce the total execution time of parallel tasks.

Additionally, the performance of DRTS is also enhanced (3.18% - 6.48%) compared with DelayStage due to the following reasons: DRTS schedules parallel tasks assigned to each node at optimal times, allowing tasks in the parallel stage to be delayed based on the state of the target machine. On the other hand, DelayStage only delays the overall stage without considering that tasks within the stage are assigned to different machines, resulting in uniform delays across all tasks. DRTS considers more factors and offers finer granularity compared to DelayStage.

Upon closer examination of the job completion times depicted in Fig. 5, DRTS demonstrates superior performance compared to Spark's default FIFO scheduling approach. The figure illustrates the job completion times for the three benchmarks: ConnectedComponents, CosineSimilarity, and TriangleCount. Specifically, DRTS improves performance by 8.01% for ConnectedComponents, reduces job completion time by 13.51% for CosineSimilarity, and enhances performance by 14.46% for TriangleCount. This improvement can be attributed to the nature of FIFO scheduling, which prioritizes earlier submitted tasks for execution, causing subsequent tasks to wait until prior ones are completed. When early

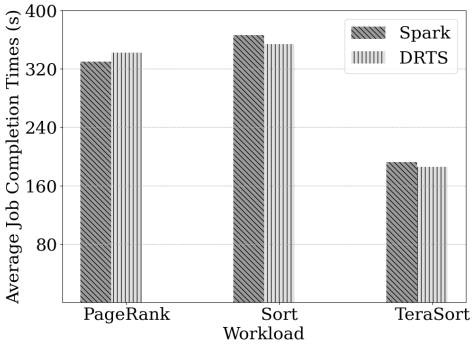

Fig. 5. Comparison of Spark and DRTS across different workloads, including PageRank, Sort, and TeraSort

tasks do not utilize resources efficiently, it may result in overall low resource utilization. By comparing these three benchmarks with DelayStage, DRTS demonstrates performance improvements of 3.18%, 6.05%, and 6.48%, respectively. This is because DelayStage only delays the stage without considering granularity such as different target machines or tasks on different target machines. In contrast, DRTS operates at the task level, sorting tasks within stages with longer execution times based on target machine execution times. It prioritizes stages with the longest execution times for scheduling while ensuring that execution scheduling delays are applied to different target machines for different tasks based on varied execution conditions. This approach alternately utilizes cluster resources to avoid resource contention, ultimately achieving the goal of reducing job execution time.

In the Spark framework, parallelism refers to the number of data blocks processed simultaneously during task execution, and it's a manually configured parameter. Excessive parallelism may over-consume cluster resources or result in frequent task startups, increasing overhead. Conversely, insufficient parallelism may underutilize cluster resources, prolonging task execution. Configuring parallelism appropriately poses challenges for developers, as improper configurations can increase job completion time, impacting cluster performance.

DRTS can mitigate performance degradation resulting from suboptimal parallelism configurations. To observe the impact of different parallelism levels on DRTS, we conducted a series of experiments. The results, depicted in Fig. 6, demonstrate that DRTS effectively reduces job completion time across varying degrees of parallelism.

Resource Utilization Effectiveness To assess whether our DRTS policy enhances resource utilization, we conducted further observations on CPU utilization and network throughput of a worker node while executing various workloads. As depicted in Fig. 7. and 8, the DRTS policy optimizes DelayStage by efficiently utilizing idle time during stage execution, thereby notably enhancing CPU utilization and network throughput. Compared to Spark's default scheduler, DRTS executes tasks in a two-stage resource interleaving manner, effectively improving system resource utilization. On average, CPU utilization improves by 4.43% to 8.77%, and network throughput utilization improves

