Tsinghua Science and Technology

Volume 23 | Issue 6 Article 10

2018

An Improved Algorithm for Optimizing MapReduce Based on Locality and Overlapping

Jianjiang Li
the Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, China.

Jie Wang
the Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, China.

Bin Lyu
University of Southern California, Los Angeles, CA 90089, USA.

Jie Wu
the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA.

Xiaolei Yang
the Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, China.

TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 10/10 pp 744–753
DOI: 10.26599 / TST.2018.9010115
Volume 23, Number 6, December 2018

An Improved Algorithm for Optimizing MapReduce Based on Locality and Overlapping

Jianjiang Li, Jie Wang, Bin Lyu*, Jie Wu, and Xiaolei Yang

Abstract: MapReduce is currently the most popular programming model for big data processing, and Hadoop is a well-known MapReduce implementation platform. However, Hadoop jobs suffer from imbalanced workloads during the reduce phase and inefficiently utilize the available computing and network resources. In some cases, these problems lead to serious performance degradation in MapReduce jobs. To resolve these problems, in this paper, we propose two algorithms, the Locality-Based Balanced Schedule (LBBS) and Overlapping-Based Resource Utilization (OBRU), that optimize the Locality-Enhanced Load Balance (LELB) and the Map, Local reduce, Shuffle, and final Reduce (MLSR) phases. The LBBS collects partition information from input data during the map phase and generates balanced schedule plans for the reduce phase. OBRU is responsible for using computing and network resources efficiently by overlapping the local reduce, shuffle, and final reduce phases. Experimental results show that the LBBS and OBRU algorithms yield significant improvements in load balancing. When LBBS and OBRU are applied, job performance increases by 15% from that of models using LELB and MLSR.

Key words: MapReduce; overlapping; load balance; data locality

1 Introduction

Hadoop[1], the open-source software framework for the distributed storage and processing of big data sets, is widely applied. Using MapReduce[2] and the Hadoop Distributed File System (HDFS), Hadoop utilizes computer clusters built by commodity hardware and provides a robust environment for various applications. Hadoop also features automatic parallelization, load balancing, and disaster management[3], as well as a simple and friendly interface for programmers and developers.

Typically, there is one NameNode and many DataNodes in a Hadoop cluster. The NameNode keeps track of monitoring tasks, runs DataNodes, and maintains the metadata of files stored on the HDFS. DataNodes are responsible for running assigned map and reduce tasks and for providing the NameNode with feedback about task progress and storage information about temporary files.

A job is usually submitted by a client to the resource manager. When informed of the submitted job's input file locations, the NameNode assigns map tasks to the DataNodes based on the proximity of the file split. During map tasks, input splits are processed by the specified mapper and converted into intermediate key-value pairs. The output of map tasks is stored locally on the DataNodes, and related information is sent to the NameNode. During the shuffle phase, each DataNode is assigned reduce tasks for some partitions of the map output. To process a

- Jianjiang Li, Jie Wang, and Xiaolei Yang are with the Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, China. E-mail: lijianjiang@ustb.edu.cn; wangjieblingbling@126.com; chinayangxiaolei@163.com.
- Bin Lyu was with the Department of Computer Science and Technology, University of Science and Technology Beijing, Beijing 100083, China when he did this research and now he is with University of Southern California, Los Angeles, CA 90089, USA. E-mail: colinlvbin@gmail.com.
- Jie Wu is with the Department of Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu.
- * To whom correspondence should be addressed.
Manuscript received: 2018-07-23; accepted: 2018-08-10

partition, each DataNode retrieves partition files from other DataNodes. The process of requesting and serving files is referred to as "shuffle". After obtaining the required partitions during the shuffle process, each partition is processed by a specified reducer, and the output files are written directly to the HDFS for the final merge.

