Journal of Grid Computing (2023) 21:33
https://doi.org/10.1007/s10723-023-09661-2

RESEARCH

Cost-Aware Scheduling and Data Skew Alleviation for Big Data Processing in Heterogeneous Cloud Environment

Hongjian Li · Lisha Zhu · Shuaicheng Wang · Lei Wang

Received: 21 August 2022 / Accepted: 7 April 2023 / Published online: 22 June 2023

Abstract For big data applications, it is important to allocate resources reasonably and schedule tasks effectively. As one of the popular big data processing frameworks, the default scheduling strategy of Spark still suffers from low resource utilization and high resource cost. In this paper, a low-cost task scheduling algorithm for Spark based on heterogeneous cloud environment is proposed to minimize cost while improving resource utilization. First of all, we construct a cost model for Spark based on the hierarchical relationship among applications, jobs, stages, and tasks. Then, based on the model, a low-cost task scheduling algorithm is proposed, which improves the utilization of computational resources by adjusting the task parallelism and achieves task scheduling with priority based on the distribution of data to be computed. We also propose a Reduce task load balancing partitioning algorithm (RTLBPA) based on prior information of data by dispersing and aggregating different keys with the same hash value in the data into different partitions. This algorithm can achieve load balancing and improve the execution time of the job by reducing the whole Reduce phase. Finally, we performed extensive experiments on the proposed algorithm using Hibench's workloads in the cloud environment. The result shows that the cost can be reduced by at least 22.71% compared with the existing algorithm under different workloads. As a result, the proposed algorithm can improve the cost efficiency of Spark cluster effectively while improving resource utilization.

H. Li (✉) · L. Zhu · S. Wang · L. Wang
Department of Computer Science and Technology, Chongqing University of Posts and Telecommunications, Chongqing 400065, China
e-mail: lihj@cqupt.edu.cn

Keywords Spark · Heterogeneous cloud · Task scheduling · Cost efficiency

1 Introduction

With the emergence and development of cloud computing [1–3], more and more enterprises and organizations choose to migrate their business systems from the local to the cloud. In this way, it is no longer necessary to build and maintain a huge computing infrastructure. Instead, enterprises can obtain computing resources flexibly according to their business requirements, which largely reduces their cost overhead. Currently, cloud service providers offer a variety of rental models to meet the computing requirements of different users, such as annual and monthly subscriptions and pay-per-volume and so on. To meet different business scenarios as much as possible, cloud service providers offer several types of cloud server instances, including general-purpose instances, compute instances, in-memory instances, etc. At the same time, to meet user-defined requirements and achieve the purpose of elastic and scalable computing resources, the resource configurations (CPU, memory, storage capacity, and network bandwidth, etc.) of all instances can be configured by users themselves and eventually paid for on a per-volume basis. Based on these features, the installation and deployment of big data processing frameworks in the cloud are also becoming prevalent.

To demonstrate the effectiveness of the proposed algorithm, we have chosen Apache Spark [4,5], as the target framework because of its fast memory-based processing speed and efficient iterative computation. Compared with the MapReduce [6–8] computing framework, the memory-based computing speed of Spark is more than one hundred times faster than the former, and its hard disk-based computing speed is more than ten times faster than the former. However, as the default scheduling mechanism of Spark, the FIFO [9,10] (First In First Out) s and Fair [11–13] strategies assign tasks to all available computing nodes by distributed polling during task scheduling, which leads to low resource utilization and high cost. In order to resolve the above problems, many scholars have conducted research on the task scheduling mechanism of Spark. For example, for the scheduling scenario of computing requests in hybrid clouds (consisting of public and private clouds), Wang et al.[14] transform the initial online optimization problem into a one-time binary linear optimization problem and proposes a hybrid cloud scheduler based on this, which can minimize the public cloud cost while meeting the job deadline. In addition, it can also improve resource utilization effectively and reduce scheduling latency. Wang et al. [15] propose a new hard real-time scheduling algorithm, DVDA (Deadline and Value Density-Aware, DVDA), to address the problem that none of the three native scheduling algorithms can satisfy the requirements of hard real-time scheduling in YARN mode. The proposed algorithm takes into account the deadline constraints and the value density of the application to maximize the value while meeting the deadline constraints. Hussain et al. [16] propose a cost-effective SLA-based load balancing scheduler to address the problems of poor resource utilization and high execution cost caused by inappropriate job assignment and dispersed scheduling of existing SLA-based heuristic algorithms, which can improve resource utilization while reducing execution time and execution cost.

However, they do not take into account such as the heterogeneity of the cluster and the distribution of the data to be computed for each stage. If the task and the data to be computed are in different nodes, the data will likely be transferred over the network, which will bring significant performance overhead; in addition, assigning tasks and taking up resources without considering whether the resources are fully utilized will lead to both waste of resources and increase the cost of resource usage during job execution. Therefore, it becomes particularly important and meaningful to minimize costs while improving utilization.

In this paper, we propose a cost-aware task scheduling strategy for Spark based on a heterogeneous cloud environment, which can reduce the cost of the cluster while improving resource utilization. The main idea of this algorithm is to adjust task parallelism and achieve the task scheduling with priority based on the distribution of data to be computed at different stages, which is aimed to minimize the cost of the cluster while improving the utilization of computational resources. RTLBPA, a reduced task load balancing partitioning algorithm based on prior information of data, is proposed in this paper. The main idea of this algorithm is to disperse and aggregate different keys with the same hash value in data into different partitions, to improve the situation of excessive load and long execution time of a few tasks caused by data skewing, thus achieving load balancing of tasks and improving the overall execution efficiency of jobs. Compared with the baseline algorithm, the proposed algorithm can reduce the cost of the cluster while improving CPU utilization.

The main contributions of this paper are as follows:

- A cost model for Spark was constructed based on a hierarchical relationship among applications, jobs, stages, and tasks.
- Based on the above model, a low-cost task scheduling algorithm for Spark based on a heterogeneous cloud environment is proposed. The core idea of the algorithm is to adjust the parallelism of tasks and to achieve priority scheduling of tasks based on the distribution of data to be computed at different stages. The algorithm can improve resource utilization while minimizing the resource cost of the cluster.
- A reduced task load balancing partitioning algorithm (RTLBPA) is proposed based on prior information of data. The algorithm can effectively reduce the execution time of the Reduce phase and improve the overall execution efficiency of the job through task load balancing.
- Based on the workloads of Hibench, we have performed extensive experiments. Moreover, compared with Spark's default scheduling algorithm

and the state-of-the-art algorithm, the proposed algorithm in this paper can reduce the resource cost of the cluster while improving CPU resource utilization.

The rest of the paper is organized as follows. Section 2 describes the related work. Sections 3 and 4 introduce the proposed model and algorithm. In Section 5, we evaluate the performance of the proposed algorithm and analyze the experimental results. In Section 6, we conclude the research of this paper and look forward to future research directions.

2 Related Work

It is crucial for Spark applications to allocate resources reasonably and schedule tasks effectively. Therefore, many scholars have already studied Spark from different perspectives, as shown in Table 1.

Therefore, many scholars have already studied Starfire from different perspectives, as shown in Table 1.

There has been a significant amount of research on the energy consumption of clusters. Maroulis et al. [9] proposed a novel framework that orchestrates the execution order of Spark applications, exploiting DVFS to tune the CPU frequency of compute nodes to minimize the energy consumption and satisfy the performance requirements of the application. Wang et al. [17] proposed a data caching framework for Spark in a hybrid DRAM/NVM memory configuration. By identifying the data access behaviors with active factor and active stage distance, cache data with higher local I/O activity is preferentially cached in DRAM, while cache data with lower activity is placed into NVM. The data migration strategy dynamically moves the cold data from DRAM into NVM to save static energy consumption. The result shows that the proposed framework can effectively reduce energy consumption and improve latency performance. Li et al. [18] designed a new energy consumption model and proposed an energy-aware task scheduling algorithm, EASAS (Energy-Aware Scheduling Algorithm for Spark), to address the shortcoming that the default scheduling algorithm of Spark is not energy-aware. The proposed algorithm optimizes task scheduling based on a history strategy table that records the execution time and energy consumption of historical tasks. It aims to allocate tasks to the optimal executor, and to reduce the energy consumption of the platform while meeting service level agreements. Li et al. [19] proposed a frequency-aware and energy-saving strategy based on dynamic voltage and frequency scaling (abbreviated as FAESS-DVFS) to reduce the energy consumption for big data processing in Spark on YARN. Experimental results show that the proposed strategy can reduce the energy

Table 1 Comparison of related works

Works	Energy consumption	Execution time	Through-put	Latency	Cost-efficient	Resource utilization	Deadline constraints
[6]	✓	×	×	×	×	×	✓
[10]	✓	×	×	✓	×	×	×
[11]	✓	×	×	×	×	×	✓
[12]	✓	×	×	×	×	×	✓
[13]	×	✓	×	✓	×	×	×
[14]	×	×	✓	✓	×	×	×
[15]	×	✓	×	×	×	✓	×
[16]	×	×	×	✓	×	×	✓
[17]	×	×	×	×	✓	✓	×
[18]	×	×	×	×	✓	×	✓
[19]	×	×	×	×	✓	×	✓
[20]	×	✓	×	×	✓	×	×
[21]	×	✓	×	✓	✓	×	✓
[22]	×	✓	×	×	✓	×	✓

consumption of big data processing significantly while satisfying SLA constraints.

As a big data processing platform, the system performance should not be neglected, and many scholars have taken this as the starting point to study Spark. For example, Hu et al. [20] proposed a new scheduler that uses a multi-level priority queue to simulate the shortest job first strategy, which can improve the overall performance of the job without knowing the scale of the job. Cheng et al. [23] proposed an adaptive parallel job scheduler based on Spark Streaming for the complex data dependencies and dynamics inherent in streaming workloads, which can schedule multiple concurrent jobs dynamically according to the data dependencies, and adjust the parallelism and resource sharing among jobs adaptively. At the same time, they combine dynamic batch processing techniques with a scheduler to resize batches dynamically in response to changing streaming workloads and processing speed. The experiments have shown that the scheduler can not only reduce the end-to-end latency but also increase the throughput of workloads. At present, most task scheduling mechanisms of Spark only consider the parallelism of tasks but ignore the usage of memory, which results in unreasonable resource utilization and inefficient execution. Based on this, Tang et al. [24] proposed a dynamic memory-aware task scheduler (DMATS). They regard CPU, memory, and network IO as computing resources, and adjust the concurrency of task execution dynamically through specific feedback information during task scheduling, so as to improve resource utilization and reduce application execution time. Neciu et al. [25] proposed the earliest deadline first (EDF) real-time scheduling algorithm for the problem that the default scheduling algorithm of Spark does not support real-time scheduling with deadline constraints. The proposed algorithm completes real-time scheduling based on the deadlines of Spark jobs and reduces the job latency effectively with the given constraints.

