Computer Science and Information Systems 20(1):491–512 https://doi.org/10.2298/CSIS220131065F

Optimizing Data Locality by Executor Allocation in Spark Computing Environment*

Zhongming Fu[1], Mengsi He[1]**, Zhuo Tang[2], and Yang Zhang[3]

[1] Computer School, University of South China, and Hunan Provincial Base for Scientific and Technological Innovation Cooperation
Hengyang, Hunan, China, 421001
fuzhongming@hnu.edu.cn
mengsih@163.com

[2] College of Information Science and Engineering, Hunan University, and National Supercomputing Center
Changsha, Hunan, China, 410082
ztang@hnu.edu.cn

[3] Science and Technology on Parallel and Distributed Laboratory (PDL), National University of Defense Technology
Changsha, Hunan, China, 410073
yangzhang15@nudt.edu.cn

Abstract. Data locality is an important concept in big data processing. Most of the existing research optimized data locality from the aspect of task scheduling. However, as the execution container of tasks, the executors started on which nodes can directly affect the locality level achieved by the tasks. This paper tries to improve the data locality by executor allocation for reduce stage in Spark computing environment. Firstly, we calculate the network distance matrix of executors and formulate an optimal executor allocation problem to minimize the total communication distance. Then, when the network distance between executors satisfies the triangular inequality, an approximate algorithm is proposed; and when the network distance between executors does not satisfy the triangular inequality, a greedy algorithm is proposed. Finally, we evaluate the performance of our algorithms in a practical Spark cluster by using several representative micro-benchmarks (Sort and Join) and macro-benchmarks (PageRank and LDA). Experimental results show that the proposed algorithms can decrease the execution time of tasks for lower data communication.

Keywords: communication distance, data locality, executor allocation, spark framework.

1. Introduction

With the increasingly high response requirement of applications in the era of big data, Spark [1] attracts great attention in academia and has become the popular parallel com-

* This is an extension of work presented in the conference paper: Fu, Z., He, M., Tang, Z., Zhang, Y.: Optimizing Data Locality by Executor Allocation in Reduce Stage for Spark Framework. In: Parallel and Distributed Computing, Applications and Technologies, PDCAT 2021, Lecture Notes in Computer Science, Springer, Cham, Vol 13148. DOI: https://doi.org/10.1007/978-3-030-96772-7_32.

** Corresponding author

puting framework for massive data processing [28]. The core abstraction of Spark is resilient distributed dataset (RDD) that can be cached to memory, thus avoiding writing and reading data in HDFS between jobs. Therefore, compared with Hadoop [2], Spark can take advantage of in-memory computing to perform jobs more efficiently.

A typical Spark application contains one or more jobs, and a job usually consists of many stages. Since the stages are executed sequentially, the intermediate output of the former stage is used as the input of the later stage. When the tasks of a stage run in parallel on different nodes, the data communication across the network occurs [22]. As shown in Fig. 1, in the map stage (i.e., the first stage), each task reads a data block to process and outputs the intermediate data to local disks. In the reduce stage (i.e., the subsequent stages), each task fetches part of the intermediate data from the previous stage for processing. This is the so-called shuffle that is a many-to-many communication. The resulting large amount of network traffic in the two stages can congest the network and extend execution time, thereby hindering the system [23].

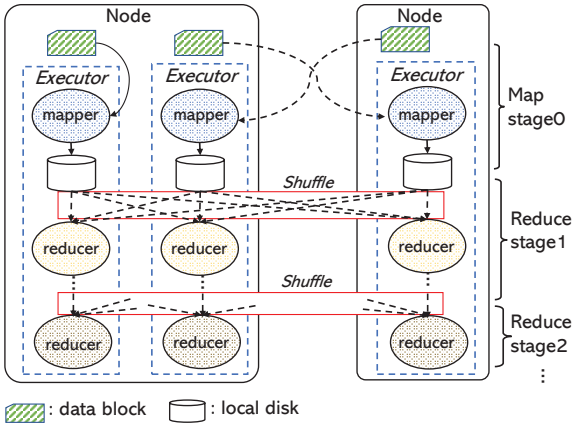


Fig. 1. Data communication between Spark stages.

For improving performance, data locality is a key factor considered by the task scheduling of Spark stages [10]. The task scheduling determines the executor on which node the task runs and the data locality refers to scheduling task close to data, so that the communication overload can be reduced [19], [15]. In particular, in the map stage, the *taskScheduler* uses the delay scheduling algorithm [34] that tries to assign the map task to the node which stores the data block, and in the reduce stage, the *taskScheduler* assigns the reduce task to one of the nodes that holds more intermediate data to the task, thus to minimize the data transfer volume.

In MapReduce frameworks (e.g. Hadoop), most of the existing research focused on optimizing data locality from the aspect of task scheduling [17], [21], [9]. In particular, Guo et al. [16] assign map task to maximize the number of node-local tasks, and Shang et al. [25] dispatch reduce task to minimize the network resources consumption. These task scheduling methods are the direct way to improve the data locality and achieves better performance, but they do not involve the problem of executor allocation in Spark.

As the execution container of tasks, the executors can restrict the nodes available for the task scheduling. This actually affects the locality level achieved by the tasks. On the one hand, if the executor is not started on the nodes in which the data block is located in the map stage, the map task is almost impossible to obtain data locally. On the other hand, if the executors are started on the nodes away from each other in the reduce stage, the reducer has to go through a long network distance to get data. Spark provides two algorithms: *spreadOut* and *noSpreadOut* to decide the executors start up. Unfortunately, neither of them fully exploit the benefits of data locality.

In this paper, we improve the data locality from the view of executor allocation considering the reduce stage in Spark computing environment. In general, the number of reduce stage is much greater than that of map stage, so the reduce stage has an important impact on the whole performance. Compared with the conference version [12], which proposed an approximate algorithm for the case that the network distance between executors satisfies the triangular inequality, this paper also focuses on the executor allocation for the case that the network distance between executors does not satisfy the triangular inequality. The main contributions of this paper are summarized as below.

- We calculate the distance matrix of executors, and formulate an executor allocation problem to minimize the total communication distance. This problem is proved to be an NP-Hard problem.
- When the network distance between executors satisfies the triangular inequality, we propose an approximate algorithm and prove the approximate factor is 2.
- When the network distance between executors does not satisfy the triangular inequality, we propose a greedy algorithm and prove the correctness of the algorithm.
- We implement our algorithms in Spark-3.0.1 and evaluate their performance on representative benchmarks. The experiment results explain that the proposed algorithms can reduce the task execution time for better data locality.

The rest of this paper is organized as follows. Section 2 reviews more related research. Section 3 describes the motivation of our optimization. Section 4 presents the proposed executor allocation algorithms. Experiments and performance evaluation are given in Section 5. Section 6 concludes this paper.

2. Related Work

Many research has been done to optimize the data locality in MapReduce frameworks, which can be categorized as follows:

Task scheduling. In the design of MapReduce, Dean et al. [11] took the locality of map tasks into account to save bandwidth consumption. The priority of tasks scheduled to nodes is classified into three levels: *node-local*, i.e., the task and its data block are on the same node; *rack-local*, i.e., the task and its data block are on different nodes but on the same rack; and *off-rack*, i.e., the task and its data block are on different racks but on a cluster.