Despite the wide application of Hadoop, a number of factors have resulted in performance bottlenecks. First, in some ways, the performance of Hadoop jobs has become dependent on the partitioner that divides intermediate key-value pairs into partitions to be executed during the reduce tasks. A bad partitioner or even a bad input can lead to a load imbalance during the reduce phase because some partitions might include many more files than others. Second, most computing resources are idle during the shuffle phase and network resources remain idle during the local and final reduce tasks, which can result in poor parallelization performance and significant delay.

In a previous study[4], researchers proposed two algorithms, Locality-Enhanced Load Balance (LELB) and Map, Local reduce, Shuffle, and final Reduce (MLSR), to cope with the above disadvantages. The LELB algorithm defines two types of data locality in each partition, i.e., node locality and internal locality, and introduces a combined locality that is the product of both. Then, this combined locality serves as a workload measure for scheduling partitions to reduce tasks. In the MLSR algorithm, decisions about whether to apply MLSR to some partitions are made based on the required computation and complexity of the reduce task. Next, some partitions are locally reduced and shuffled, whereas others are shuffled immediately. Our experimental results show that scheduling with LELB and MLSR outperforms Hadoop's traditional scheduling by up to 14.4% (WordCount). The LELB and MLSR algorithms take two perspectives in optimizing the scheduling of Hadoop: load balancing and resource utilization.

Although LELB and MLSR have enhanced Hadoop performance, there is room for improvement. LELB relies on sampling input data for partition information, which can use up resources as it tries to approximate the real distribution of data. In reality, load imbalances continue to occur by the incomplete usage of partition information and poor threshold choice. With respect to MLSR, although some partitions should be locally reduced before being shuffled in light of their computing costs, it may be better to shuffle these partitions immediately if the destination DataNode is waiting for it to perform the final reduce. For better performance, partial parallel execution of the local reduce, shuffle, and final reduce phases is a necessary option. If all the data in some partitions have been collected via shuffle, it is better to perform a final reduce of that partition directly without performing local reduce. Based on knowledge of the Hadoop source code[1], in this study, we developed a distributed computing framework based on MapReduce[5]. Using this framework, we then tested and compared the performance of the improved algorithms.

The remainder of this paper is organized as follows. In Section 2, we briefly describe the LELB and MLSR algorithms, and in Section 3, we present the improved algorithms, Locality-Based Balanced Schedule (LBBS) and Overlapping-Based Resource Utilization (OBRU), and discuss their implementation details. In Section 4, we consider the performance of traditional MapReduce, LELB&MLSR, and LBBS&OBRU. We introduce related work in Section 5. Lastly, in Section 6, we summarize our results.

2 LELB and MLSR

As mentioned in Section 1, two algorithms, LELB and MLSR[4], have been proposed to cope with the load imbalance and idling resource problems.

2.1 LELB algorithm

The LELB algorithm relies on a sampler to collect distribution information from partitions before running MapReduce jobs. Then, partition information is sent as input to the LELB algorithm to generate schedule plans for the reduce tasks. Schedule plans are proposed based on combined localities, which serve as a measure of workload. The input of LELB is a matrix of combined localities. The algorithm repeatedly searches for the maximum combined locality corresponding to a DataNode and partition, assigns a partition to the DataNode, and updates the workload of this DataNode if the workload is acceptable. LELB ends when all partitions have been assigned. The output is a list of partitions to be final reduced for each DataNode.

2.2 MLSR algorithm

The MLSR algorithm deals with the overlapping issue of the local reduce and shuffle phases on each node. After map tasks are completed and schedule plans are established by the LELB algorithm, each DataNode is assigned some partitions to perform the final reduce. Partitions on DataNodes that are not their destinations for final reduce are either local reduced and shuffled or

shuffled immediately in light of their computation costs.

3 LBBS and OBRU Algorithms

Although both LELB and MLSR have demonstrated good performance experimentally, there remains room for improvement. In this paper, we propose the LBBS and OBRU algorithms as enhanced versions of LELB and MLSR, respectively.