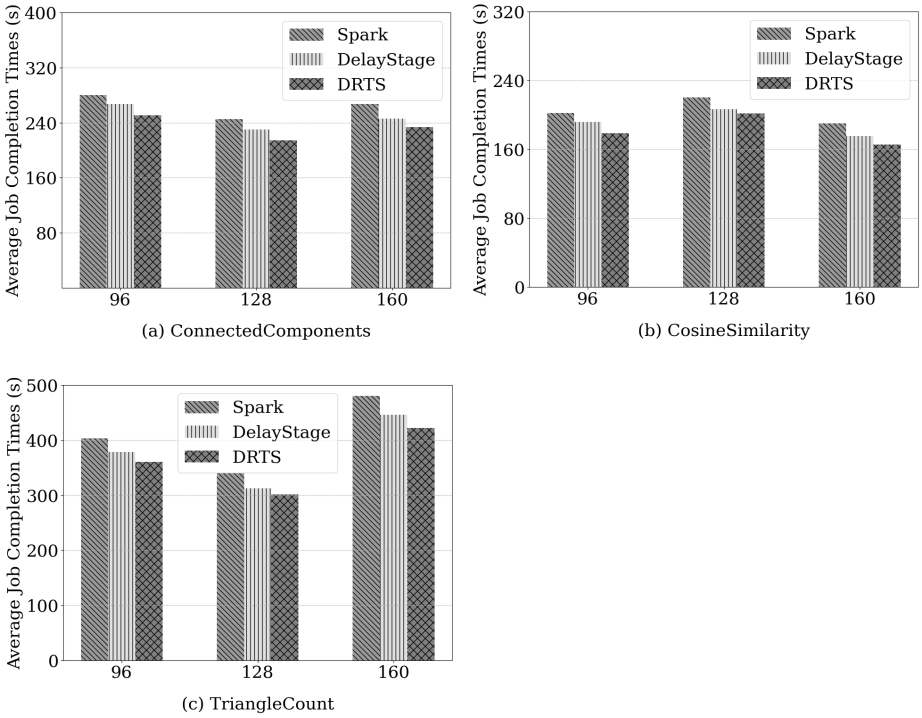

Fig. 6. Comparison of Spark, DelayStage, and DPRS at different levels of parallelism

by 5.19% to 8.84% with DRTS. Additionally, DRTS enhances resource utilization compared to DelayStage, with CPU utilization improving by 8.42% to 14.64% and network throughput improving by 10.43% to 12.98%. This improvement stems from DRTS's finer granularity in scheduling tasks to the same target machine at optimal times, whereas DelayStage simply delays stage scheduling, potentially resulting in idle resources on some machines. By scheduling CPU-intensive tasks at opportune moments during data acquisition, DRTS effectively fills resource gaps during CPU idle periods, thereby enhancing job operational efficiency through optimal resource utilization.

Table 4. Average of network throughput (MB/s)

	Spark	DelayStage	DRTS
ConnectedComponents	18.88	19.86	20.85
CosineSimilarity	136.21	148.25	152.14
TriangleCount	23.66	25.32	26.73

Further, we compute the average values of network throughput and CPU utilization of a work node while executing these four workloads, as summarized in Tab. 4. and 5. respectively.