Further, using the time-for-space strategy, Zaharia et al. [34] proposed the delay scheduling algorithm. If there is no map task can obtain data locally on the request node, a small amount of time is waited in the hope of obtaining better locality from subsequent nodes. In a cluster that quickly releases resources, the delay scheduling can achieve a higher proportion of node-local tasks while preserving fairness.

Besides the map stage, the data locality for reduce tasks also affects the performance [27], [8], [25]. Tang et al. [31] presented a minimum transmission cost reduce task scheduler (MTCRS). It decides the appropriate launching locations for reduce tasks according to two factors: the waiting time of each reducer and the transmission cost set, which is computed by the sizes and the locations of intermediate data partitions.

In order to alleviate data skew at the same time, Tang et al. [29] provided a reduce placement algorithm CORP. It first uses a reservoir algorithm for sampling the input data to estimate the distribution of keys/values, then on the basis of this, it calculates the distance and cost matrices among the cross node communication. Finally, the related map and reduce tasks are scheduled relatively nearby physical nodes.

Data pre-fetching. From another angle, Sun et al. [26] designed HPSO (High Performance Scheduling Optimizer), a prefetching service based task scheduler to improve data locality for MapReduce jobs. Their idea is to predict the most appropriate nodes to which future map tasks should be assigned and then pre-load the input data to memory without any delaying on running normal tasks.

In [35], Zhang et al. proposed a pre-fetching method based on pre-scheduling in Hadoop systems. The method hides the remote data access delay by pre-fetching, and controls the resource competition by adjusting resource allocation of reduce tasks. Nevertheless, the above pre-fetching techniques may incur additional overhead and could not help to alleviate the network traffic of cluster.

High speed network. In addition, some researchers were dedicated to finding high speed network to speed up the data transmission. Lu et al. [18] proposed a novel design (RPCoIB) of Hadoop RPC with RDMA over InfiniBand networks. RPCoIB provides a JVM-bypassed buffer management scheme and utilizes message size locality to avoid multiple memory allocations and copies in data serialization and deserialization.

In [32], Yan et al. introduced R3S, an RDMA-based in-memory RDD storage layer for Spark. R3S leverages high bandwidth networks and low-latency one-sided RDMA operations to allow Spark nodes to efficiently access intermediate output from a remote node.

In the above studies, these task scheduling methods are the direct way to improve the data locality, but they do not involve the problem of executor allocation in Spark computing environment. Therefore, the executors can restrict the nodes available for the task scheduling. In our early work [13], we proposed an optimal task scheduling algorithm and an executor allocation algorithm to optimize the data locality in the map stage. While in this paper, we focus on the executor allocation in the reduce stage, with the purpose of providing the possibility of better locality level when scheduling the reduce tasks.

3. Motivation

There are two methods: *spreadOut* and *noSpreadOut* provided by Spark to decide on which nodes the executors start. The idea of the *spreadOut* strategy tries to launch the required executors on as many nodes as possible, while *noSpreadOut* goes a inverse way, it launches the executors on as few nodes as far as possible. We show how the executor allocation affects the data locality in reduce task scheduling.

As shown in Fig. 2, suppose that a cluster has 4 idle nodes namely node0, node1, node2, and node3, and the number of executors allowed to start on each node is 3, 2, 2, and

1 respectively, as shown in Fig. 2(a). The number of executors required by the application is 5. For these nodes that are sorted according to the number of executors allowed to start, *spreadOut* takes turns to start the executor on each node until reaching the number requirement, as shown in Fig. 2(b). In contrast, *noSpreadOut* starts the allowed number of executors on a node in turn until the number requirement is met, as shown in Fig. 2(c). For simplicity, we use the hop count to calculated the network distance, so the sum of the communication distance between executors under *spreadOut* and *noSpreadOut* is 21 and 18 respectively. However, given an optimal executor allocation strategy shown by Fig. 2(d), the minimum total communication distance is 8. Then when scheduling the reduce tasks to the nodes and run in the executors, the tasks need to go through a longer network distance to get data under *spreadOut* and *noSpreadOut* than under the optimal strategy.

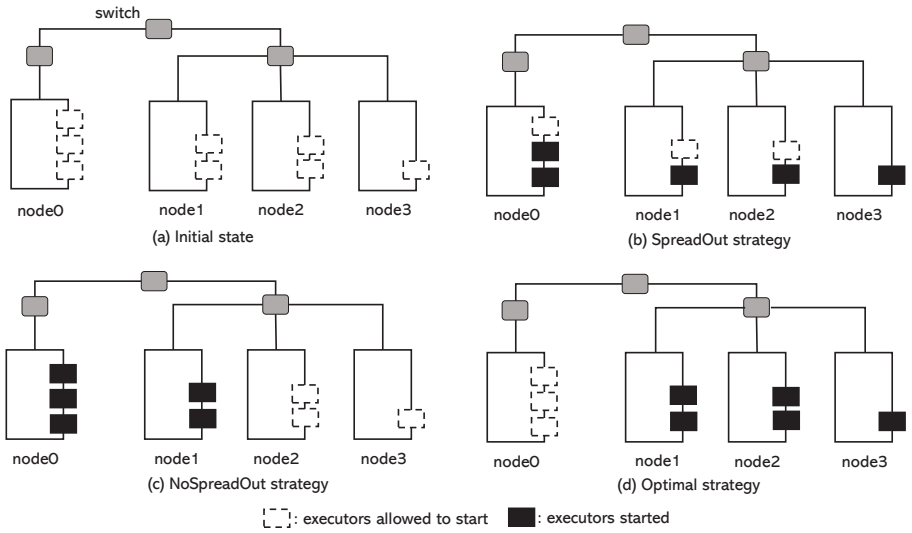


Fig. 2. Different executor allocation strategies.

From the above analysis, we can conclude that *spreadOut* and *noSpreadOut* may cause the executors to start on the nodes away from each other, bringing a great communication overhead when the reduce tasks fetch the intermediate data. Therefore, the poor data locality could be obtained in the task scheduling. It also should be noted that if there is little intermediate data generated in the previous stage, it is difficult to improve job performance by improving the data locality in the reduce stage.

4. Proposed Executor Allocation Algorithm

This section formulates the optimal executor allocation problem, and then presents an approximate algorithm and a greedy algorithm respectively.

4.1. Optimal Executor Allocation Problem

When a Spark application is submitted to the cluster and to be executed, the *master* registers with the resource manager (such as YARN [30]) and applies for the resources to start a group of executors. An executor is the container of running tasks, which is a collection of computing resources (i.e., cpu and memory). A task can be scheduled to run on a node requiring to have idle executors.

For illustrative purposes, some important variables involved in the model are declared in Table 1.

Table 1. Variable Declaration

Variable	Declaration
$N_l, 0 \leq l < \alpha$	The l^{th} node of the cluster
$R_r, 0 \leq r < \beta$	The r^{th} rack of the cluster
$e_i^l, 0 \leq i < m$	The i^{th} executor allowed to start, which is located on the l^{th} node
$d_{ij}, 0 \leq j < m$	The network distance between executor e_i and e_j

We first initialize the network topology of the cluster as the node set $\{N_0, N_1, \cdots, N_{\alpha-1}\}$ and rack set $\{R_0, R_1, \cdots, R_{\beta-1}\}$, $1 \leq \beta \leq \alpha$, where α is the number of nodes and β is the number of racks. In the initial state of allocating executor for an application, some particular data structures are defined as follows:

(1) E: A set of executors allowed to be started on the nodes, the number is m. The element e_i^l represents the i^{th} executor that can be started on the l^{th} node if marked. In the Spark framework, the number of executors allowed to start on each node can be calculated based on the free resources of the node, formalized as:

$$exe\_num_i = \min\{[\frac{free\_cpu_i}{cpu\_conf}], [\frac{free\_memory_i}{memory\_conf}]\}, \tag{1}$$

where exe_num_i indicates the number of executors allowed to start on node N_i, and cpu_conf and $memory\_conf$ are the number of CPUs and memory capacity configured by the executor respectively.