3.1 LBBS algorithm

As mentioned in Section 1, LELB is highly dependent on sampling before the execution of map tasks. However, sufficient sampling costs too much, whereas inadequate sampling results in inefficient schedule plans. We made our first improvement with respect to the method for collecting partition information. Partition information can be dynamically collected while map tasks are running, as in ishuffle[6]. Practically speaking, in Hadoop, by inspecting "Spill Events" during map tasks, partition distribution information can be collected in the DataNodes and sent to the NameNode as part of HeartBeat[3] information, as shown in Fig. 1.

Upon each iteration in LELB, the maximum combined locality from all DataNodes is selected. However, a DataNode with partitions of average volume will probably never be assigned a final reduce task. Therefore, to ensure that the load is balanced, a threshold is introduced to ensure that the workload on each DataNode is within a certain range. However, this threshold is difficult to determine considering the variety of applications and the varied distributions of the partitions. Too large a threshold will affect the load balance, but too small a threshold may not be applicable. Practically speaking, the threshold value should be small when the job starts. When it is determined that this threshold value cannot produce a balanced scheduling plan, it should be improved dynamically until the generated plan becomes acceptable.

Thus, we propose a more intuitive algorithm, as shown in Algorithm 1. We developed the LBBS algorithm based on Ref. [4] and by optimizing the LELB algorithm. Upon each iteration of the LBBS, it first finds the DataNode with the minimum workload. Next, it selects

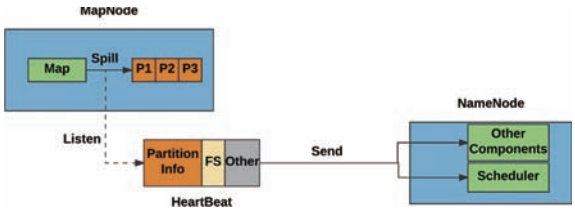

Fig. 1 Sending partition information as part of HeartBeat.

Algorithm 1 LBBS algorithm

Input:
$numPartitions$: Total number of partitions
$numMapNodes$: Total number of nodes performing Map tasks
Output:
1: M=$\{partition_n^p, 1 \leqslant p \leqslant numPartitions\}$;
2: **for all** $1 \leqslant n \leqslant numMapNodes$ **do**
3: $R_n = \varnothing$;
4: **end for**
5: **for all** $1 \leqslant p \leqslant numPartitions$ **do**
6: **for all** $1 \leqslant n \leqslant numMapNodes$ **do**
7: $Locality1_n^p = partition_n^p / \sum_{p=1}^{numPartitions} partition_n^p$;
8: $Locality2_n^p = partition_n^p / \sum_{n=1}^{numMapNodes} partition_n^p$;
9: $Locality_n^p = Locality1_n^p * Locality_n^p$;
10: **end for**
11: **end for**
12: $finished = \varnothing$; $zeroLoad = 0$
13: loadHeap = MakeMinHeap();
14: **for all** $1 \leqslant n \leqslant numMapNodes$ **do**
15: $localityHeap_n$ = MakeMaxHeap();
16: **for all** $1 \leqslant p \leqslant numPartitions$ **do**
17: HeapPush ($localityHeap_n$, $[p,\ Locality_n^p]$);
18: **end for**
19: HeapPush ($loadHeap$, $[n,\ zeroLoad,\ localityHeap_n]$);
20: **end for**
21: **while** $finished \neq M$ **do**
22: $(n, load, localityHeap_n)$ = HeapPop ($loadHeap$);
23: **while** true **do**
24: $(p, Locality_n^p)$ = HeapPop($localityHeap_n$);
25: **if** $p \notin finished$ **then**
26: add p to R_n;
27: add p to finished;
28: break;
29: **end if**
30: **end while**
31: $newLoad = load + \sum_{n=1}^{numMapNodes} partition_n^p$;
32: HeapPush ($loadHeap$, $[n,\ newLoad,\ localityHeap_n]$);
33: **end while**
34: **return** $\{R_n, 1 \leqslant n \leqslant numMapNodes\}$;

the maximum locality on that DataNode. Then, it assigns the partition corresponding to the maximum locality to the DataNode for the final reduce. In this manner, the load is intuitively balanced since the algorithm always considers the DataNode with the minimum workload. Furthermore, a threshold is no longer needed. A min heap and several max heaps can also be used, respectively, to manage the workload of the DataNodes and to partition the information on each. Below, we detail the key step of the LBBS algorithm.