In addition, some studies focus on cost savings while satisfying conditional constraints. Islam et al. [21] proposed two dynamic scheduling algorithms to address the problem that the default scheduling algorithm of Spark cannot reduce the cost of virtual machines while satisfying constraints of job performance in the cloud environment. These two algorithms take into account the impact of differences in VM types and job features on job performance and cost, to reduce the cost of the Spark cluster in the cloud while improving resource utilization. Chen et al. [26] proposed two low-complexity algorithms, DCO and DUCO, for the cost optimization of parallel applications with deadline constraints in the heterogeneous cloud environment. The former refines the deadline constraint of the application downward and transforms it into the deadline constraint of each task during task scheduling, while the latter uses the relaxation time to optimize upward based on the former so that the execution cost of the job can be minimized while meeting the deadline constraint of the application. To resolve the problem of limited resources by only using local resources and the problem of high cost by only using cloud resources, Islam et al. [22] designed a deployment model which combines local resources and cloud resources and proposed an efficient scheduling algorithm based on it. The proposed algorithm utilizes different VM pricing models to optimize costs for both local and cloud. The results show that the algorithm can not only reduce the usage cost of virtual machines significantly in the hybrid cloud environment but also maximize the percentage of job deadline satisfaction. To address the problems that existing scheduling algorithms in the cloud environment cannot handle multiple SLA targets simultaneously and capture the inherent characteristics of workloads, Islam et al. [27] introduced a new reinforcement learning model and proposed two schedulers based on deep reinforcement learning. The proposed schedulers can not only place Spark executors by using the pricing model of VM instances in the cloud but also find the most appropriate placement based on the inherent characteristics of different jobs. In this way, the execution time of jobs and the usage cost of virtual machines can be reduced.

The problems with most cluster schedulers are that they do not take into account the heterogeneity of the cluster and the distribution of the data to be computed at different stages. All of them only select the total number of resources or nodes needed for each job while making any scheduling decision. In this way, if the task and the data to be computed are in different nodes, it is highly likely that the data will be transferred over the network, which will bring great performance overhead; in addition, assigning tasks and taking up resources without considering whether the resources are fully utilized will lead to both waste of resources and increase the cost of resource usage during job execution, and that is what this paper considers. Therefore, in this paper, we propose a low-cost task scheduling algorithm for Spark based on a heterogeneous cloud environment, which is aimed to minimize the cost while improving CPU resource utilization.

3 Cost Model for Spark

In this section, a two-dimensional cost model is constructed for the Spark cluster, which is based on abstracting the hierarchical relationship among applications, jobs, stages, and tasks.

The related symbols and descriptions of the cost model for Spark are shown in Table 2.

In the Spark framework, the hierarchical relationship among applications, jobs, stages, and tasks can be abstracted as (1-3). As shown in (4), Exe is the set of executor process in spark cluster. The expressions are shown below:

$$App = \{Job_0, Job_1, Job_2, \dots, Job_m\} \tag{1}$$
$$Job_m = \{Stage_{m0}, Stage_{m1}, Stage_{m2}, \dots, Stage_{mn}\} \tag{2}$$
$$Stage_{mn} = \{Task^0_{mn}, Task^1_{mn}, Task^2_{mn}, \dots, Task^l_{mn}\} \tag{3}$$
$$Exe = \{Ex_0, Ex_1, Ex_2, \cdots, Ex_z\} \tag{4}$$

In (1), since the App generates a job for each Action operator it executes, and the number of Action operators in the App is not fixed, an App may consist of multiple jobs. In (2), since each job can be divided into multiple stages after Shuffle [28] operation, and there is a certain dependency between them, the actual execution process should be executed in order according to the dependencies. In (3), the smallest unit of computing is a task in the Spark computation framework, so each stage may contain multiple tasks. However, the task will eventually be distributed by the driver to the executor on the compute node for processing. By default, an executor is allocated by a node and this executor has all CPU cores of the current node.

Table 2 Definition of symbols

Symbol	Definition
App	User-submitted applications
Job_m	The m^{th} job in the application
$Stage_{mn}$	The n^{th} stage in the m^{th} job
$Task^l_{mn}$	The l^{th} task in the n^{th} stage of the m^{th} job
RC	Resource usage cost of Spark cluster
η	The set of all VM types in the Spark cluster
δ_k	The set of all VMs of type k
f_k	Fixed cost to use a VM of type k
Exe	The set of executor process in spark cluster
Ex_q	The m^{th} executor in Exe
$RC_{Task^l_{mn}}$	The resource cost of Ex_q when executing $Task^l_{mn}$
$t_{Task^l_{mn}}$	The processing time of $Task^l_{mn}$ when executed on Ex_q
$RC_{Stage_{mn}}$	The resource cost of the n^{th} stage in Job_m
α^{lq}_{mn}	Binary decision variable, indicates whether the $Task^l_{mn}$ is assigned to Ex_q
RC_{Job_m}	The resource cost of the m^{th} job in a Spark application

In a Spark cluster, there is a set of available computational resources (e.g., CPU, memory) in each node (virtual machine) for use during task execution. This subsection focuses on two dimensions, CPU and memory, to abstract computational resources and constructs a cost model for Spark based on them, as shown in (5-11).

$$RC_{Task_{mn}^{l}} = f_k \cdot t_{Task_{mn}^{l}} \tag{5}$$

$$RC_{Stage_{mn}} = \sum_{0\le l\le|Stage_{mn}|}\sum_{0\le q\le|Exe|} \times RC_{Task_{mn}^{l}} \cdot \alpha_{mn}^{lq} \tag{6}$$

$$RC_{Job_m} = \sum_{0\le n\le|Job_m|}\sum_{0\le l\le|Stage_{mn}|}\sum_{0\le q\le|Exe|} \times RC_{Task_{mn}^{l}} \cdot \alpha_{mn}^{lq} \tag{7}$$

$$RC = \sum_{0\le m\le|App|}\sum_{0\le n\le|Job_m|}\sum_{0\le l\le|Stage_{mn}|} \times \sum_{0\le q\le|Exe|} RC_{Task_{mn}^{l}} \cdot \alpha_{mn}^{lq} \tag{8}$$

$$\alpha_{mn}^{lq} = \begin{cases} 1 & \text{if the } Task_{mn}^{l} \text{ is assigned to } Ex_q \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

The related constraints are as follows:

$$\sum_{0\le l\le|Exe|} \alpha_{mn}^{lq} = 1 \tag{10}$$

$$\forall \,\text{task}_{mn}^{l} \in \text{Stage}_{mn} \tag{11}$$

In addition, some experimental indicators are shown in Table 3.

1.Throughput

In order to evaluate the job execution performance of the experimental cluster under different scheduling algorithms, this section introduces the indicator of throughput. During the experiment, the specific information of throughput is recorded in bench.log (HiBench log file).

2.Performance cost ratio

In order to quantify the availability of different scheduling algorithms, this section introduces the indicator of Performance Cost Ratio (PCR). PCR, fully known as the performance cost ratio, originally means the ratio of the performance of a product to its price, which is used to reflect whether the product is worth buying. However, this section combines the performance cost ratio with the relevant concepts of Spark and eventually abstracts it as (12).

$$PCR = \frac{Performance_{alg_i}}{Cost_{alg_i}}, i \in (0, 1, 2, 3) \tag{12}$$

Since this section introduces throughput to evaluate the job execution performance of the cluster under different scheduling algorithms, the relationship between them can be abstracted as (13).

$$Performance_{alg_i} = Throughput_{alg_i}, i \in (0, 1, 2, 3) \tag{13}$$

In addition, the fixed cost of nodes used under different scheduling algorithms needs to be summed up to measure the usage cost of different scheduling algorithms, as shown in (14).

$$Cost_{alg_i} = \left(\sum_{k\in\eta}\sum_{j\in\delta_k} f_k \cdot \alpha_{jk}\right)_{alg_i}, i \in (0, 1, 2, 3) \tag{14}$$

Table 3 Definition of symbols

Symbol	Definition
PCR	Performance cost ratio, to measure the availability of different scheduling algorithms
$Performance_{alg_i}$	Job execution performance of cluster under the i^{th} scheduling algorithm
$Cost_{alg_i}$	Usage cost of the i^{th} scheduling algorithm, i.e., the total fixed cost of the nodes required by the i^{th} scheduling algorithm
$Throughput_{alg_i}$	The throughput of the cluster under the i^{th} scheduling algorithm, which measures the execution performance of the job
alg_i	The i^{th} scheduling algorithm,$i \in (0, 1, 2, m)$, corresponding to FIFO, FAIR, BFD, ILP,SLCTS

Ultimately, the experimental indicator PCR is shown in (15).

$$PCR = \frac{Throughput_{alg_i}}{((\sum_{k\in\eta}\sum_{j\in\delta_k} f_k\cdot\alpha_{jk})_{alg_i})}, i\in(0,1,2,3) \quad (15)$$

4 Cost-aware Scheduling Algorithm and Partitioning Algorithm

4.1 Cost-aware Scheduling Algorithm

Based on the hierarchical relationship among Spark applications, jobs, stages and tasks, the cost model is constructed in Section 3. In this section, a low-cost task scheduling algorithm for Spark is designed according to the above model, and the core steps of the algorithm are shown in Fig. 1.

For the first step, there are four sub-steps as follows: evaluate the performance of nodes, calculate the input data hit rate of nodes, prioritize nodes and set the parallelism of tasks. Next, the above sub-steps are described in detail in this paper, and some relevant symbols will be presented in Table 4.

For the evaluation of node performance, this paper mainly focuses on CPU and I/O performance. Firstly, for the CPU performance of the node, we will use the calibration frequency of CPU as the experimental data. And then, for the I/O performance of the node, we will use the following Linux commands to test, as shown in Table 5.

For the data obtained from the test, it can be brought into (16) to quantify the node performance.

$$PS_{node_i} = w_{cpu}\cdot\frac{cpuFre_{node_i}}{cpuFre_{stand}} + w_{io}\cdot(\frac{1}{2}\cdot\frac{iow_{node_i}}{iow_{stand}} + \frac{1}{2}\cdot\frac{ior_{node_i}}{ior_{satnd}}) \quad (16)$$

$$w_{cpu} + w_{io} = 1 \quad (17)$$

By default, the node will not perceive the specific type of the task (e.g., computing-intensive or IO-intensive). Therefore, there is no need to prefer CPU or I/O while evaluating node performance, which means that w_{cpu} and w_{io} in formula (11) should be equal, and combined with (17), we can get that both w_{cpu} and w_{io} are 0.5. If there are some different emphases on CPU or I/O according to the type of task, just adjust the values of w_{cpu} and w_{io}.