(2) D: A matrix of $m \times m$ represents the network distance between executors of E, represented as:

$$D = \begin{bmatrix} d_{00} & d_{01} & \cdots & d_{0(m-1)} \\ d_{10} & d_{11} & \cdots & d_{1(m-1)} \\ \vdots & \vdots & \ddots & \vdots \\ d_{(m-1)0} & d_{(m-1)1} & \cdots & d_{(m-1)(m-1)} \end{bmatrix},$$

where d_{ij} represents the network distance between executor e_i and e_j. It is noted that d_{ij} is set to 0 when $i = j$.

With the aim of capturing the data locality, we divide the proximity level (PL) of two executors into three levels: (1) if two executors are on the same node, then PL is equal to

0; (2) if two executors are on different nodes of the same rack, then PL is equal to 1; (3) if two executors are on different nodes of different racks, then PL is equal to 2. Then the network distance d_{ij} of D can be specifically calculated as:

$$d_{ij} = \begin{cases} 0, if PL = 0 \\ 2 \times \left(\frac{1}{band_{NS}} + latency_{NS}\right), if PL = 1 \\ 2 \times \left(\frac{1}{band_{NS}} + latency_{NS}\right) + 2 \times \left(\frac{1}{band_{SS}} + latency_{SS}\right), if PL = 2 \end{cases}, \quad (2)$$

where $band_{NS}$ is the network bandwidth from node to switch, $band_{SS}$ is the network bandwidth from switch to switch, $latency_{NS}$ is the network delay from node to switch, and $latency_{SS}$ is the network delay from switch to switch.

In this model, our purpose is to start the required executors on nodes close to each other. Assuming that the number of executors required by the application is k, so the optimal executor allocation problem can be described as selecting a subset $E^{'} \in E$ to minimize the total communication distance between executors. This problem can be formalized as follows by using *Integer Programming*:

$$\begin{aligned} & min \sum_{i=0}^{m-1} \sum_{j=0}^{m-1} d_{ij} \times (x_i \times x_j), \\ subject\ \ to\ \ & \sum_{i=0}^{m-1} x_i = k, x_i \in \{0,1\}, 0 \leq i < m-1, \end{aligned} \quad (3)$$

where x_i is a binary variable, whose value is 1 means that the executor e_i is selected, and value is 0 means that the executor is not selected.

Theorem 1. *The optimal executor allocation problem (abbreviated as the OEA problem) is NP-Hard.*

Proof. The k-clique problem in graph theory can be educible to the *OEA* problem. That is, for any instance of the k-clique, an instance of *OEA* can be created in polynomial time such that solving the instance of *OEA* solves the instance of k-clique as well. According to the NP completeness of the k-clique problem, the *OEA* problem can be proved to be NP-Hard [14].

Definition 1. *k-clique problem: Given a graph $G(V,U)$ and an integer k, determine whether G has a clique of size k.*

Let $P = (V,U,k)$ be an instance of k-clique, where $V = \{0,1,...,m-1\}$ is the vertex set, $U = \{u_{ij} | i,j \in V\}$ is the edge set, and k is a positive integer.

Then give an instance of *OEA*: $Q = (E,D,k)$, where E is the set of executors allowed to start, D is the distance matrix between executors, and k is a integer. For the executors e_i and e_j of E corresponding to i and j of V, the element d_{ij} of D is assigned to:

$$d_{ij} = \begin{cases} 0, & if\ u_{ij} \in U \\ 1, & otherwised' \end{cases} \quad (4)$$

We claim that a clique of size k exists if and only if E contains a subset $E^{'}$ such that the number of elements in the $E^{'}$ is equal to k, and $\sum_{e_i,e_j \in E'} d_{ij} = 0$.

Figure 3 shows an instance of k-clique, which contains the vertices 0, 1, 2, 3, 4, 5. On this basic, an instance of *OEA* can be created and its network distance matrix D is shown in Table 2.

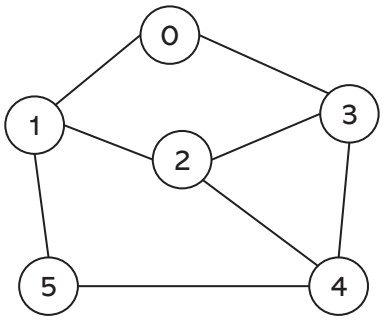


Fig. 3. An instance of k-clique.

Table 2. Network distance matrix

Executor	e_0	e_1	e_2	e_3	e_4	e_5
e_0	0	0	1	0	1	1
e_1	0	0	0	1	1	0
e_2	1	0	0	0	0	1
e_3	0	1	0	0	0	1
e_4	1	1	0	0	0	0
e_5	1	0	1	1	0	0

It can be seen from the above example that the instance of k-clique has a clique of size 3. In the meantime, *OEA* has a solution $E^{'} = \{2, 3, 4\}$ such that k=3, and $\sum_{e_i, e_j \in E'} d_{ij} = 0$.

Hence, the existence of k-clique is a necessary and sufficient condition for the *OEA* problem to be solved, with the size of k and the minimum total communication distance. Hence the proof.

4.2. Approximate Algorithm

We first consider that the network distance between two executors satisfies the triangular inequality [24], such as the homogeneous network of nodes, that is, the distance between any two executors satisfies the following relationship: $d_{ij} \leq d_{iv} + d_{vj}$.

Algorithm 1 describes the approximate algorithm for the optimal executor allocation problem. Firstly, the algorithm selects k nearest executors (including e_i itself) for each executor e_i. For executor e_i, the set of its k nearest executors is represented as $S(e_i)$, and

the sum of network distance from executor e_i to other $k-1$ executors is calculated and represented as $C(e_i)$. Then, find the smallest $C(e_v)$ among all executors, and assign the executor set $S(e_v)$ to $MinSet$. Finally, return to $MinSet$.

Algorithm 1: Approximate Algorithm

Input:
The set of executors allowed to start: E;
The network distance matrix: D;
The number of executors required: k;
Output:
The executors selected to start.

```
begin
    for each executor e_i ∈ E do
        find the set of its k nearest executors: S(e_i);
        calculate the sum of network distance from executor e_i to other k − 1 executors:
        C(e_i) = Σ_{e_j ∈ S(e_i)} d_ij;
    end
    find the smallest C(e_v) and the executor set: MinSet;
    return MinSet.
end
```

The algorithm takes $O(m)$ time to find the nearest k executors by using the optimal algorithm. For m executors, the time it takes is $m \times O(m)$. Therefore, the time complexity of Algorithm 1 is $O(m^2)$, where m is the number of executors allowed to start.