• Lines 1–4: Initialize M as a collection of all partitions, and initialize each node's reduce collection to an empty set.

• Lines 5–11: The internal locality is the locality of a partition relative to other partitions within a node, with $Locality1_n^p$ representing the internal locality of the p-th partition on the n-th node; the node locality is a partition on a node relative to all partial partitions of this partition case, with $Locality2_n^p$ representing the node locality of the p-th partition on the n-th node. The locality refers to

the comprehensive locality of a partition on a node, taking into account the internal and node localities. $Locality_n^p$ represents the locality of the *p*-th partition on the *n*-th node. Calculate the internal locality, node locality, and the locality based on the partition information.

• Line 12: Initialize the complete partition allocation set to an empty set, and zero load.

• Lines 13–20: Create a load min heap and a locality max heap for each node, and push the locality information into the max heap and the node information into the min heap.

• Lines 21–33: When there is still an unallocated partition, first extract information from the node with the least load from the minimum load heap, take the unallocated partition from the node locality maximum heap, assign this partition to this node, update the finished set, and finally re-push the updated node load into the minimum load heap.

• Line 34: Return the distribution plan R set when all partitions are allocated.

3.2 OBRU algorithm

Two costs are computed when determining whether to use MLSR or the traditional MapReduce, but, in reality, it is difficult to estimate the communication and computation costs of the reduce function. Therefore, we arrive at the solution from another perspective: resource utilization. Two types of resources, computation and network, are needed to perform the local reduce, shuffle, and final reduce phases. Shuffle generally requires only network resources, whereas local reduce and final reduce usually require only computation resources. As such, we propose that shuffle, local reduce, and final reduce be parallelized.

In summary, we obtained the OBRU algorithm by optimizing the MLSR algorithm. We divided this algorithm into three phases: local reduce, shuffle, and final reduce, which correspond to Algorithms 2–4, respectively. Suppose R_n represents the collection of partitions to be final reduced on the *n*-th MapNode, which is generated by the LBBS algorithm; LR_n is the collection of partitions to be locally reduced on the *n*-th MapNode; LR_done_n is the collection of partitions that have been locally reduced on the *n*-th DataNode; and *M* is the collection of all the partitions. Suppose $Shuffle\_in_n$ is the collection of partitions that have been fully shuffled into the *n*-th DataNode from all other DataNodes and $Shuffle\_out_n$ is the collection of partitions that have been shuffled out to their destinations for final reduce. Thus, we have $Shuffle\_in_n \subseteq R_n$ and $Shuffle\_out_n \subseteq M - R_n$. For

Algorithm 2 Local reducer algorithm

```
for all p in LR_n do
    if p in R_n then
        if p in Shuffle_in_n then
            continue;
        end if
    else
        if p in Shuffle_out_n then
            continue;
        end if
    end if
    Local Reduce p and generate reduced file
    add p to LR_done_n
end for
```

Algorithm 3 Shuffler algorithm

```
while true do
    if partition p is fetched then
        if all data for p is ready then
            add p to Shuffle_in_n;
        end if
    else
        if p in LR_done_n then
            serve locally reduced file for p;
        else
            serve original file for p;
        end if
        add p to Shuffle_out_n;
    end if
end while
```

Algorithm 4 Final reducer algorithm

```
while true do
    get partition p from Shuffle_in_n;
    if p in LR_done_n then
        use locally reduced file as local file for p;
    else
        use original file as local file for p;
    end if
    final reduce p;
end while
```

the *n*-th DataNode, we describe the local reducer, shuffler, and final reducer, respectively. In these algorithms, $Shuffle\_in_n = \varnothing$, $Shuffle\_out_n = \varnothing$, $LR_n = M$, $LR\_done_n = \varnothing$.