A custom script counts the distribution of HDFS data blocks and their copies (3 by default) in the nodes and then brings in (18) to get the input data to hit the ratio of the nodes.

$$DH_{node_i} = \frac{DR_{node_i}}{\sum_{i\in N} DR_{node_i}} \quad (18)$$

The determination of node priority is achieved by the following steps:

Firstly, the process of data normalization is shown in (19) and (20).

$$\theta^{PS}_{node_i} = \frac{PS_{node_i} - \min(PS_{node_i})}{\max(PS_{node_i}) - \min(PS_{node_i})}, \forall i\in N \quad (19)$$

$$\theta^{DH}_{node_i} = \frac{DH_{node_i} - \min(DH_{node_i})}{\max(DH_{node_i}) - \min(DH_{node_i})}, \forall i\in N \quad (20)$$

Secondly, the standard deviation method is used here to calculate the target weights, which can be expressed as (21) to (24).

$$\sigma_{PS} = \sqrt{\frac{\sum_{i\in N}(PS_{node_i} - \overline{PS})^2}{|N|}} \quad (21)$$

$$\sigma_{DH} = \sqrt{\frac{\sum_{i\in N}(DH_{node_i} - \overline{DH})^2}{|N|}} \quad (22)$$

$$\omega_{PS} = \frac{\sigma_{PS}}{\sigma_{PS}+\sigma_{DH}} \quad (23)$$

$$\omega_{DH} = \frac{\sigma_{DH}}{\sigma_{PS}+\sigma_{DH}} \quad (24)$$

Finally, quantify the priority of nodes, which is shown in (25).

$$NP_{node_i} = \omega_{PS}\cdot\theta^{PS}_{node_i} + \omega_{DH}\cdot\theta^{DH}_{node_i}, \forall i\in N \quad (25)$$

For the setting of task parallelism, we refer to the official recommendations of Spark, which states that the number of tasks per CPU core should be limited to 2-3 to make the best use of computing resources. It means that the ratio between the number of tasks and the number of CPU cores should be set to 2:1 or 3:1 to make more full use of the cluster's CPU resources. During the experiment, the ratio will be set to 3:1 and

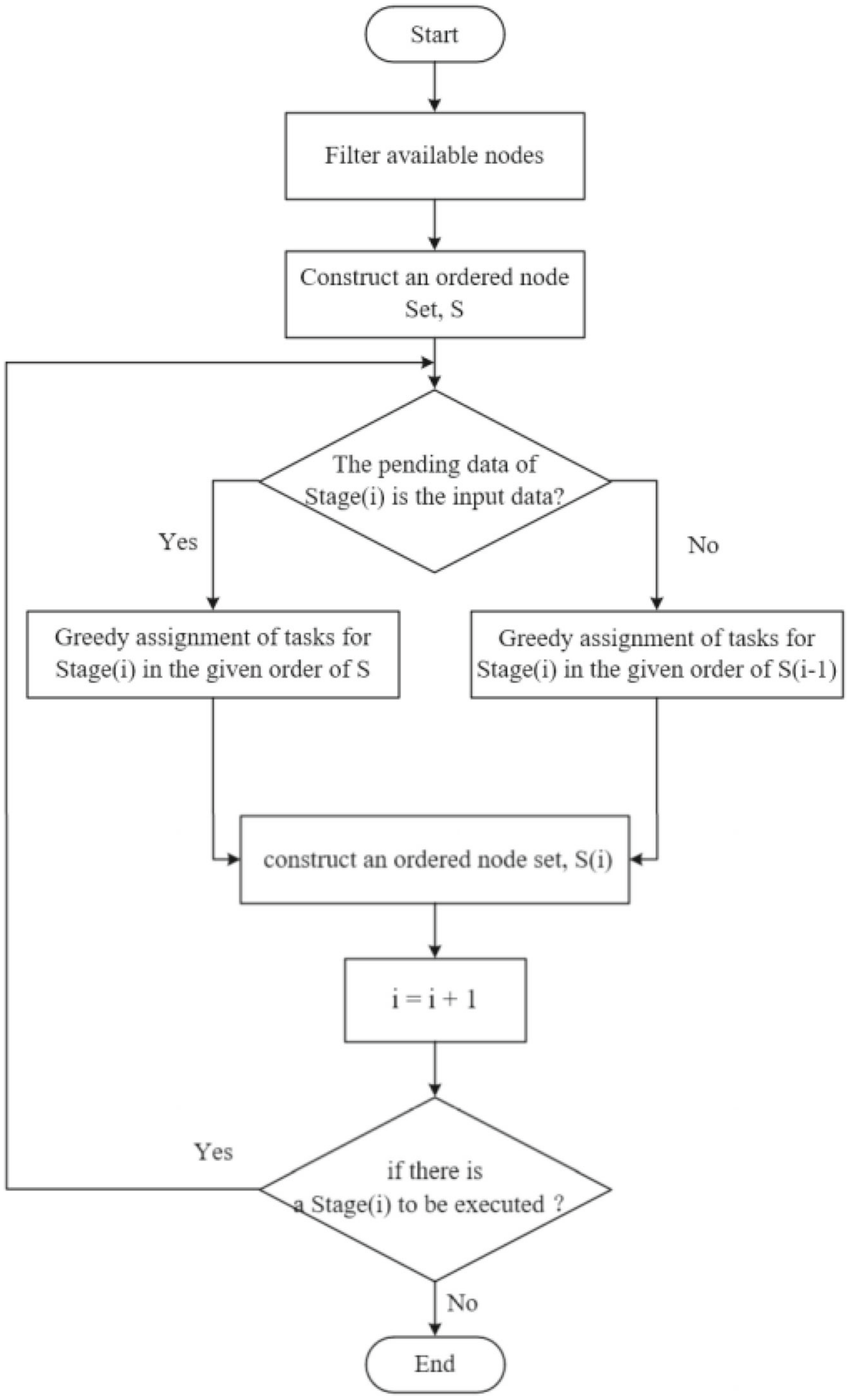

Start
Filter available nodes
Construct an ordered node Set, S
The pending data of Stage(i) is the input data?
Yes
No
Greedy assignment of tasks for Stage(i) in the given order of S
Greedy assignment of tasks for Stage(i) in the given order of S(i-1)
construct an ordered node set, S(i)
i = i + 1
Yes
if there is a Stage(i) to be executed ?
No
End

Fig. 1 Flowchart of the SLCTS algorithm

Table 4 Symbol reference table

Symbol	Definition
PS_{node_i}	Performance score of the i^{th} node
$cpuFre_{node_i}$	CPU calibration frequency of the i^{th} node
$cpuFre_{stand}$	The lowest CPU calibration frequency of all nodes
iow_{node_i}	I/O write speed of the i^{th} node
iow_{stand}	The lowest I/O write speed of all nodes
ior_{node_i}	I/O read speed of the i^{th} node
ior_{stand}	The lowest I/O read speed of all nodes
w_{cpu}	The weight of CPU performance
w_{io}	The weight of IO performance
DH_{node_i}	Data hit rate of the i^{th} node
DR_{node_i}	The number of data hits for the i^{th} node
N	The set of node index, $i \in N, N = (0, 1, 2, \dots)$
$\theta^{PS}_{node_i}$	Performance score of the i^{th} node after data normalization
$\theta^{DH}_{node_i}$	Data hit rate of the i^{th} node after data normalization
$\min(\dots)$	The minimum value
$\max(\dots)$	The maximum value
$\overline{PS}$	Average of the performance score of all nodes
$\overline{DH}$	Average of the data hit rate of all nodes
σ_{PS}	Standard deviation of performance score
σ_{DH}	Standard deviation of data hit rate
ω_{PS}	The weight of performance for node
ω_{DH}	The weight of data hit rate for node
NP_{node_i}	The priority of the i^{th} node

the available nodes will be filtered according to the priority of the above nodes.

The pseudo-code of the Spark low-cost task scheduling algorithm is shown in Algorithm 1. The general process is as follows. Firstly, try to obtain the priority of nodes. Moreover, if the pending data of the current stage is the input data from HDFS, sort the available nodes according to the existing node priority. If the pending data of the current stage is the output data from the previous stage, sort the available nodes according to the distribution of task execution in each node in the previous stage. Then, take out the nodes in turn and judge whether they have enough CPU resources. If the CPU resources of the current node are sufficient, the data locality of the tasks to be assigned will be judged next. At last, assign tasks to the current node centrally until its CPU resources have been allocated.

The complexity of the algorithm is analyzed as follows: firstly, in the session of node sorting, fast ranking is used as the sorting algorithm. Therefore, the average time complexity of this session is $O(n \times logn)$. Secondly, in the session of task placement, assuming that the number of available nodes is m, the number of available CPU cores of each node is n, and the number of tasks in the task set is l, in that way, the time complexity of this session is $O(m \times n \times l)$. In summary, the overall time complexity of the algorithm is $O(n \times logn + m \times n \times l)$.

Table 5 Performance test instruction table

Test objectives	Test methods
CPU performance	Calibration frequency
I/O write speed	time dd if= /dev/zero of=test.file bs=8k count=100000 oflag=direct
I/O read speed	time dd if=test.file of=/dev/null bs=8k count=100000 iflag=direct

4.2 Load Balancing Partitioning Algorithm

Due to the particularity of the digital era of the Internet, the volume of data to be processed is bound to grow larger and more complex. Therefore, there is a high risk of using inappropriate data partitioning strategies due to the unknown characteristics of data, resulting in data skewing and thus degrading the overall performance of data processing. In fact, data skew includes key skew and hash skew. The key skewing refers to the fact that in the Spark Shuffle process, all identical keys are aggregated into a partition by the corresponding Reduce task after hashing and modulo operations, and data skewing occurs if the amount of data corresponding to the key is huge. The hash skew refers to the fact that in the Spark Shuffle process, all different keys with the same hash value are processed by the same Reduce task after the hash modulo operation. If the amount of data corresponding to the hash value is particularly huge at that time, a hash skew will occur. In fact, the key skew problem has been effectively mitigated by preaggregation on the map side. However, the hash skew problem has not been improved, and it still affects the overall exe-

Algorithm 1 Pseudo-code of SLCTS algorithm.

```
Input: offers
Output: task allocation
1: function RESOURCE_OFFERS()
2:     nodeInfo ⇐ sc.getNodePriority
3:     sortedTaskSets ⇐ rootPool.getSortedTaskSetQueue
4:     stage ⇐ sortedTaskSets.taskSet.stage
5:     if stage's pending data is input data then
6:         sortedOffers ⇐ getSortedOffersByBasicIndicators(offers, nodeInfo)
7:     else
8:         sortedOffers ⇐ getSortedOffersByShuffleMapTask Number(offers,host-ToTaskNum)
9:     end if
10:    tasks ⇐ sortedOffers.map
11:    availableCpus ⇐ sortedOffers.map(o => o.cores)
12:    for taskSet in sortedTaskSets do
13:        resourceOfferSingleTaskSet(taskSet,sortedOffers, available Cpus,tasks)
14:    end for
15:    return tasks
16: end function
17: function RESOURCE_OFFER_SINGLE_TASKSET(taskSet, sortedOffers, availableCpus, tasks)
18:    for i ⇐ sortedOffers.indices do
19:        execId ⇐ sortedOffers(i).executorId
20:        host ⇐ sortedOffers(i).host
21:        if availableCpus(i) >= CPUS_PER_TASK then
22:            for maxLocality ⇐ taskSet.myLocalityLevels do
23:                task ⇐ taskSet.resourceOffer(execId, host, maxLocality)
24:                stage ⇐ taskSet.taskSet.stage
25:                if stage is ShuffleMapStage then
26:                    countTaskNumberOnEachHostForShuffle-Map-Stage(hostToTaskNum, host)
27:                end if
28:                tasks(i) + = task
29:                availableCpus(i) − = CPUS_PER_TASK
30:                Continue
31:            end for
32:        end if
33:    end for
34: end function
```

cution efficiency of the job and increases the resource usage overhead during job execution.

We propose a reduced task load balancing partitioning algorithm RTLBPA based on prior information data. The core idea of this algorithm is to disperse and aggregate different keys with the same hash value in the data into different partitions to be processed by different reduce tasks. This algorithm can achieve load balancing for all tasks. Finally, the execution time of the whole reduce phase can be reduced and the overall execution efficiency of the job can be improved.