Theorem 2. *The approximate factor of the approximate algorithm to the optimal executor allocation problem is 2.*

Proof. The solution of the approximate algorithm for the optimal executor allocation is $MinSet$, so the total communication distance between executors of $MinSet$ can be represented as $MinCost$. Let $MinSet^*$ be the optimal solution, and the total communication distance between executors of $MinSet^*$ is $MinCost^*$. Then for $MinCost^*$, there is:

$$\begin{aligned} MinCost^* &= \frac{1}{2} \sum_{e_i \in MinSet^*} \sum_{e_j \in MinSet^*} d_{ij} \geq \frac{1}{2} \sum_{e_i \in MinSet^*} \sum_{e_j \in S(e_i)} d_{ij} \\ &= \frac{1}{2} \sum_{e_i \in MinSet^*} C(e_i) \geq \frac{1}{2} \sum_{e_i \in MinSet^*} MinCost = \frac{k}{2} \times MinCost. \end{aligned} \tag{5}$$

For $MinCost$, there is:

$$MinCost = \frac{1}{2} \sum_{e_i \in MinSet} \sum_{e_j \in MinSet} d_{ij}. \tag{6}$$

Let C_{e_v} gets the minimum total communication distance $MinCost$. According to the triangular inequality, there is:

$$\sum_{e_i \in MinSet} \sum_{e_j \in MinSet} d_{ij} \leq \sum_{e_i \in MinSet} \sum_{e_j \in MinSet} (d_{iv} + d_{vj})$$
$$= \sum_{e_i \in MinSet} \sum_{e_j \in MinSet} d_{iv} + \sum_{e_i \in MinSet} \sum_{e_j \in MinSet} d_{vj}$$
$$= \sum_{e_j \in MinSet} \left(\sum_{e_i \in MinSet} d_{vi} \right) + \sum_{e_i \in MinSet} \left(\sum_{e_j \in MinSet} d_{jv} \right)$$
$$= k \times \left(\sum_{e_i \in MinSet} d_{iv} \right) + k \times \left(\sum_{e_j \in MinSet} d_{jv} \right)$$
$$= k \times MinCost + k \times MinCost. \tag{7}$$

Therefore, for $MinCost$, there is:

$$\frac{1}{2} \times 2k \times MinCost = k \times MinCost. \tag{8}$$

According to equations (5), (6), (7), and (8), the approximate factor of our solution $MinSet$ is calculated as:

$$\sigma = \frac{MinCost}{MinCost^*} \leq \frac{k \times MinCost}{\frac{k}{2} \times MinCost} = 2. \tag{9}$$

Therefore, the approximate algorithm for the optimal executor allocation problem is a 2-approximate algorithm.

4.3. Greedy Algorithm

When the triangular inequality cannot be guaranteed in a data center, such as the heterogeneous network of nodes, we propose a greedy algorithm for the optimal executor allocation problem.

Algorithm 2 uses the distance *threshold* to select the executors. To minimize the total communication distance, firstly, the algorithm calculates the maximum and minimum distance between executors of E, and assigns the *threshold* to the minimum distance. Secondly, it finds all executor pairs in E whose network distance is equal to *threshold*, and puts them in $E^{'}$. Thirdly, the algorithm expands the executor set $E^{'}$ by searching executor $e_v \in U$ that satisfies the distance between e_v and any executor of $E^{'}$ is not greater than *threshold*. This process is repeated until e_v does not exist or the number of executors of $E^{'}$ equals k. Finally, if the number of executors of $E^{'}$ equals k, return $E^{'}$; Otherwise, increases *threshold* and cycles the above steps.

Because $k \leq m$, as long as the $threshold$ is set reasonably, it can always return an executor set of size k. The time complexity of Algorithm 2 is $O(k \times m^3)$, where k is the number of executors required.

The correctness of Algorithm 2 is proved as follows:

Algorithm 2: Greedy Algorithm

Input:
The set of executors allowed to start: E;
The network distance matrix: D;
The number of executors required: k;
Output:
The executors selected to start.

begin
  calculate the maximum and minimum distance between executors of E: $Max(D)$ and $Min(D)$;
  $Threshold = Min(D)$;
  while $Threshold \leq Max(D)$; **do**
    initialize $U = E$; $E^{'} = \varnothing$;
    repeat
      find executor pair $e_u, e_v \in E$, such that $d_{uv} == Threshold$;
      $E^{'} = E^{'} \cup \{e_u, e_v\}$;
      $U = U - \{e_u, e_v\}$;
    until e_u, e_v does not exist;
    repeat
      find $e_v \in U$ such that for each $e_u \in E^{'}$: $d_{uv} \leq Threshold$;
      $E^{'} = E^{'} \cup e_v$;
      $U = U - e_v$;
    until e_v does not exist or $|E^{'}| = k$;
    if $|E^{'}| = k$ **then**
      go to 21;
    end
    else
      $Threshold++$;
    end
  end
  return $E^{'}$.
end

Proof. Algorithm 2 returns an executor set $E^{'}$ with a $Threshold$. This means that no more executor set can be found so that the size is at least k and the threshold is less than $Threshold$. Assume that the executor set E^{*} has a $Threshold^{*}$ such that $\sum_{u,v \in E^{*}} d_{uv} < \sum_{u,v \in E^{'}} d_{uv}$ and $Threshold^{*} < Threshold$. For the executor set $E^{'}$, it starts from the minimum distance and gradually expands the executor set according to the selected executors. Because $Min(D) < Threshold^{*} < Threshold$, when $Threshold$ is $Min(D)$, the set cannot be expanded to a value size of k, Algorithm 2 will continue to increase the $threshold$ and expand the executor set; when the $threshold$ is $Threshold^{*}$, the executor set will be expanded to a size equal to k, and exit. At this time $Threshold = Threshold^{*}$, which contradicts the assumption.

5. Experimental Evaluation

In this section, we evaluate the performance of our proposed algorithms. The executor allocation strategies: *approximate algorithm* and *greedy algorithm* are implemented by modifying the function *scheduleExecutorsOnWorkers* of Spark-3.0.1 source codes. Hence the system can use our achievement when allocating the executors on idle nodes.

5.1. Experiment Setup

The experiments are carried out in a data center that contains 9 servers organized by 3 racks. These racks contain 2, 3, and 4 servers respectively, and each server has one Intel(R) Xeon(R) CPU E5-2678, 64GB RAM and 512GB Disk. The KVM virtualization technology is used to build medium-sized virtual machines, every VM is equipped with 4 virtual cores, 8GB RAM and 64GB disk space.

For software configuration, we use Java JDK-1.8, Apache Hadoop-2.7.3 and Apache Spark-3.0.1, and the deployment mode is YARN. Unless otherwise stated, all configurations are set by default. In particular, the block size of HDFS is set to 128MB and the replication factor is 3.

For the purpose of evaluating the performance, as shown in Table 3, two micro-benchmarks (join and sort) and two macro-benchmarks (pageRank and LDA (Latent Dirichlet Allocation)) are chosen for testing. These benchmarks are characterized by distinct workload types and representative in big data processing.