The local reduce, shuffle, and final reduce phases are described above. These three phases are also dynamically overlapped with one another and have several shared objects for task-tracking and synchronization. As much, OBRU differs from MLSR in the following ways.

First, in MLSR, final reduce is described as a fixed phase that executes after the local reduce and shuffle phases. In OBRU, on the other hand, final reduce is initiated by the shuffler when all the data for a partition are retrieved from all the other DataNodes. Second, upon receiving a data request from another DataNode, the shuffler in OBRU decides which data to send back based on whether the data has been processed by the local

reducer. Finally, in OBRU, all partitions are set as the initial input of local reduce. When a partition is selected by the local reducer, the local reducer first checks whether the partition has been shuffled out or final reduced. Only partitions that have not been processed by the shuffler and final reducer are locally reduced.

4 Performance Evaluation and Analysis

Here, we propose two enhanced algorithms, LBBS and OBRU, that address load balancing and resource utilization, respectively. In our experiment, we analyzed the performances of our LBBS and OBRU algorithms and compared them with those of LELB and MLSR in the simulation environment. By experimenting with different DataNode numbers and different input data scales, we were able to analyze the approaches from different perspectives, focusing specifically on load balancing and overall performance. Table 1 shows the test environment.

The logic of our experiments is shown in Algorithm 5. The input scale is 1.0x, which is $512\ \mathrm{KB} \times 20$, 1.5$x$ is 1.5 times the input size of 1.0x, and so on. For example, in the first round, we set the input scale to 1.0x and the DataNode number to 2, and we ran the WordCount application five times using the traditional MapReduce.

The record of MapReduce jobs includes the execution time of each DataNode in each of the map, shuffle, and reduce phases along with the total execution time. We define the DataNode time as the total time consumed by the map, local reduce, shuffle, and final reduce phases of each DataNode. We then prepared job records for further analysis by computing the standard deviations of the map executions and DataNode times.

4.1 Load balancing

To measure the load balancing of different scheduling algorithms, in our experiment, we first defined the standard deviation of the map execution time using each group of parameters as the standard value, and assigned similar map workloads to the DataNodes. Then, we compared the standard value with the schedule execution times using different scheduling algorithms. Table 2 shows the results of our comparisons when we set the number of DataNodes to eight. In Table 2, STDEV (DataNode time) refers to the mean square error of the DataNode running time, and STDEV (Map time) refers to the mean square error of the running time during the map phase, which is the standard value.

As shown in Table 2, for each input scale, the standard deviations of the map execution times are similar despite different scheduling algorithms. For each group of input scales, the standard deviation of the DataNode time of LBBS&OBRU is the smallest of the three, and is also closest to the standard deviation of the map execution time. This means that the workload of the DataNode running with LBBS&OBRU is more balanced than those running with LELB&MLSR and the traditional MapReduce.

Next, we grouped the test data by the number of DataNodes with respect to each input scale, the test results of which are shown in Figs. 2–5.

We compared the schedule execution times for the three types of scheduling algorithms. As shown in the figures, the WordCount application using our improved algorithms, LBBS and OBRU, generally exhibited better performance in terms of load balancing. Typically, for example, when the application is running with four DataNodes and a basic input scale, the standard deviations of the schedule execution times for both LBBS and OBRU were 0.6 s, which is close to the standard deviation of the map execution time, 0.58 s. Running with eight DataNodes and a 2.0x input scale, the standard deviations of the

Table 1 Experimental environment for NameNode and DataNode.