Table 6 Hash distribution factor symbol and definition table

Test objectives	Test methods
α	Hash distribution factor
DV_{hash}	The amount of data with the same hash value but different keys in the HDFS input partition
DV_{total}	Total size of HDFS input partition data volume

The symbols and interpretations related to hash distribution factors are shown in Table 6.

The hash distribution factor is introduced mainly to quantify the proportion of data with the same hash value but different keys in the input partition in the whole partition data, which can be abstracted as (26).

$$\alpha = \frac{DV_{hash}}{DV_{total}} \tag{26}$$

The relevant symbols and interpretations of the experimental indicators are shown in Table 7.

To measure the degree of data skewing during job execution, the CoV (coefficient of variation [29]) is introduced in this chapter to evaluate it, as shown in (27). A larger coefficient of variation indicates a higher degree of data skewing.

$$CoV = \frac{\sqrt{\frac{\sum_{i=1}^{N_r}(TL_i - \overline{TL})^2}{N_r}}}{\overline{TL}} \tag{27}$$

The pseudo-code implementation of the RTLBPA algorithm is shown in Algorithm 2.

Table 7 Relevant symbols and definitions of experimental indicators

Symbol	Definition
CoV	Coefficient of variation, which measures the degree of data skew during job execution
N_r	Number of tasks in the Reduce phase
TL_i	The amount of data processed by the i^{th} Reduce task
$\overline{TL}$	The average of the amount of data processed by all tasks in the Reduce phase, i.e. the average of the task load

Algorithm 2 Pseudo-code of RTLBPA algorithm.

```
Input: key, numPartitions(the Number of downstream Stage partitions)
Output: partitionIndex(The index of the downstream partition corresponding to the current key)
1: hashContainer ⇐ new HashSet()
2: keyContainer ⇐ new HashSet()
3: keyHashContainer ⇐ new HashMap()
4: function GET_PARTITION()
5:     case key of
6:     case 1:
7:     null
8:     return 0
9:     case 2:
10:     keyHash ⇐ key.hashCode()
11:     if !keyContainer.contains(key) && !hashContainer.contains(keyHash) then
12:         keyContainer.add(key)
13:         hashContainer.add(keyHash)
14:         keyHashContainer.put(key, keyHash)
15:         return Utils.nonNegativeMod(keyHash, numPartitions)
16:     else if $!keyContainer.contains(key)\&\&hash-Container.\ contains(keyHash)$ then
17:         keyContainer.add(key)
18:         newHashValue ⇐ keyHash + Random.nextInt (numPartitions)
19:         keyHashContainer.put(key, newHashValue)
20:         return Utils.nonNegativeMod(newHashValue, numPartitions)
21:     else
22:         realHash ⇐ keyHashContainer.get(key)
23:         return Utils.nonNegativeMod(realHash, numPartitions)
24:     end if
25:     end Case
26: end function
```

5 Evaluation and Analysis

5.1 Experimental Setup

In this experiment, this paper uses HiBench workloads to test the proposed algorithm in a Spark cluster composed of 12 nodes. Moreover, this paper also records the execution time of each stage, application execution time, the resource cost of the cluster, average CPU utilization, throughput, and PCR. The details of the cluster are shown in Table 8. Hadoop [30], HiBench, and other software required for experiments are also installed in these nodes and the details of the software are shown in Table 9.

HiBench is a big data benchmarking tool developed by Intel, and users can evaluate the speed, throughput, and resource utilization of different big data frameworks during task execution with it. Currently, there are five sub-modules in HiBench: hadoopbench, stormbench, sparkbench, flinkbench, and gearpumpbench, which correspond to the benchmarks of five big data frameworks such as Hadoop, Storm, Spark, and Flink, respectively. The workloads in the benchmark test set can generally be divided into six categories: Micro, ML (Machine Learning), SQL, Websearch, Graph, and Streaming. To investigate the batch processing capability of the Spark framework, this paper mainly used Sort and WordCount in Micro benchmark under the sparkbench module as the benchmark test applications.

5.2 Performance Evaluation for Cost-aware Scheduling Algorithm

This section evaluates the cost of a Spark cluster with different workloads, data sizes, and the number of partitions. Here we mainly use HiBench to generate test data for two workloads(Sort and Wordcount) under two data scales(small and large).

In theory, task-level cost minimization can be achieved using (5- 9) in Section 3. Also, it is possible to choose the granularity of calculating tasks, stages, or jobs for costing purposes. However, in actual experiments, the virtual machines used do not have such a flexible rental method. In this paper, three sizes of virtual machines, large, medium and small, are used to conduct experiments based on the cost of computing jobs, as shown in in (28). Where, t_{Job_i} denotes the execution time of the i^{th} job in the application.

$$RC = \sum_{k \in \eta} f_k \cdot \sum_{i \in [0,m]} t_{Job_i} \tag{28}$$

This section compares the proposed algorithm with the FIFO scheduling algorithm, FAIR scheduling algorithm and Best Fit Decreasing (BFD) scheduling algorithm and ILP algorithm from four aspects: execution time, cost of the cluster, CPU resource utilization, throughput and performance cost ratio.

The following schedulers are compared with our proposed scheduling algorithms:

1. FIFO: FIFO is the default scheduler for Apache Spark, which is deployed on top of Apache Mesos. It places the executors of jobs sequentially in a round-robin fashion. In this way, all executors of

Table 8 Experimental cluster details

VM type	CPU cores	Memory(GB)	Pricing(ECS)	Quantity
m1.small	4	8	0.78¥/h	4
m1.medium	6	12	1.17¥/h	4
m1.large	8	16	1.56¥/h	4

multiple jobs are evenly distributed throughout the cluster, thus improving job performance.

2. FAIR: The FAIR scheduler is similar to the FIFO scheduler, but it prevents resource contention among jobs.
3. BFD: The BFD is one of the state-of-the-art models proposed in [21] which can effectively improve the cost efficiency of a cloud deployed Spark cluster. The BFD algorithm sorts the list of virtual machines (VMs) according to the resource availability of VMs, and reduces the use of nodes by placing the executors centrally. In fact, Buyya et al. in [22] also proposed the First Fit (FF) algorithm, which is very similar to BFD and both belong to the heuristic algorithm for solving the bin packing problem. The BFD algorithm is further optimized and improved in this paper, so it is chosen as a comparison algorithm.
4. ILP: The ILP algorithm is also one of the state-of-the-art models proposed in [21]. In fact, the optimization target, executor placement constraints, and resource capacity constraints are dynamically generated by using the current cluster resource availability and the current job executor's resource requirements, which can generate the optimal cost-efficient placements for all the executors of each job.

The analysis of the time complexity of these state-of-the-art algorithms is as follows. The time complexity of BFD is $\mathrm{O}\left(\left(m^2 \log(m)\right)(p+r)\right)$, where p and r is the total number of deadline constrained and regular jobs, respectively that need to be scheduled. And the total number of VMs in the cluster is m. In addition, The time complexity of ILP is $\mathrm{O}\left(\left(2^k m\log(m)\right)(p+r)\right)$, where k is the maximum number of slots available for placing executors across all the VMs.

Table 9 Experimental software details

Software	Version
Hadoop	2.7.6
HiBench	7.1.1
Spark	2.4.7
Scala	2.11.12
Java	1.8.0.131
Maven	3.6.3

5.2.1 Sort

The Sort workload is mainly used to sort words. As can be seen from Fig. 2, the workload consists of a Job that is ultimately divided into two Stages (ShuffleMapStage and ResultStage) due to the Shuffle operation, sortByKey. The Shuffle Write in Spark Shuffle occurs in the ShuffleMapStage, which is mainly responsible for persisting intermediate data, while the Shuffle Read occurs in the ResultStage, which is mainly responsible for pulling intermediate data. Since the Sort program does not contain a pre-aggregation operator and the number of partitions is always less than the value of spark.shuffle.sort.bypassMergeThreshold (200 by default), the Shuffle Writer used in the Shuffle Write process is BypassMergeSortShuffleWriter.

Figures 3, 4, 5, 6, 7, 8 and 9 show the execution time of each stage, application execution time, cost of the cluster, average CPU utilization, throughput, and PCR for Sort at different partition numbers.

As shown in Fig. 3, compared with the other four algorithms, the proposed algorithm SLCTS has the shortest execution time of ShuffleMapStage and improves at least 10.77% and 11.29% over the existing improved algorithm (BFD) and ILP, respectively. For the proposed algorithm, it takes into account the performance difference among nodes and the distribution of input data in each node, which avoids the weak use of high-performance nodes to a certain extent. At the same time, during task scheduling, it allocates tasks to nodes with a higher hit rate of input data in priority, so that the performance overhead caused by data transmission in the network is relatively reduced. The FIFO and the

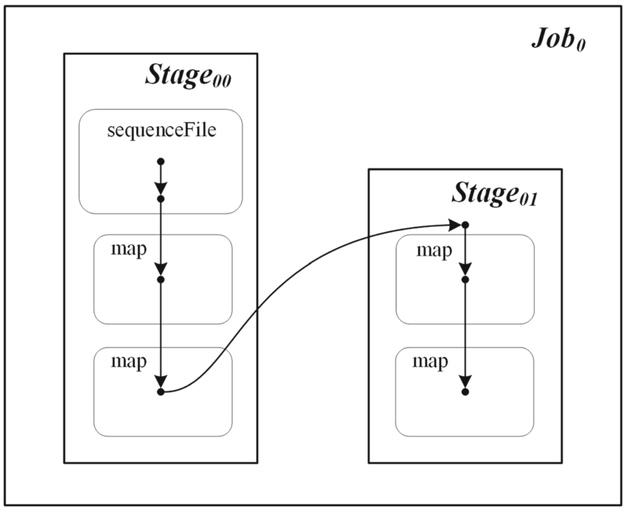
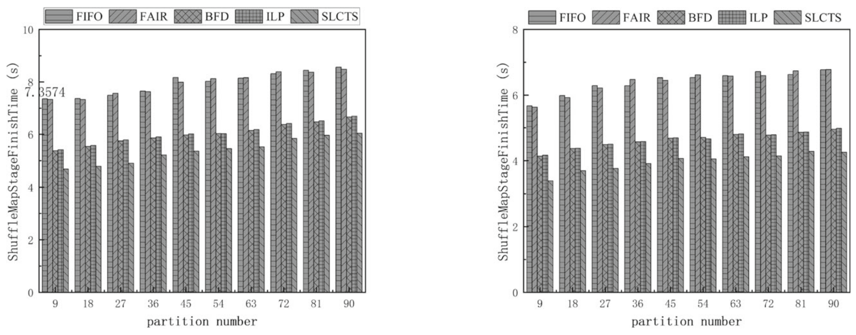


Fig. 2 DAG logical plan of Sort

FAIR algorithms use distributed polling to place tasks to all nodes, which can not fully use CPU resources for computation, so the execution time is the longest. The BFD and ILP algorithms reduce the use of nodes by placing executors centrally, which improve the utilization of CPU resources. In this way, the computational power is increased and the execution time is shortened.