Table 3. Benchmark and workload type

Benchmark	Workload type
Sort	Map and Reduce-Input Heavy Job
Join	Reduce-Input Heavy Job
PageRank	Iterative Application
LDA	Iterative Application

We first compare the performance of *approximate algorithm* and *greedy algorithm* with *spreadOut* and *noSpreadOut* [3] provided by Spark, and then compare them with other recent executor allocation methods: *iSpark* [33] and *Warm-up Manager* [20]. *iSpark* aims to timely scale up or scale down the number of executors in order to fully utilize the allocated resources, and *Warm-up Manager* aims to reduce the initialization overhead and to enable latency-sensitive applications to apply dynamic strategies. For fairness, the following performance indicators are used:

Job execution time: the time from the start to the end of a job. Because the executor allocation can also affect the data locality in the map stage, thereby influencing the execution time of the job. Therefore, this indicator can comprehensively reflect the overall performance of relevant algorithms.

Reduce stage execution time: the time from the reduce task obtains intermediate data to the end of the task. Since there is a large amount of data communication in the reduce stage, the impact of different executor allocation algorithms on job performance can be directly observed through the execution time of the stage.

5.2. Performance

Satisfy the triangular inequality. We first deploy the Spark cluster on the data center with 18 nodes (each server starts 2 VMs). To provide the homogeneous network of nodes,

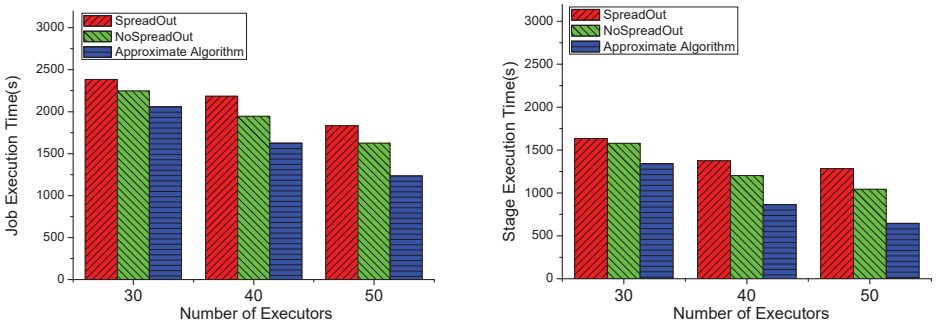
the bandwidth inside and outside the racks is set 10Gbps, so that the network distance matrix of executors satisfies the triangular inequality. Then we estimate the performance of *approximate algorithm*.

(1) Micro-benchmark

Sort is a popular application with the function of making data objects in order. The experiment uses 30GB data set of the *Wikipedia Corpus* [6]. This application contains a job with two stages: map stage and reduce stage, each stage has 240 tasks. To evaluate the performance under different numbers of executors, the number of executors required is set to 30, 40, and 50 respectively in the procedure.

Fig. 4(a) shows the performance comparison of the three executor allocation methods (*spreadOut*, *noSpreadOut* and *approximate algorithm*). It illustrates that *approximate algorithm* has a lower job execution time than other two methods. Meanwhile, as the number of executors increases, the job execution time is shorter due to the increase in the parallelism of tasks.

We further observe the performance comparison in the reduce stage, as shown in Fig. 4(b). In this stage, the reducer takes a lot of time to obtain the intermediate data from previous tasks. Since the reduce stage is considered in our optimization of data locality through executor allocation, it can be seen that *approximate algorithm* has a obvious reduction in the execution time. In particular, when the number of executors required is 40, comparing with *spreadOut* and *noSpreadOut*, *approximate algorithm* reduces the execution time by 37.1% and 28.2% respectively.



(a) Job execution time

(b) Reduce stage execution time

Fig. 4. Performance comparison under Sort.

Furthermore, Fig 5 displays the empirical CDF of these reduce tasks when the number of executors required is 40. We can see that the execution time of most tasks is between 100 and 200 seconds under *approximate algorithm*, which is better than the performance under *spreadOut* and *noSpreadOut*. Specifically, the average execution time of reduce tasks for *spreadOut*, *noSpreadOut*, and *approximate algorithm* is $229s$, $200s$, and $144s$, respectively.

To explore the reasons for performance improvement, we analyze the data locality of reduce tasks, which is divided into three levels: local access data, cross-node traffic, and cross-rack traffic. Table 4 shows the network traffic of reduce tasks during the stage

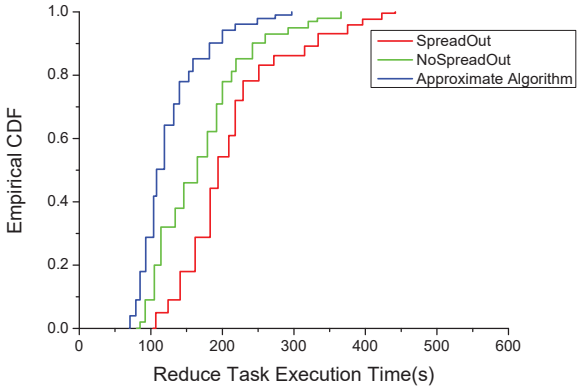


Fig. 5. The empirical CDF of reduce tasks.

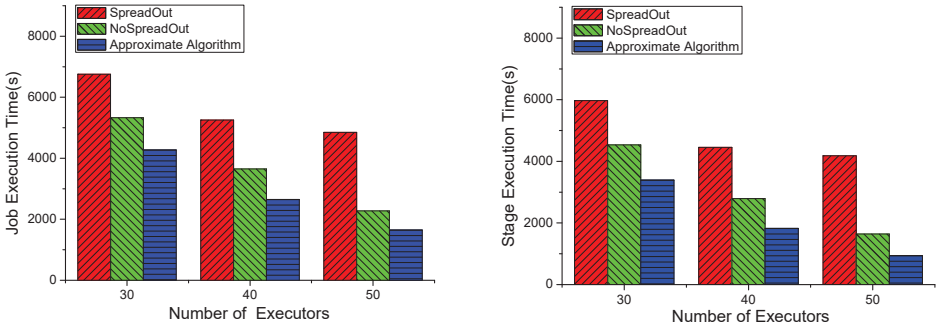
execution. In general, the locality level of *approximate algorithm* is much better than *spreadOut* and *noSpreadOut*, with more local access data and less cross-node/rack traffic. This is because *approximate algorithm* starts executors on the nodes close to each other, providing reduce tasks with the possibility of better locality in task scheduling. In contrast, *spreadOut* and *noSpreadOut* do not fully consider the data locality factor, leading to the reduce tasks to traverse longer network distances to obtain data, so the overhead of data communication is relatively high.

Table 4. Network Traffic of Reduce Tasks

Locality Level	spreadOut	noSpreadOut	approximate algorithm
Local access data	23.5%	39.4%	57.7%
Cross-node traffic	45.9%	42.5%	33.2%
Cross-rack traffic	30.6%	18.1%	9.1%

Join is a widely used operation in data query. The application utilizes the left-outer-join that connects a large data set to a small data set (i.e., 2GB$\times$512MB) from *Ratings and classification data* [4]. The same as before, we set the number of executors required to 30, 40, and 50 respectively in the procedure.