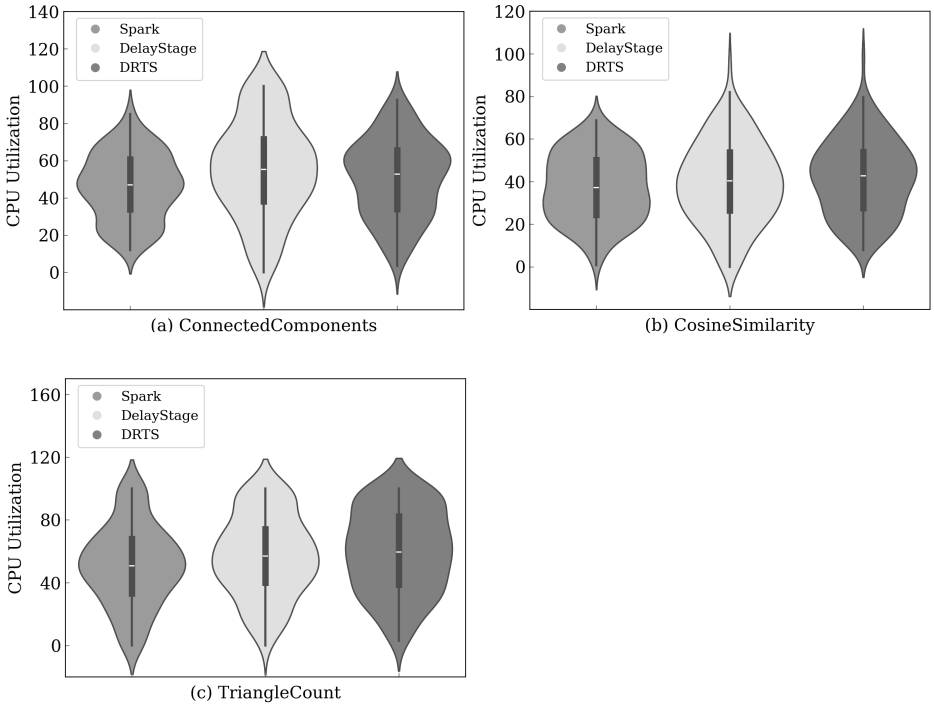

Fig. 7. CPU utilization under the three workloads: ConnectedComponents, CosineSimilarity, and TriangleCount

Clearly, DRTS achieves higher and more stable CPU utilization and network throughput compared to Spark's default scheduling strategy. In more detail, DRTS improves network utilization by 11.42% and network throughput by 11.7% over Spark, and also improves CPU utilization and network throughput by 6.33% and 7.02%, respectively, compared to DelayStage.

6. Conclusion

In this paper, we address the task scheduling problem within job execution scenarios. Underutilized assigned tasks can significantly prolong job execution times, thereby impacting cluster performance. To mitigate this issue, we propose a strategy focusing on

Table 5. Average CPU utilization (%)

	Spark	DelayStage	DRTS
ConnectedComponents	46.22	48.89	50.11
CosineSimilarity	37.51	39.17	41.71
TriangleCount	50.88	55.34	58.33