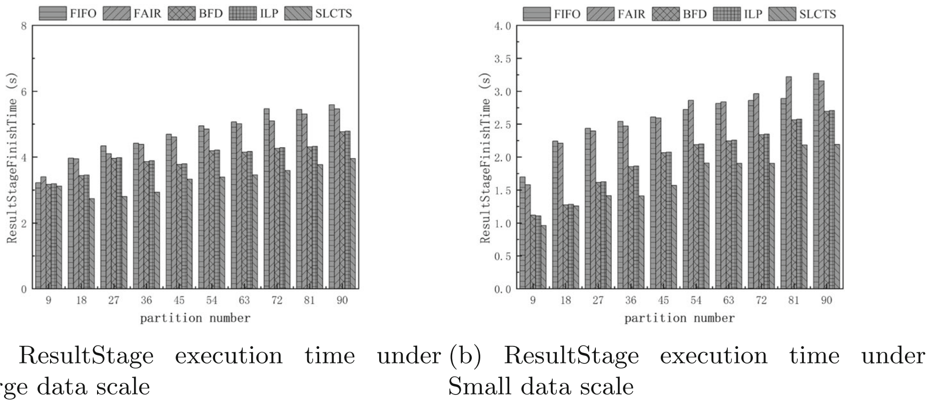
As shown in Fig. 4, compared with the other four algorithms, the proposed algorithm SLCTS has the shortest execution time of ResusltStage, which is at least 15.54% better than the existing improved algorithm (BFD) and 15.84% better than ILP. The proposed algorithm, since it fully considers the distribution of the output data of the previous stage in each node, prioritizes the assignment of tasks to the nodes with more local intermediate data during task scheduling, which will reduce the amount of data that needs to be retrieved from the remote end. In this way, the performance overhead caused by data transmission in the network will be reduced. The other four algorithms do not take this into account, so a large amount of data is transmitted through the network, which prolongs the execution time of this stage.

As shown in Fig. 5, compared with the other four algorithms, the proposed algorithm SLCTS has the shortest execution time of the Sort workload and our algorithm improves by at least 13.06% and 13.4% compared to the existing improved algorithm (BFD) and ILP, respectively. At the same time, as the number of partitions increases, the execution time of the application tends to increase gradually. Since the execution time of the application is mainly determined by the



(a) ShuffleMapStage execution time under Large data scale (b) ShuffleMapStage execution time under Small data scale

Fig. 3 The comparison of ShuffleMapStage execution time for Sort: (a) ShuffleMapStage execution time under Large data scale; (b) ShuffleMapStage execution time under Small data scale



(a) ResultStage execution time under Large data scale (b) ResultStage execution time under Small data scale

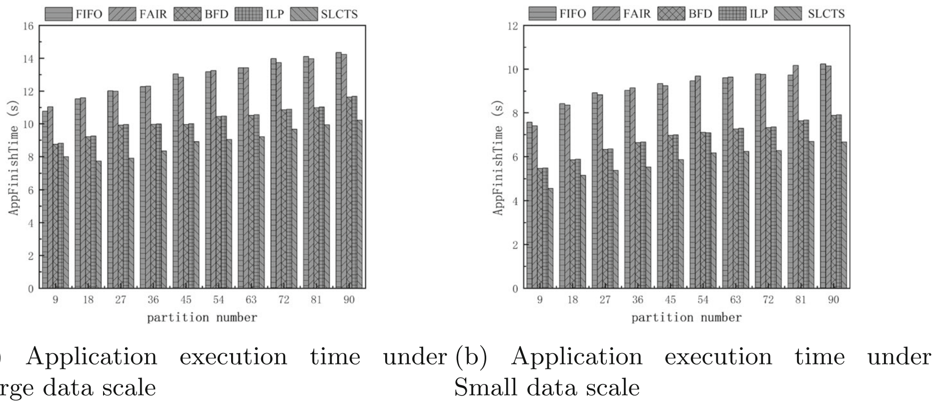
Fig. 4 The comparison of ResultStage execution time for Sort: (a) ResultStage execution time under Large data scale; (b) ResultStage execution time under Small data scale

Job, and the execution time of the Job is mainly determined by the Stage, based on the above analysis of the execution time of the two Stages, this section will not elaborate on the execution time of the application here. In general, the proposed algorithm outperforms the other three algorithms.

Also, as the number of partitions increases, the execution time of the application tends to increase gradually.

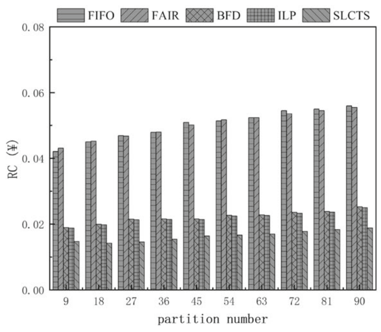
Based on the cost model in Secion 3 and the application execution time obtained from the tests, we can easily calculate the cost of the Sort workload during the job execution. As shown in Fig. 6, the cost of the cluster (RC) tends to increase slowly with the increase of the number of partitions under both Large and Small data scales for all scheduling algorithms. Compared with the other four algorithms, the proposed algorithm SLCTS has the lowest cost of the cluster and improves by at least 26.18% and 25.44% compared to BFD and ILP, respectively. For the proposed algorithm, it ensures the efficiency of task execution while reducing the use of low-priority nodes, so the cost of the cluster is reduced to a large extent. The FIFO algorithm and the FAIR algorithm place tasks on all available compute nodes by distributed polling during task scheduling, so that all nodes are used for task execution, and thus the cost is quite high. Although the BFD and ILP algorithms reduce the number of nodes used, the tasks are executed less efficiently than the proposed algorithm, and therefore, the cost of the cluster is slightly higher.

In order to count the CPU resource usage of the cluster during job execution, here this section mainly uses Linux scripts to monitor the Coarse-Grained Executor Backend process of each node at the second level

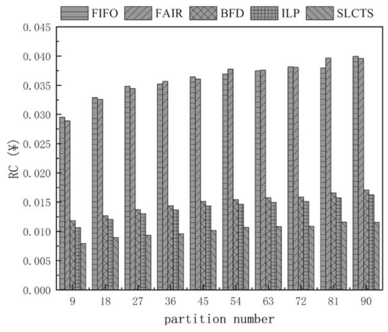


(a) Application execution time under Large data scale (b) Application execution time under Small data scale

Fig. 5 The comparison of application execution time for Sort: (a) Application execution time under Large data scale; (b) Application execution time under Small data scale



(a) Resource usage cost under Large data scale

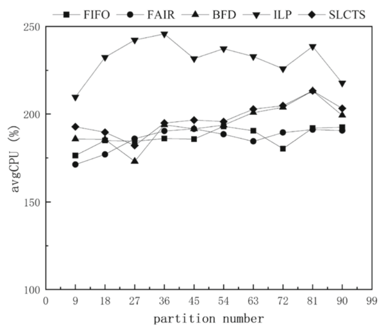


(b) Resource usage cost under Small data scale

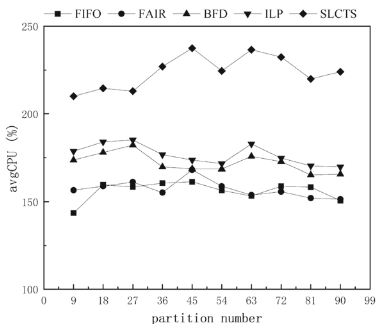
Fig. 6 The comparison of cost for Sort: (a) Resource usage cost under Large data scale; (b) Resource usage cost under Small data scale

and write the monitoring data to the log file. After the application is executed, the log file will be analyzed and counted uniformly. The results are shown in Fig. 7. Compared with the other four algorithms, the proposed algorithm SLCTS has the highest CPU utilization. For the proposed algorithm, it adjusts the ratio between the number of tasks and the number of CPU cores (task parallelism), which reduces the idle time of the CPU to a certain extent and improves CPU utilization. The FIFO and FAIR algorithms simply allocate tasks to occupy computing resources, which results in wasted CPU resources and low utilization. The BFD and ILP algorithms place the executors as centrally as possible to reduce the waste of computing resources. However, there are still a small number of CPU cores unused due to factors such as resource requirements of executors and resource constraints of nodes. Therefore, the CPU utilization of the BFD and ILP algorithms is slightly lower than that of the proposed algorithm.

In order to evaluate the job execution performance of the cluster under different scheduling algorithms, we use throughput as an indicator to evaluate. By analyzing and counting the experimental data in the log files, the final results are shown in Fig. 8. The throughput tends to decrease slowly with the increase of the number of partitions under both Large and Small data scales for all scheduling algorithms. In general, compared with the other four algorithms, the proposed algorithm SLCTS has the highest throughput. This is because the proposed algorithm outperforms the other three scheduling algorithms in terms of application execution time and the total amount of input data is the same. Therefore, the throughput of the cluster is highest overall under this scheduling algorithm. In other words, the job exe-

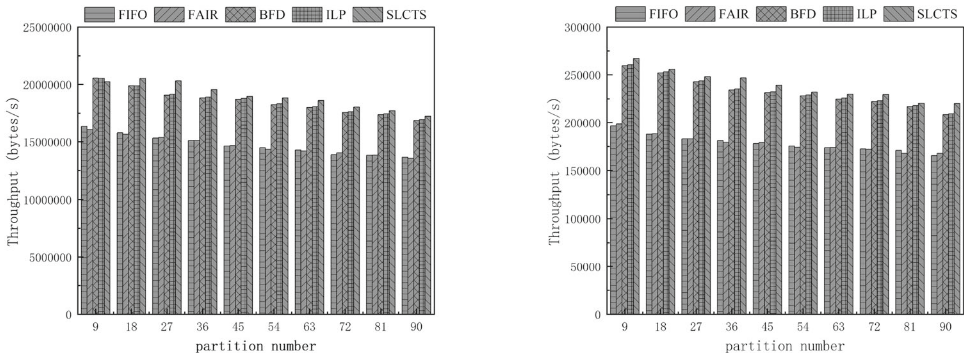


(a) Average CPU utilization under Large data scale



(b) Average CPU utilization under Small data scale

Fig. 7 The comparison of average CPU utilization for Sort: (a) Average CPU utilization under Large data scale; (b) Average CPU utilization under Small data scale



(a) Throughput under Large data scale (b) Throughput under Small data scale

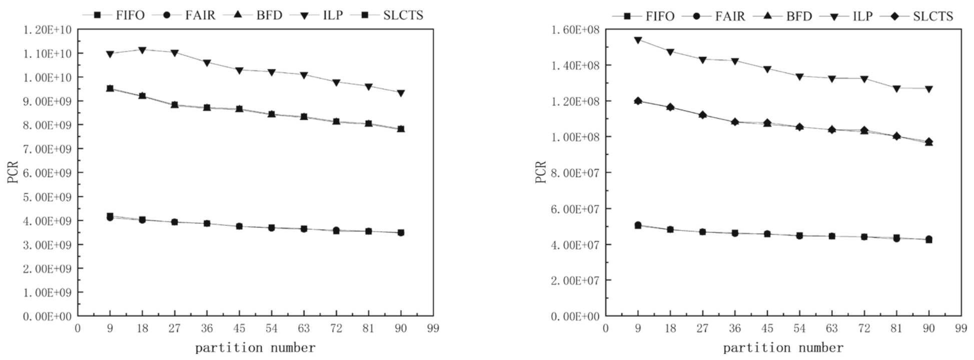
Fig. 8 The comparison of throughput for Sort: (a) Throughput under Large data scale; (b) Throughput under Small data scale

cution performance of the cluster under this scheduling algorithm is optimal.