Fig. 6 shows the performance comparison of relevant methods. Fig. 6(a) explains that when the number of executors required is 50, by comparison with *spreadOut* and *noSpreadOut*, *approximate algorithm* reduces the job execution time by 66.0% and 27.5% separately. It observes that compared with the performance under the sort benchmark, the performance of *approximate algorithm* under the join benchmark is more significant. This is because join generates a larger amount of intermediate data, which leads to much more data communication for the reduce tasks. It in turn makes the effect of optimizing the data locality by executor allocation more prominent. Fig. 6(b) further illustrates that *approximate algorithm* outperforms others. In particular, when the required number of executors



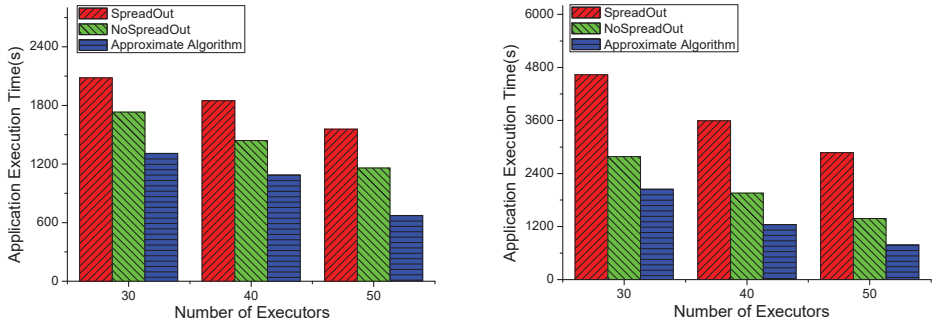
(a) Job execution time (b) Reduce stage execution time

Fig. 6. Performance comparison under Join.

is 30, compared with *spreadOut* and *noSpreadOut*, *approximate algorithm* decreases the reduce stage execution time by 43.2% and 25.2%, respectively.

(2) Macro-benchmark

To evaluate the performance under more complex applications, we select two popular machine learning algorithms pageRank and LDA from the Spark examples for testing. Since these two applications contain one or more jobs, in which every job usually contains a lot of stages, the application execution time is used for evaluation.



(a) PageRank (b) LDA

Fig. 7. Performance comparison under macro-benchmark.

PageRank is a widely recognized iterative algorithm for ranking web pages according to their importance. The experiment uses 10GB data set of the *WT10g* [7], and set the parameter *numIterations* to 10 in the procedure. The application execution consists of 1 job and 13 stages.

From the experimental result of Fig. 7(a), it can be seen that compared with *spreadOut* and *noSpreadOut*, *approximate algorithm* has the shortest application execution time. In

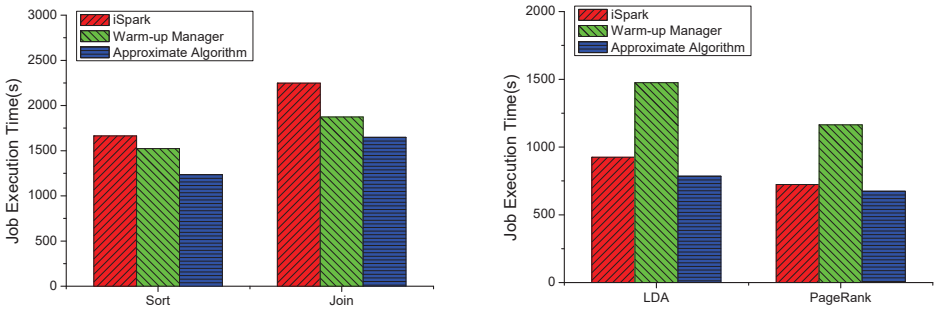
particular, when the number of executors required is 40, *approximate algorithm* reduces the application time by 41.2% and 24.6% respectively.

LDA is a document generation model in natural language processing, which identifies the hidden subjects in a large-scale documents. The experiment runs on 20GB *arXiv Bulk Data data set* [5] and the procedure sets the parameter $maxIterations$ to 20. This application is concretely executed as 26 jobs and 90 stages totally.

The experimental results illustrate that *approximate algorithm* has a greater performance advantage than other two methods, as shown in Fig. 7(b). In particular, when the number of executors required is 50, *approximate algorithm* decreases the application time by 72.7% and 43.2% compared with *spreadOut* and *noSpreadOut*, respectively. As we can see for the application with many jobs and stages, such as pageRank and LDA, optimizing the data locality by executor allocation in multiple reduce stages can bring a substantial performance benefit.

(3) Performance comparison with other methods

We also compare the performance of *approximate algorithm* with other two recent executor allocation methods: *iSpark* and *Warm-up Manager*. Fig. 8(a) shows the experimental results under the micro-benchmark: sort and join when the number of executors required is set to 50. We can see that *approximate algorithm* decreases more job execution time than *iSpark* and *Warm-up Manager*. In particular, under the join benchmark, the execution time of *approximate algorithm* is reduced by 26.7% and 12.1%, respectively. Meanwhile, Fig. 8(b) also explains that *approximate algorithm* has better performance than *iSpark* and *Warm-up Manager*, and *iSpark* outperforms *Warm-up Manager* under the macro-benchmark: LDA and pageRank which are iterative applications.



(a) Under micro-benchmark (b) Under macro-benchmark

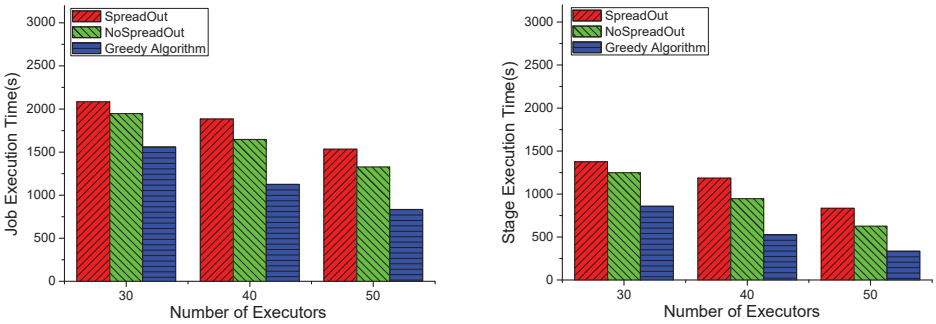
Fig. 8. Performance comparison with other methods.

Not satisfy the triangular inequality. We replace the bandwidth 10Gbps of half inside and outside racks with 20Gbps based on the original data center, so that the network distance matrix of executors does not satisfy the triangular inequality.

(1) Micro-benchmark

Fig. 9 shows the performance comparison of *spreadOut*, *noSpreadOut* and *greedy algorithm* under the sort benchmark. It can be seen that *greedy algorithm* has a shorter

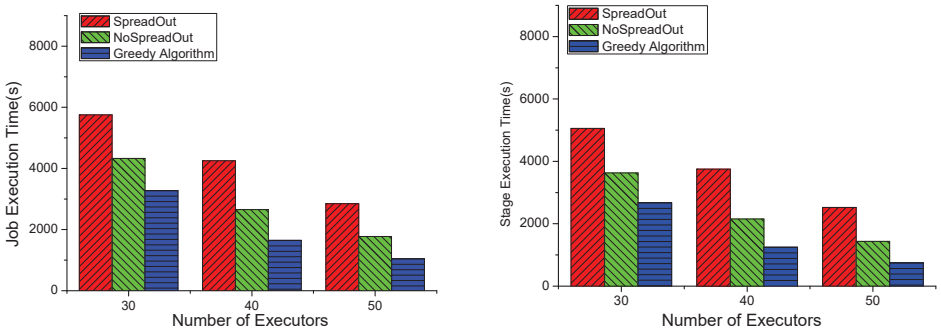
job running time than other methods. In particular, when the number of executors required is 30, *greedy algorithm* shortens the job execution time by 25.1% and 20.0% compared with *spreadOut* and *noSpreadOut*, respectively. Fig. 9(b) shows that when the number of executors required is 50, the execution time of reduce stage reduced by *greedy algorithm* is 59.7% and 46.3% respectively.