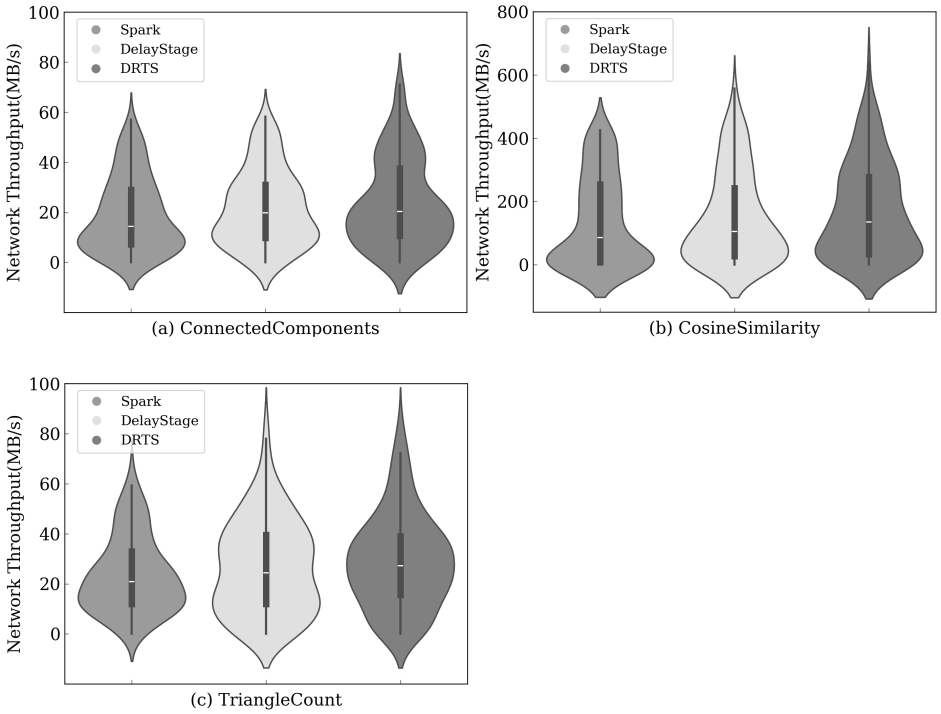

Fig. 8. Network throughput under the three workloads: ConnectedComponents, CosineSimilarity, and TriangleCount

cross-utilization of resources to minimize job completion times. Firstly, we provide a comprehensive theoretical analysis of the task scheduling problem and devise an algorithm to compute task execution times for both data acquisition and processing phases. Subsequently, we introduce a task scheduling algorithm based on resource polling, employing a delayed scheduling approach at the task level across different nodes. For stages with extended execution times, tasks are scheduled on target machines in an alternating pattern of long and short task durations. For other stages, appropriate scheduling algorithms are employed to ensure tasks are executed at optimal times, thereby facilitating cross-resource utilization within the cluster. Finally, we conduct extensive experiments across three benchmarks using various datasets. Our experimental results demonstrate that the proposed DRTS approach effectively harnesses cluster resources and reduces job completion times.

Acknowledgments. This work was supported by a National Natural Science Foundation of China, Major Research Program, 92367302, Integrated System and Validation of Industrial Internet Based on the New Architecture of AllFactor On-Demand Collaborative Interconnection.

References

1. Apache flink — stateful computations over data streams. https://flink.apache.org.

2. Apache spark - unified engine for large-scale data analytics. https://spark.apache.org.
3. Apache storm is a free and open source distributed realtime computation system. https://storm.apache.org.
4. Dhawalia p, kailasam s, janakiram d. chisel: A resource savvy approach for handling skew in mapreduce applications[c]//2013 ieee sixth international conference on cloud computing. ieee, 2013: 652-660.
5. Duan y, wang n, wu j. accelerating dag-style job execution via optimizing resource pipeline scheduling[j]. journal of computer science and technology, 2022, 37(4): 852-868.
6. Fu z, tang z, yang l, et al. an optimal locality-aware task scheduling algorithm based on bipartite graph modelling for spark applications[j]. ieee transactions on parallel and distributed systems, 2020, 31(10): 2406-2420.
7. Gu r, tang y, tian c, et al. improving execution concurrency of large-scale matrix multiplication on distributed data-parallel platforms[j]. ieee transactions on parallel and distributed systems, 2017, 28(9): 2539-2552.
8. "hadoop," 2021. [online]. https://hadoop.apache.org.
9. He x, shenoy p. firebird: Network-aware task scheduling for spark using sdns[c]//2016 25th international conference on computer communication and networks (icccn). ieee, 2016: 1-10.
10. Hu z, li b, qin z, et al. job scheduling without prior information in big data processing systems[c]//2017 ieee 37th international conference on distributed computing systems (icdcs). ieee, 2017: 572-582.
11. Hu z, li d. improved heuristic job scheduling method to enhance throughput for big data analytics[j]. tsinghua science and technology, 2021, 27(2): 344-357.
12. Jiang j, ma s, li b, et al. symbiosis: Network-aware task scheduling in data-parallel frameworks[c]//ieee infocom 2016-the 35th annual ieee international conference on computer communications. ieee, 2016: 1-9.
13. Li x, ren f, yang b. modeling and analyzing the performance of high-speed packet i/o[j]. tsinghua science and technology, 2021, 26(4): 426-439.
14. Lu s x, zhao m, li c, et al. time-aware data partition optimization and heterogeneous task scheduling strategies in spark clusters[j]. the computer journal, 2023: bxad017.
15. Pan f, xiong j, shen y, et al. h-scheduler: Storage-aware task scheduling for heterogeneous-storage spark clusters[c]//2018 ieee 24th international conference on parallel and distributed systems (icpads). ieee, 2018: 1-9.
16. Shao w, xu f, chen l, et al. stage delay scheduling: Speeding up dag-style data analytics jobs with resource interleaving[c]//proceedings of the 48th international conference on parallel processing. 2019: 1-11.
17. Tang z, xiao z, yang l, et al. a network load perception based task scheduler for parallel distributed data processing systems[j]. ieee transactions on cloud computing, 2021, 11(2): 1352-1364.
18. Tang z, zeng a, zhang x, et al. dynamic memory-aware scheduling in spark computing environment[j]. journal of parallel and distributed computing, 2020, 141: 10-22.
19. Wang j, gu h, yu j, et al. research on virtual machine consolidation strategy based on combined prediction and energy-aware in cloud computing platform[j]. journal of cloud computing, 2022, 11(1): 50.
20. Wang j, yu j, song y, et al. an efficient energy-aware and service quality improvement strategy applied in cloud computing[j]. cluster computing, 2023, 26(6): 4031-4049.
21. Wang j, yu j, zhai r, et al. gmpr: a two-phase heuristic algorithm for virtual machine placement in large-scale cloud data centers[j]. ieee systems journal, 2022, 17(1): 1419-1430.
22. Xu g, xu c z, jiang s. prophet: Scheduling executors with time-varying resource demands on data-parallel computation frameworks[c]//2016 ieee international conference on autonomic computing (icac). ieee, 2016: 45-54.