In addition, in order to quantify the availability of different scheduling algorithms, we use PCR as an indicator to evaluate, as shown in Fig. 9. For different data scales, the proposed algorithm is the best in terms of performance cost ratio, followed by the ILP algorithm, the BFD algorithm, the FIFO algorithm, and the FAIR algorithm are the lowest and comparable. For the proposed algorithm, the throughput of the cluster is the highest and the sum of the fixed cost of the required nodes is the smallest, so the performance cost ratio of this algorithm is the highest. However, the throughput of the cluster is slightly lower with the BFD and ILP algorithms and the sum of the fixed costs of the required nodes is relatively higher. Therefore, the performance cost ratio of the BFD and ILP algorithms is lower than that of the proposed algorithm.

5.2.2 WordCount

The WordCount workload is mainly used to count the number of occurrences of each word. As shown in Fig. 10, the workload consists of a job that is ultimately divided into two stages (ShuffleMapStage and ResultStage) due to the Shuffle operation, reduceByKey. Shuffle refers to the process of transferring data from the Map side to the Reduce side, so it is usually divided into two parts: Shuffle Write and Shuffle Read. The former corresponds to the ShuffleMapStage, which is responsible for persisting the output data on the Map side, and the latter corresponds to the ResultStage, which is responsible for pulling data to the Reduce side. The reduceByKey operator in the WordCount program is a pre-aggregation operator. Therefore, in the current Spark version, the Shuffle Writer used in the Shuffle Write process is SortShuffleWriter.



(a) PCR under Large data scale (b) PCR under Small data scale

Fig. 9 The comparison of PCR for Sort: (a) PCR under Large data scale; (b) PCR under Small data scale

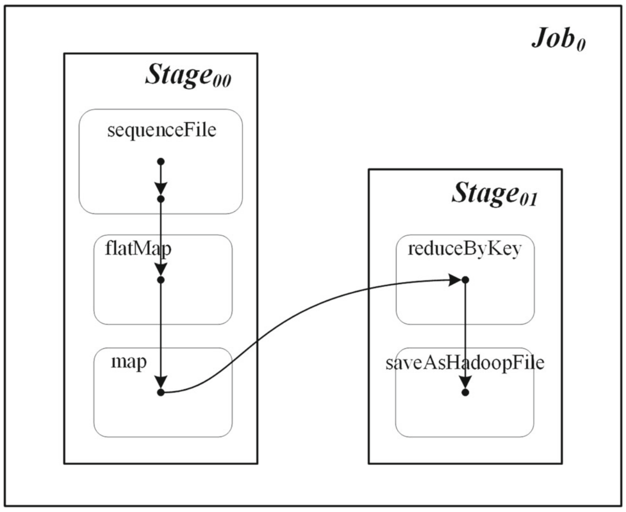
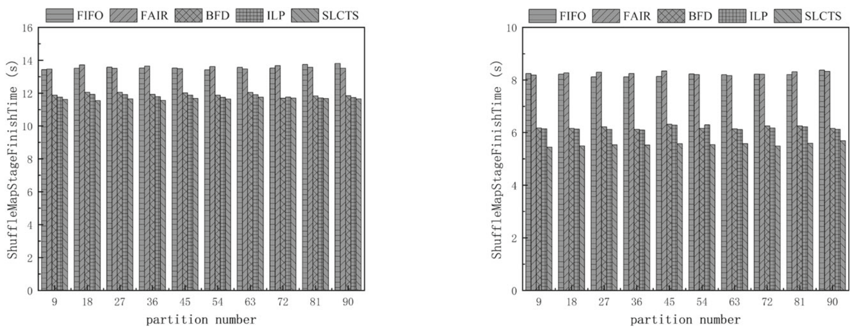


Fig. 10 DAG logical plan of WordCount

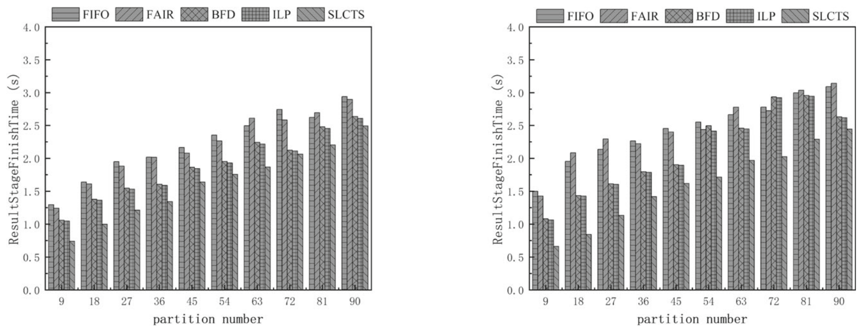
Figures 11, 12, 13, 14, 15, 16 and 17 show the execution time of each stage, application execution time, cost of the cluster, average CPU utilization, throughput and PCR for WordCount at different partition numbers.

As shown in Fig. 11, under Large and Small data scales, there is not any significant trend for the execution time of ShuffleMapStage with the increasing number of partitions under the four scheduling algorithms. This is because SortShuffleWriter does not create too many temporary files during the persistence of the data but only performs a swipe operation after the partition data has been processed. And compared with the other three algorithms, the proposed algorithm SLCTS has the shortest execution time of ShuffleMapStage, with at least 2.27% and 1.37% improvement compared to the existing improved (BFD) and ILP algorithms, respectively. For the proposed algorithm, it takes into account the performance difference among nodes and the distribution of input data in each node, which avoids the weak use of high-performance nodes to a certain extent. At the same time, it will give priority to assigning tasks to nodes with a higher hit rate of input data during task scheduling, so that the performance overhead caused by data transmission in the network is relatively reduced. The FIFO and FAIR algorithms use distributed polling to place tasks to all nodes, which can not fully use CPU resources for computation, so the execution time is the longest. The BFD and ILP algorithm reduce the use of nodes by placing executors centrally, which improve the utilization of CPU resources. In this way, the computational power is increased and the execution time is shortened.



(a) ShuffleMapStage execution time under Large data scale (b) ShuffleMapStage execution time under Small data scale

Fig. 11 The comparison of ShuffleMapStage execution time for WordCount: (a) ShuffleMapStage execution time under Large data scale; (b) ShuffleMapStage execution time under Small data scale

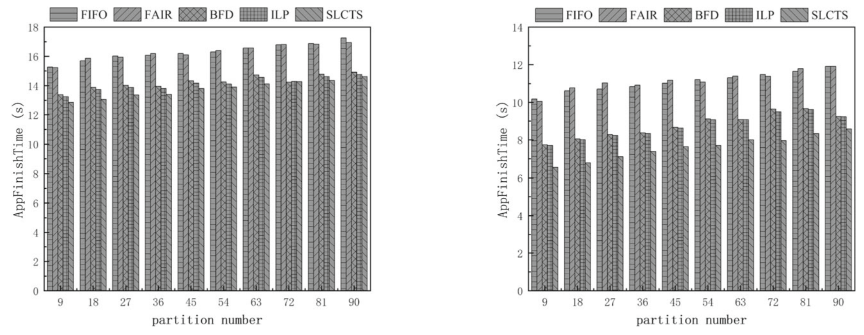


(a) ResultStage execution time under Large data scale (b) ResultStage execution time under Small data scale

Fig. 12 The comparison of ResultStage execution time for WordCount: (a) ResultStage execution time under Large data scale; (b) ResultStage execution time under Small data scale

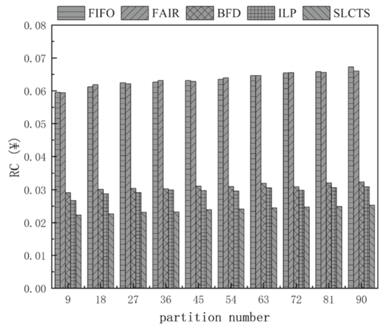
As shown in Fig. 12, compared with the other four algorithms, the proposed algorithm SLCTS has the shortest execution time of ResusltStage and improves at least 15.39% and 14.49% over the existing improved algorithm (BFD) and ILP, respectively. The proposed algorithm will relatively reduce the amount of data that needs to be retrieved from the remote end since it fully considers the distribution of the output data of the previous phase at each node. In this way, the performance overhead caused by data transmission in the network will be reduced. The other four algorithms do not take this into account so that a large amount of data is transmitted through the network, which prolongs the execution time of this stage.

As shown in Fig. 13, compared with the other four algorithms, the proposed algorithm SLCTS has the shortest execution time of the Sort workload and improves over the BFD and ILP algorithms by at least 3.36% and 2.43%, respectively. At the same time, the execution time of the application tends to increase gradually with the increase in the number of partitions. Since the execution time of the application is mainly determined by the Job, and the execution time of the Job is mainly determined by the Stage, based on the above analysis of the execution time of the two Stages, we will not elaborate on the execution time of the application here. In general, the proposed algorithm outperforms the other four algorithms.

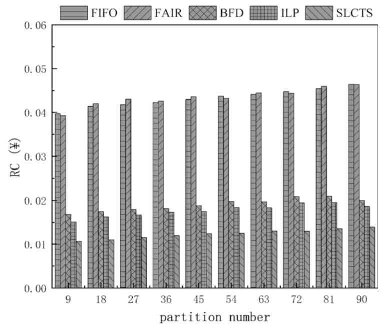


(a) Application execution time under Large data scale (b) Application execution time under Small data scale

Fig. 13 The comparison of application execution time for WordCount: (a) Application execution time under Large data scale; (b) Application execution time under Small data scale



(a) Resource usage cost under Large data scale

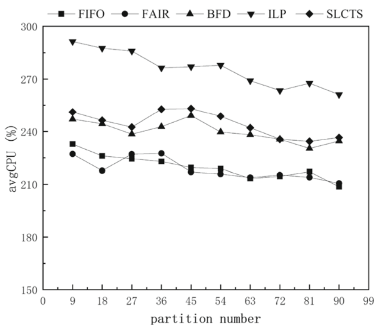


(b) Resource usage cost under Small data scale

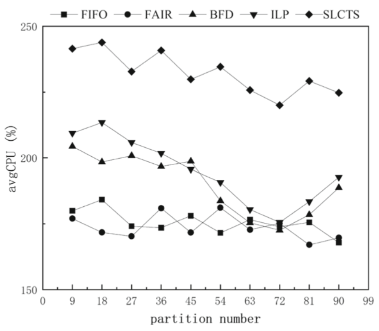
Fig. 14 The comparison of cost for WordCount: (a) Resource usage cost under Large data scale; (b) Resource usage cost under Small data scale

Based on the resource cost model in Secion 3 and the application execution time obtained from the tests, we can easily calculate the cost of the Sort workload during the job execution. As shown in Fig. 14, the cost of the cluster (RC) tends to increase slowly with the increase in the number of partitions under both Large and Small data scales for all scheduling algorithms. Compared with the other four algorithms, the proposed algorithm SLCTS has the lowest cost of the cluster and improves at least 22.71% and 19.2% compared to the BFD and ILP algorithms, respectively. For the proposed algorithm, it ensures the efficiency of task execution while reducing the use of low-priority nodes, so the cost of the cluster is reduced to a large extent. The FIFO algorithm and the FAIR algorithm place tasks on all available compute nodes by distributed polling during task scheduling, so that all nodes are used for task execution, and thus the cost is quite high. Although the BFD and ILP algorithms reduce the number of nodes used, the tasks are executed less efficiently than the proposed algorithm. Therefore, the cost of the cluster is slightly higher.