(a) Job execution time (b) Reduce stage execution time

Fig. 9. Performance comparison under Sort.

Fig. 10 depicts the performance results under the join benchmark. Fig. 10(a) shows that when the number of executors required is 40, *greedy algorithm* decreases the job running time by 61.4% and 37.9% over *spreadOut* and *noSpreadOut*, respectively. Fig. 10(b) further verifies the performance advantage of *greedy algorithm*: when the number of executors required is 50, it decreases the reduce stage execution time by 70.4% and 47.9%, respectively.



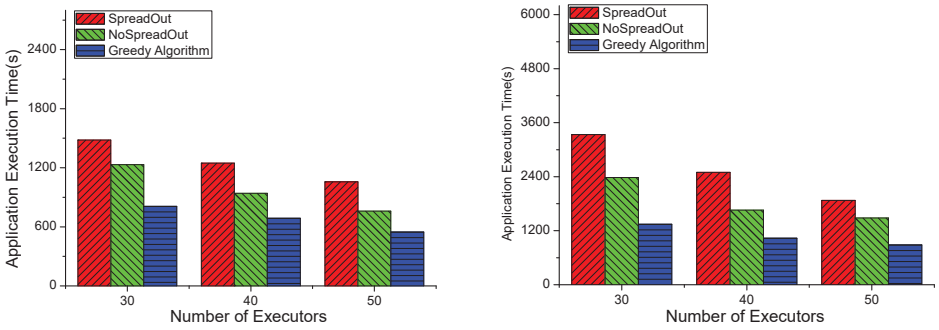
(a) Job execution time (b) Reduce stage execution time

Fig. 10. Performance comparison under Join.

(2) Macro-benchmark

The performance comparison under pageRank is exhibited in Fig. 11(a). It can be seen that compared with *spreadOut* and *noSpreadOut*, *greedy algorithm* has the shortest application execution time. In particular, when the number of executors required is 30, *greedy algorithm* decreases the application time by 45.4% and 34.3% compared with *spreadOut* and *noSpreadOut*, respectively.

Fig. 11(b) shows the experimental results under the LDA benchmark. It illustrates that *greedy algorithm* can run the LDA application faster than *spreadOut* and *noSpreadOut*. In particular, when the number of executors required is 50, *greedy algorithm* decreases the application execution time by 70.4% and 47.9%, respectively.



(a) PageRank (b) LDA

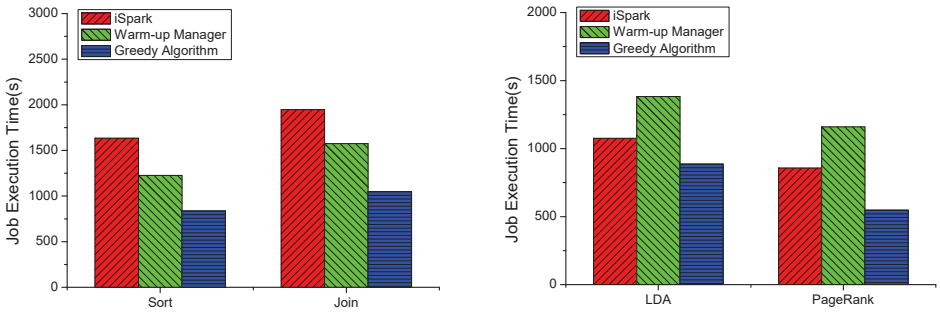
Fig. 11. Performance comparison under macro-benchmark.

(3) Performance comparison with other methods

When the number of executors required is set to 50, the performance comparison of *greedy algorithm* with *iSpark* and *Warm-up Manager* is shown in Fig. 12. The experimental results illustrate that under the micro-benchmark and macro-benchmark, *greedy algorithm* always has the shortest job execution time. In particular, under the pageRank benchmark, *greedy algorithm* reduces the execution time by 36.2% and 52.8% compared with *iSpark* and *Warm-up Manager*, respectively.

5.3. Time Overhead of Algorithm

During the above experiment process, we recorded the average time required to start a set of executors for an application, the results are shown in Table 5. Comparing with *spreadOut* and *noSpreadOut*, because *approximate algorithm* and *greedy algorithm* needs additional computation in order to select a subset from the executors that are allowed to start on each node, they will take more time to launch the executors. This has a negative impact on the performance of our proposed algorithms, especially when the number of executors required is large. In addition, the time complexity of *approximate algorithm* and *greedy algorithm* are $O(m^2)$ and $O(k \times m^3)$ respectively, where k is the number of executors required, and m is the number of executors allowed to start, so *greed algorithm* is slower than *approximate algorithm* for the executor start time. However, in contrast



(a) Under micro-benchmark (b) Under macro-benchmark

Fig. 12. Performance comparison with other methods.

with the application/job execution time, the time overhead of *approximate algorithm* and *greedy algorithm* only takes up only a small part, so it can be ignored.

Table 5. Average Executor Start Time

Number of Executors	30	40	50
spreadOut	43.7ms	49.4ms	54.6ms
noSpreadOut	27.2ms	29.5ms	32.8ms
approximate algorithm	1.25s	2.67s	3.38s
greedy algorithm	2.36s	3.73s	4.75s

6. Conclusion

This paper has attempted to optimize the data locality by executor allocation for the reduce stage in Spark computing environment. We first calculate the distance matrix of executors and formulate the optimal executor allocation problem to minimize the total communication distance. This problem is proved to be an NP-Hard problem. Then, for the cases where the network distance between executors satisfies and does not satisfy the triangular inequality, an approximate algorithm and a greedy algorithm are proposed respectively. Finally, we conduct extensive experiments and the results show that our algorithms can optimize the data locality for reduce tasks and improve the application/job performance. In general, for different workload types, the proposed algorithms can bring more performance gain to the reduce-input heavy jobs and iterative applications than the map and reduce-input heavy jobs.

Acknowledgments. The work is supported by Scientific Research Projects funded by Hunan Provincial Department of Education (22B0451), and Doctoral Research Startup Foundation of University of South China (No.200XQD083).