23. Xu y, liu l, ding z. dag-aware joint task scheduling and cache management in spark clusters[c]//2020 ieee international parallel and distributed processing symposium (ipdps). ieee, 2020: 378-387.
24. Zaharia m, borthakur d, sen sarma j, et al. delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[c]//proceedings of the 5th european conference on computer systems. 2010: 265-278.
25. Zhang x, li z, liu g, et al. a spark scheduling strategy for heterogeneous cluster[j]. computers, materials continua, 2018, 55(3).
26. Dean J, G.S.: Mapreduce: simplified data processing on large clusters. In: Proceedings of the 1st International Conference on Preparing ComSIS Articles. pp. 107–113. Communications of the ACM (2008)
27. Islam M T, Karunasekera S, B.R.: Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. In: IEEE Transactions on Parallel and Distributed Systems. pp. 107–113. Communications of the ACM (2021)

Yanhao Zhang is currently studying for a master's degree. In 2018, he received a bachelor's degree in Software Engineering from Shangqiu Normal University. Her research interests include parallel and distributed computing and Spark.

Congyang Wang is currently studying for his master's degree. He received his Bachelor of Engineering degree from Henan University in 2018. His research interests include big data and parallel computing.

Xin He is a professor at the School of Software at Henan University. He received his Master of Science degree in Applied Mathematics from Henan University in 2005. In 2011, he received his Ph.D. degree in Computer System Structure from Xi 'an Jiaotong University. His research interests include mobile computing, cloud computing and big data processing, and computer networking.

Junyang Yu is an associate professor at the School of Software at Henan University. He received his master's degree from the School of Computer and Information Engineering, Henan University in 2007. In 2016, he received his PhD in Software Engineering from Central South University. His research interests include cloud computing, big data, and distributed systems.

Rui Zhai received his Ph.D. degree in computer science from University of Chinese Academy of Sciences in 2015. Currently, he is a lecturer at the Software School of Henan University. His research interests include machine learning, federated learning, and GNN.

Yalin Song is an associate professor at the School of Software, Henan University, and holds a Ph.D. degree from Tongji University. His research interests include cognitive computing and computer vision.

Received: August 31, 2024; Accepted: January 20, 2025.