To count the CPU resource usage of the cluster during job execution, we mainly use Linux scripts to monitor the CoarseGrainedExecutorBackend process of each node at the second level and write the monitoring data to the log file. After the application is executed, the log file will be analyzed and counted uniformly. The results are shown in Fig. 15. Compared with the other three algorithms, the proposed algorithm SLCTS has the highest CPU utilization. For the proposed algorithm, it adjusts the ratio between the number of tasks and the number of CPU cores (task parallelism), which reduces the idle time of the CPU to a certain extent and



(a) Average CPU utilization under Large data scale



(b) Average CPU utilization under Small data scale

Fig. 15 The comparison of average CPU utilization for WordCount: (a) Average CPU utilization under Large data scale; (b) Average CPU utilization under Small data scale

improves CPU utilization. The FIFO algorithm and the FAIR algorithm simply allocate tasks to occupy computing resources, which results in the waste of CPU resources and low utilization. The BFD and ILP algorithms place the executors as centrally as possible to reduce the waste of computing resources. However, there are still a small number of CPU cores unused due to factors such as resource requirements of executors and resource constraints of nodes. Therefore, the CPU utilization of the BFD and ILP algorithms is slightly lower than that of the proposed algorithm.

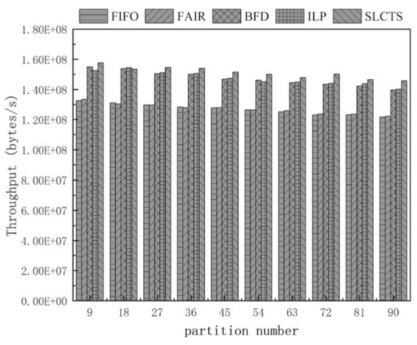
In order to evaluate the job execution performance of the cluster under different scheduling algorithms, we use throughput as an indicator to evaluate. By analyzing and counting the experimental data in the log files, the final results are shown in Fig. 16. The throughput tends to decrease slowly with the increase in the number of partitions under both Large and Small data scales for all scheduling algorithms. In general, compared with the other four algorithms, the proposed algorithm SLCTS has the highest throughput. This is because the proposed algorithm outperforms the other four scheduling algorithms in terms of application execution time, and the total amount of input data is the same. Therefore, the throughput of the cluster is highest overall under this scheduling algorithm. In other words, the job execution performance of the cluster under this scheduling algorithm is optimal.

In addition, in order to quantify the availability of different scheduling algorithms, PCR is used as an evaluation metric, as shown in Fig. 17. For different data scales, the proposed algorithm is the best in terms of performance cost ratio, followed by the ILP algorithm, the BFD algorithm, the FIFO algorithm and the FAIR algorithm are the lowest and comparable. For the proposed algorithm, the throughput of the cluster is the highest and the sum of the fixed cost of the required nodes is the smallest, so the performance cost ratio of this algorithm is the highest. However, the throughput of the cluster under the BFD and ILP algorithms is slightly lower and the sum of the fixed costs of the required nodes is relatively higher. Therefore, the performance cost ratio of the BFD and ILP algorithms is lower than that of the proposed algorithm.

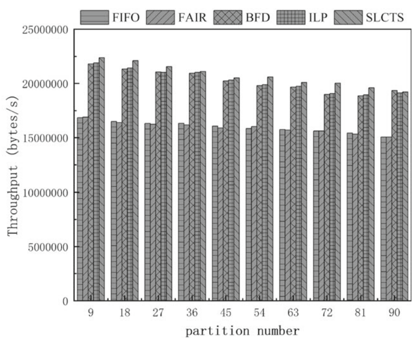
5.3 Performance Evaluation for RTLBPA Algorithm

In order to verify the effectiveness of the proposed RTLBPA partitioning algorithm, WordCount containing the reduceByKey operator is mainly selected as the benchmark test program in this section, and experiments are conducted based on RTLBPA scheduling algorithm and partition number 90 at different data sizes. First, generating different datasets based on different values of α, thus simulating different distribution characteristics of operational data. Then, the input data required for the WordCount test is generated by the HiBench script, which is stored in HDFS in the form of chunked files. Replace the original chunk file in HDFS with a data file with a value of 0 for α using a Linux script and then experiment with this as a benchmark. Finally, a chunked file is randomly selected and replaced with a data file of different values of α for comparison experiments.

The algorithm proposed in this section is implemented based on prior knowledge of the distribution



(a) Throughput under Large data scale



(b) Throughput under Small data scale

Fig. 16 The comparison of throughput for WordCount: (a) Throughput under Large data scale; (b) Throughput under Small data scale

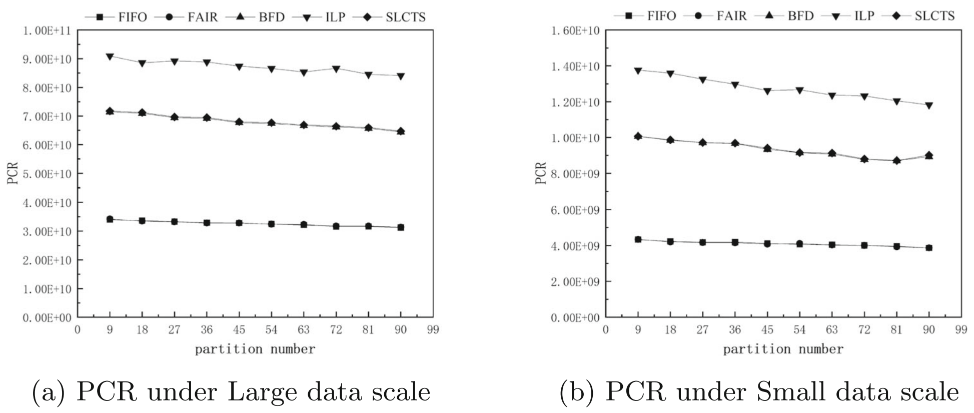


Fig. 17 The comparison of PCR for WordCount: (a) PCR under Large data scale; (b) PCR under Small data scale

characteristics of data. Therefore, in order to simulate the distribution characteristics of data (input data), this section constructs the following ten datasets based on two different data sizes and five different hash distribution factors, as shown in Table 10.

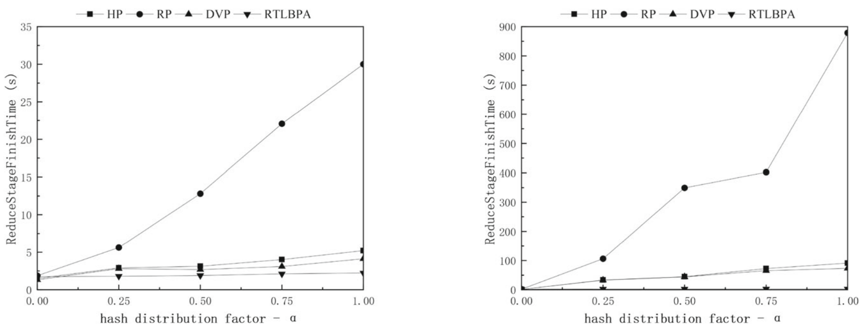
Experiments are conducted based on WordCount, and the comparison algorithms used include HashPartitioner, Range-Partitioner, and DVP. The core idea of HashPartitioner is to aggregate all keys with the same hash value into one partition to be processed by one Reduce task. The core idea of RangePartitioner is to use a reservoir sampling algorithm to calculate the maximum key for each partition in the Reduce phase and use it as a boundary to divide all the keys into the corresponding partitions. The DVP algorithm turns the same key into multiple different keys by attaching random prefixes so that the data originally processed by one task is dispersed to multiple partitions for local aggregation, and then the random prefixes are removed for global aggregation.

The operation of four algorithms(HashPartitioner, RangePartitioner, DVP, and RTLBPA) with different hash distribution factors when the input partition size is 4MB and 20MB is shown in Fig. 18. In the figure, HP stands for HashPartitioner and RP stands for RangePartitioner. As α increases, the other three algorithms show an increasing trend in the execution time of the Reduce phase, while the algorithm proposed in this section has no significant change. It can be seen that the HashPartitioner, RangePartitioner, and DVP take longer to execute in the Reduce phase, and the overall execution efficiency of the job is lower as the size of the data increases. The algorithm proposed in this section can effectively reduce the execution time of the Reduce phase and improve the overall execution efficiency of the job through task load balancing.

For the algorithm proposed in this section, it aggregates all the different keys with the same hash value into different partitions to be handled by different tasks, relieving the pressure of a single task that needs to handle excessive data and improving the overall execution efficiency of the job. Therefore, as the value of α increases, even if the input data with the same hash value but different keys accounts for an increasing proportion, it does not significantly impact the execution time of the Reduce phase. The HashPartitioner algorithm aggregates all keys with the same hash value into a partition handled by a single Reduce task. Then, as the proportion of data corresponding to each hash value

Table 10 Synthetic datasets with different data sizes and different hash distribution factors

Datasets	Size	α
uniform		0
hash distribution 1-4	4MB	0.25
hash distribution 2-4		0.5
hash distribution 3-4		0.75
hash distribution 4-4		1
uniform		0
hash distribution 1-4		0.25
hash distribution 2-4	20MB	0.5
hash distribution 3-4		0.75
hash distribution 4-4		1



(a) ReduceStage execution time under small data scale (b) ReduceStage execution time under large data scale

Fig. 18 The comparison of ReduceStage execution time for WordCount: (a) ReduceStage execution time under small data scale; (b) ReduceStage execution time under large data scale

becomes larger and larger, the number of data records aggregated to the corresponding partition increases relatively, and the execution time of the Reduce phase increases. However, since the data corresponding to different hash values is divided into different partitions after the modulo operation, the execution time increment is smaller. The RangePartitioner algorithm calculates the maximum key value for each partition in the Reduce phase and divides all the keys into the corresponding partitions using this as the boundary. Since data with the same hash value but different keys are divided into the same partition after being calculated by the reservoir sampling algorithm, they are always processed by one task, which leads to a significantly longer execution time of the task. The DVP algorithm improves the processing of too much data by a single task through a two-stage aggregation, but the data with the same hash value and different keys cannot be decentralized, and can only be processed through the original modular operation. Therefore, as the value of α increases, the execution time of the Reduce stage of the DVP algorithm increases.

As shown in Fig. 19, HP stands for HashPartitioner and RP stands for RangePartitioner. When the value of α is 0, i.e., when the input data are uniformly randomly distributed, the CoV values of the algorithm proposed in this section are not significantly different compared to those of the other three algorithms. When the value of α is not 0, the CoV value of the algorithm proposed in this section is significantly lower than that of the other three algorithms. And as the value of α increases, the value does not change significantly and stabilizes overall. The overall CoV value of RangePartitioner is the highest and the data skew is the most serious, while the overall CoV values of HashPartitioner and DVP are in the middle, and the data skew is more moderate than the former. As a result, the algorithm proposed in this section can effectively balance the load of Reduce tasks and reduce the skew of data during job execution.