References

1. ”Apache Spark”,https://spark.apache.org/
2. ”Apache Hadoop”,https://hadoop.apache.org/
3. ”Apache Spark”,https://spark.apache.org/docs/3.3.0/job-scheduling.html/
4. ”Ratings and classification data.”,http://webscope.sandbox.yahoo.com
5. arxivbulkdata. https://arxiv.org/help/bulk_data_s3/
6. Wikipedia corpus. https://www.english-corpora.org/wiki/
7. Wt10g. http://ir.dcs.gla.ac.uk/test_collections/
8. Arslan, E., Shekhar, M., Kosar, T.: Locality and network-aware reduce task scheduling for data-intensive applications. In: Tang, W., Zhao, Y., Zheng, Z. (eds.) Proceedings of the 5th International Workshop on Data-Intensive Computing in the Clouds, DataCloud '14, New Orleans, Louisiana, USA, November 16-21, 2014. pp. 17–24. IEEE (2014)
9. Beaumont, O., Lambert, T., Marchal, L., Thomas, B.: Performance analysis and optimality results for data-locality aware tasks scheduling with replicated inputs. Future Gener. Comput. Syst. 111, 582–598 (2020)
10. Cheng, L., Wang, Y., Liu, Q., Epema, D.H.J., Liu, C., Mao, Y., Murphy, J.: Network-aware locality scheduling for distributed data operators in data centers. IEEE Trans. Parallel Distributed Syst. 32(6), 1494–1510 (2021)
11. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (Jan 2008)
12. Fu, Z., He, M., Tang, Z., Zhang, Y.: Optimizing data locality by executor allocation in reduce stage for spark framework. In: Shen, H., Sang, Y., Zhang, Y., Xiao, N., Arabnia, H.R., Fox, G., Gupta, A., Malek, M. (eds.) Parallel and Distributed Computing, Applications and Technologies. pp. 349–357. Springer International Publishing, Cham (2022)
13. Fu, Z., Tang, Z., Yang, L., Liu, C.: An optimal locality-aware task scheduling algorithm based on bipartite graph modelling for spark applications. IEEE Trans. Parallel Distributed Syst. 31(10), 2406–2420 (2020)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
15. Gu, H., Li, X., Lu, Z.: Scheduling spark tasks with data skew and deadline constraints. IEEE Access 9, 2793–2804 (2021)
16. Guo, Z., Fox, G., Zhou, M.: Investigation of data locality in mapreduce. IEEE/ACM International Symposium on Cluster Cloud & Grid Computing pp. 419–426 (2012)
17. Lee, S., Jo, J., Kim, Y.: Survey of data locality in apache hadoop. In: Iwashita, M., Shimoda, A., Chertchom, P. (eds.) 2019 IEEE International Conference on Big Data, Cloud Computing, Data Science & Engineering, BCD 2019, Honolulu, HI, USA, May 29-31, 2019. pp. 46–53. IEEE (2019)
18. Lu, X., Islam, N.S., Wasi-ur-Rahman, M., Jose, J., Subramoni, H., Wang, H., Panda, D.K.: High-performance design of hadoop RPC with RDMA over infiniband. In: 42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013. pp. 641–650. IEEE Computer Society (2013)
19. Ma, X., Fan, X., Liu, J., Li, D.: Dependency-aware data locality for mapreduce. IEEE Trans. Cloud Comput. 6(3), 667–679 (2018)
20. Morisawa, Y., Suzuki, M., Kitahara, T.: Flexible executor allocation without latency increase for stream processing in apache spark. In: 2020 IEEE International Conference on Big Data (Big Data). pp. 2198–2206 (2020)
21. Naik, N.S., Negi, A., Bapu, B.R.T., Anitha, R.: A data locality based scheduler to enhance mapreduce performance in heterogeneous environments. Future Gener. Comput. Syst. 90, 423–434 (2019)

22. Neciu, L., Pop, F., Apostol, E.S., Truica, C.: Efficient real-time earliest deadline first based scheduling for apache spark. In: Potolea, R., Iancu, B., Slavescu, R.R. (eds.) 20th International Symposium on Parallel and Distributed Computing, ISPDC 2021, Cluj-Napoca, Romania, July 28-30, 2021. pp. 97–104. IEEE (2021)
23. Shabeera, T.P., Kumar, S.D.M.: A novel approach for improving data locality of mapreduce applications in cloud environment through intelligent data placement. Int. J. Serv. Technol. Manag. 26(4), 323–340 (2020)
24. Shabeera, T.P., Kumar, S.D.M., Chandran, P.: Curtailing job completion time in mapreduce clouds through improved virtual machine allocation. Comput. Electr. Eng. 58, 190–202 (2017)
25. Shang, F., Chen, X., Yan, C.: A strategy for scheduling reduce task based on intermediate data locality of the mapreduce. Clust. Comput. 20(4), 2821–2831 (2017)
26. Sun, M., Hang, Z., Zhou, X., Lu, K., Li, C.: Hpso: Prefetching based scheduling to improve data locality for mapreduce clusters. In: Conference on Design (2014)
27. Tan, J., Meng, S., Meng, X., Zhang, L.: Improving reducetask data locality for sequential mapreduce jobs. In: 2013 Proceedings IEEE INFOCOM. pp. 1627–1635 (2013)
28. Tang, S., He, B., Yu, C., Li, Y., Li, K.: A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications. IEEE Trans. Knowl. Data Eng. 34(1), 71–91 (2022)
29. Tang, Z., Ma, W., Li, K., Li, K.: A data skew oriented reduce placement algorithm based on sampling. IEEE Trans. Cloud Comput. 8(4), 1149–1161 (2020)
30. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., Baldeschwieler, E.: Apache hadoop YARN: yet another resource negotiator. In: Lohman, G.M. (ed.) ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013. pp. 5:1–5:16. ACM (2013)
31. Xia, T., Wang, L., Geng, Z.: A reduce task scheduler for mapreduce with minimum transmission cost based on sampling evaluation. International Journal of Database Theory & Application (2015)
32. Yan, X., Wong, B., Choy, S.: R3S: rdma-based RDD remote storage for spark. In: Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware, ARM@Middleware 2016, Trento, Italy, December 12-16, 2016. pp. 4:1–4:6. ACM (2016)
33. Yang, D., Rang, W., Cheng, D., Wang, Y., Tian, J., Tao, D.: Elastic executor provisioning for iterative workloads on apache spark. In: 2019 IEEE International Conference on Big Data (Big Data). pp. 413–422 (2019)
34. Zaharia, M., Borthakur, D., Sarma, J.S., Elmeleegy, K., Shenker, S., Stoica, I.: Delay scheduling:a simple technique for achieving locality and fairness in cluster scheduling. In: European Conference on Computer Systems. pp. 265–278 (2010)
35. Zhang, X., Luo, F., Jia, Z., Shen, J.: Prefetching method for hadoop mapreduce environments. Xi'an Dianzi Keji Daxue Xuebao/Journal of Xidian University 41(2), 191–196 (2014)

Zhongming Fu received the PhD degree at the College of Computer Science and Electronic Engineering, Hunan University, China, in 2020. He is currently a lecturer of School of Computer Science, University of South China. His research interests include the parallel computing, the improvement and optimization of task scheduling module in Hadoop and Spark platforms. He has published several papers in IEEE TCC and TPDS.

Mengsi He received the master degree at the College of Computer Science and Electronic Engineering, Hunan University, China, in 2020. She is currently a technician of School of Computer Science, University of South China. Her current research interests include

parallel computing, the improvement and optimization of the graph computation module in Spark platforms.

Zhuo Tang received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2008. He is currently a professor with the College of Computer Science and Electronic Engineering, Hunan University. He is also the chief engineer with the National Supercomputing Center, Changsha. He has authored or coauthored almost 90 journal articles and book chapters. His research interests include distributed computing system, cloud computing, and parallel processing for big data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He is a member of the IEEE/ACM and CCF.

Yang Zhang received the Ph.D. degree in software engineering from the National University of Defense Technology (NUDT), China, in 2019. He is currently an Assistant Professor of the PDL laboratory, NUDT. His research interests include empirical software engineering, social network analysis, and DevOps.

Received: January 31, 2022; Accepted: December 10, 2022.