6 Conclusions and Future Directions

In this paper, a low-cost task scheduling algorithm is proposed for Spark based on a heterogeneous cloud environment. The main idea of this algorithm is to prioritize all nodes by integrating the heterogeneity of the cluster and the distribution of data to be computed at different stages. Moreover, adjust the parallelism of tasks and set the ratio between the number of tasks and the number of CPU cores used during job execution to 3:1 to improve the CPU resource utilization of the whole cluster, trying to reduce the use of low priority nodes to avoid unnecessary resource usage overhead based on ensuring good performance. Finally, the experimental results show that the proposed algorithm can reduce the cost of the cluster while improving CPU resource utilization.

The proposed SLCTS, a low-cost task scheduling algorithm based on Spark heterogeneous clusters, is

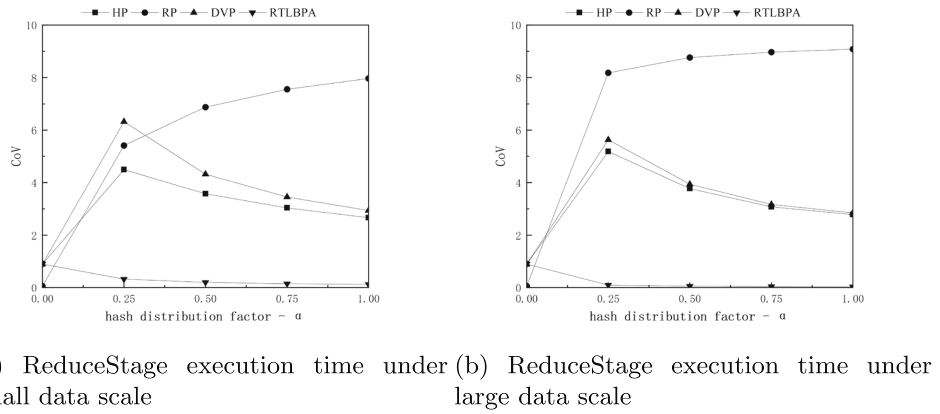


Fig. 19 The comparison of CoV for WordCount: (a) ReduceStage execution time under small data scale; (b) ReduceStage execution time under large data scale

deployed and run in Standalone mode, and its resource scheduling is in coarse-grained mode. It means that the required resources must be requested before the program runs and cannot be changed during the runtime. In the future, we intend to use Apache Mesos as the resource scheduler for the cluster, which can realize on-demand allocation of computing resources, dynamic supply, and release of resources. For the RTLBPA algorithm, it needs to analyze and capture the distribution characteristics of business data in advance. In the future, an algorithm can be studied to capture and analyze the characteristics of data during the program operation, and then a more general and effective partitioning strategy can be developed to solve the different data skewing problems according to the characteristics of data. In addition, with the popularity of hybrid cloud, we also plan to apply the ideas presented in this paper to hybrid cloud environments in the future.

Acknowledgements This work was supported by Chongqing Science and Technology Commission Project (Grant No: cstc2018jcyjAX0525), Key Research and Development Projects of Sichuan Science and Technology Department (Grant No: 2019YFG0107).

Author contributions Hongjian Li: Proposed an idea, Experiment, Wrote the manuscript. Lisha Zhu: Proposed an idea, Experiment, Wrote the manuscript. Shuaicheng Wang: Helped to wrote also several sections of the manuscript, Proofreading. Lei Wang: Helped to wrote also several sections of the manuscript, Proofreading.

Funding This work was supported by Chongqing Science and Technology Commission Project (Grant No: cstc2018jcyjAX0525; Recipient: Hongjian Li), Key Research and Development Projects of Sichuan Science and Technology Department (Grant No: 2019YFG0107;Recipient: Hongjian Li).

Data Availability Statement The datasets generated during the current study are available from the corresponding author on reasonable request.

Declarations

Competing interests The authors declare that they have no known conflict financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

1. Ruan, J., Zheng, Q., Dong, B.: Optimal resource provisioning approach based on cost modeling for spark applications in public clouds. 1–4 (2015). https://doi.org/10.1145/2843966.2843972
2. Chen, J., Li, K., Tang, Z., Bilal, K., Yu, S., Weng, C., Li, K.: A parallel random forest algorithm for big data in a spark cloud computing environment. IEEE Trans. Parallel Distrib. Syst. **28**(4), 919–933 (2017). https://doi.org/10.1109/TPDS.2016.2603511
3. Li, C., Li, L.Y.: Optimal resource provisioning for cloud computing environment. J. Supercomput. **62**(2), 989–1022 (2012). https://doi.org/10.1007/s11227-012-0775-9
4. Xu, F., Zheng, H., Jiang, H., Shao, W., Liu, H., Zhou, Z.: Cost-effective cloud server provisioning for predictable performance of big data analytics. IEEE Trans. Parallel Distrib. Syst. **30**(5), 1036–1051 (2019). https://doi.org/10.1109/TPDS.2018.2873397
5. Lattuada, M., Barbierato, E., Gianniti, E., Ardagna, D.: Optimal resource allocation of cloud-based spark applications. IEEE Trans. Cloud Comput. **10**(2), 1301–1316 (2022). https://doi.org/10.1109/TCC.2020.2985682
6. Cheng, D., Zhou, X., Lama, P., Wu, J., Jiang, C.: Cross-platform resource scheduling for spark and mapreduce

on yarn. IEEE Trans. Comput. **66**(8), 1341–1353 (2017). https://doi.org/10.1109/TC.2017.2669964
7. Wang, J., Li, X., Ruiz, R., Yang, J., Chu, D.: Energy utilization task scheduling for mapreduce in heterogeneous clusters. IEEE Trans. Serv. Comput. **15**(2), 931–944 (2022). https://doi.org/10.1109/TSC.2020.2966697
8. Cheng, D., Rao, J., Guo, Y., Jiang, C., Zhou, X.: Improving performance of heterogeneous mapreduce clusters with adaptive task tuning. IEEE Trans. Parallel Distrib. Syst. **28**(3), 774–786 (2017). https://doi.org/10.1109/TPDS.2016.2594765
9. Maroulis, S., Zacheilas, N., Kalogeraki, V.: A framework for efficient energy scheduling of spark workloads. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS), pp. 2614–2615 (2017). https://doi.org/10.1109/ICDCS.2017.179
10. Zacheilas, N., Kalogeraki, V.: Chess: Cost-effective scheduling across multiple heterogeneous mapreduce clusters. In: 2016 IEEE international conference on autonomic computing (ICAC), pp. 65–74. IEEE (2016)
11. Xu, Y., Liu, L., Ding, Z.: Dag-aware joint task scheduling and cache management in spark clusters. In: 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 378–387. IEEE (2020)
12. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: Fair allocation of multiple resource types. In: 8th USENIX symposium on networked systems design and implementation (NSDI 11) (2011)
13. Dimopoulos, S., Krintz, C., Wolski, R.: Justice: a deadline-aware, fair-share resource allocator for implementing multi-analytics. In: 2017 IEEE international conference on cluster computing (CLUSTER), pp. 233–244. IEEE (2017)
14. Wang, Y., Xue, G., Qian, S., Li, M.: An online cost-efficient scheduler for requests with deadline constraint in hybrid clouds. In: 2017 international conference on progress in informatics and computing (PIC), pp. 318–322. IEEE (2017)
15. Wang, G., Xu, J., Liu, R., Huang, S.: A hard real-time scheduler for spark on yarn. In: 2018 18th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID), pp. 645–652 IEEE (2018)
16. Hussain, A., Aleem, M., Iqbal, M.A., Islam, M.A.: Sla-ralba: cost-efficient and resource-aware load balancing algorithm for cloud computing. J. Supercomput. **75**(10), 6777–6803 (2019)
17. Wang, B., Tang, J., Zhang, R., Ding, W., Liu, S., Qi, D.: Energy-efficient data caching framework for spark in hybrid dram/nvm memory architectures. In: 2019 IEEE 21st international conference on high performance computing and communications; IEEE 17th international conference on smart city; IEEE 5th international conference on data science and systems (HPCC/SmartCity/DSS), pp. 305–312. IEEE (2019)
18. Li, H., Wang, H., Fang, S., Zou, Y., Tian, W.: An energy-aware scheduling algorithm for big data applications in spark. Cluster Comput. **23**(2), 593–609 (2020)
19. Li, H., Wei, Y., Xiong, Y., Ma, E., Tian, W.: A frequency-aware and energy-saving strategy based on dvfs for spark. J. Supercomput. **77**(10), 11575–11596 (2021)
20. Hu, Z., Li, B., Qin, Z., Goh, R.S.M.: Job scheduling without prior information in big data processing systems. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS), pp. 572–582. IEEE (2017)
21. Islam, M.T., Srirama, S.N., Karunasekera, S., Buyya, R.: Cost-efficient dynamic scheduling of big data applications in apache spark on cloud. J. Syst. Softw. **162**, 110515 (2020)
22. Islam, M.T., Wu, H., Karunasekera, S., Buyya, R.: Sla-based scheduling of spark jobs in hybrid cloud computing environments. IEEE Trans. Comput. **71**(5), 1117–1132 (2021)
23. Cheng, D., Zhou, X., Wang, Y., Jiang, C.: Adaptive scheduling parallel jobs with dynamic batching in spark streaming. IEEE Trans. Parallel Distrib. Syst. **29**(12), 2672–2685 (2018)
24. Tang, Z., Zeng, A., Zhang, X., Yang, L., Li, K.: Dynamic memory-aware scheduling in spark computing environment. J. Parallel Distrib. Comput. **141**, 10–22 (2020)
25. Neciu, L.-F., Pop, F., Apostol, E.-S., Truică, C.-O.: Efficient real-time earliest deadline first based scheduling for apache spark. In: 2021 20th international symposium on parallel and distributed computing (ISPDC), pp. 97–104. IEEE (2021)
26. Chen, W., Xie, G., Li, R., Li, K.: Execution cost minimization scheduling algorithms for deadline-constrained parallel applications on heterogeneous clouds. Cluster Comput. **24**(2), 701–715 (2021)
27. Islam, M.T., Karunasekera, S., Buyya, R.: Performance and cost-efficient spark job scheduling based on deep reinforcement learning in cloud computing environments. IEEE Trans. Parallel Distrib. Syst. **33**(7), 1695–1710 (2021)
28. Cheng, Y., Wu, C., Liu, Y., Ren, R., Xu, H., Yang, B., Qi, Z.: Ops: Optimized shuffle management system for apache spark. In: 49th international conference on parallel processing-ICPP, pp. 1–11. (2020)
29. Fu, Z., Tang, Z., Yang, L., Li, K., Li, K.: Imrp: a predictive partition method for data skew alleviation in spark streaming environment. Parallel Comput. **100**, 102699 (2020)
30. Samadi, Y., Zbakh, M., Tadonki, C.: Performance comparison between hadoop and spark frameworks using hibench benchmarks. Concurr. Comput. Pract. Exp. **30**(12), 4367 (2018)