Type	Memory (MB)	OS	Storage (GB)
NameNode	512	CentOS 7, 64 bit	25
DataNode	8192	CentOS 7, 64 bit	100

Algorithm 5 Logic of experiment

for $input_{scale}$ in [1.0x, 1.5x, 2.0x, 4.0x] **do**
  for $DataNode_{number}$ in [2, 4, 8, 16] **do**
    for plan in [Traditional, LELB & MLSR, LBBS & OBRU] **do**
      experiment ($[input_{scale}, DataNode_{number}]$)
      5 times;

Table 2 Job information with eight DataNodes.

Schedule	Input scale	STDEV (DataNode time) (s)	STDEV (Map time) (s)
$Traditional$	1.0x	1.694	0.1554
$LELB\&MLSR$	1.0x	0.927	0.2184
$LBBS\&OBRU$	1.0x	0.491	0.1674
$Traditional$	1.5x	2.573	0.2240
$LELB\&MLSR$	1.5x	0.947	0.2560
$LBBS\&OBRU$	1.5x	0.889	0.2094
$Traditional$	2.0x	2.816	0.4864
$LELB\&MLSR$	2.0x	1.750	0.4742
$LBBS\&OBRU$	2.0x	1.326	0.4868
$Traditional$	4.0x	6.815	0.6638
$LELB\&MLSR$	4.0x	3.440	0.7528
$LBBS\&OBRU$	4.0x	2.583	0.6746

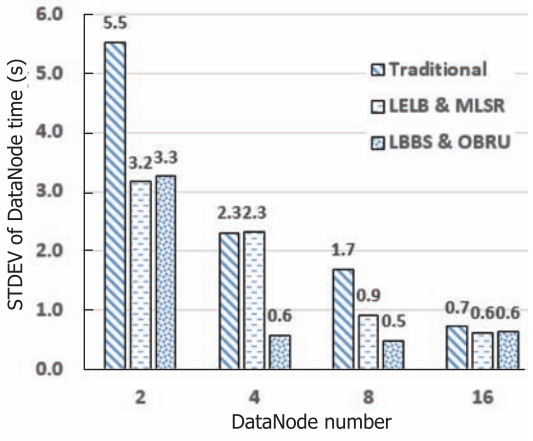

Fig. 2 STDEV group by DataNode number (base input scale.

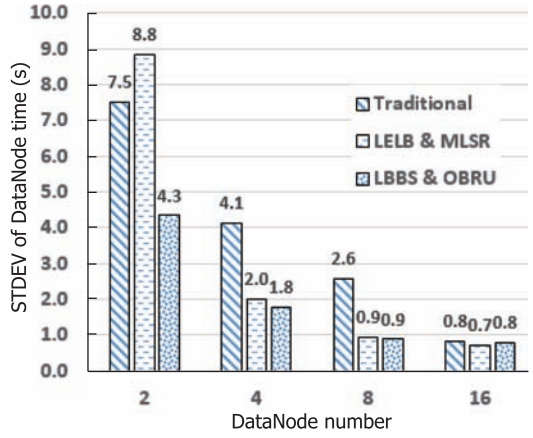

Fig. 3 STDEV group by DataNode number (1.5*x* input scale).

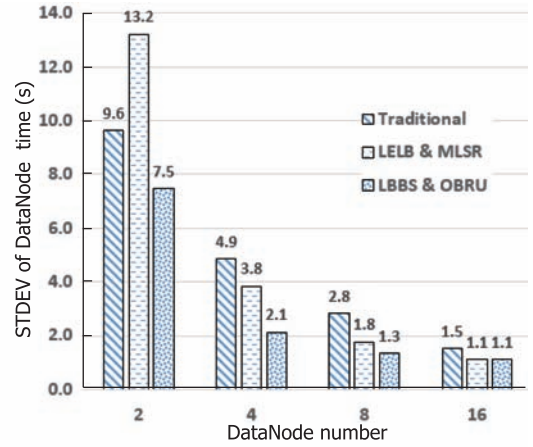

Fig. 4 STDEV group by DataNode number (2.0*x* input scale).

DataNode times of both LBBS and OBRU were 1.3 s, 28% less than that of LELB and MLSR and 54% less than that of the traditional MapReduce.

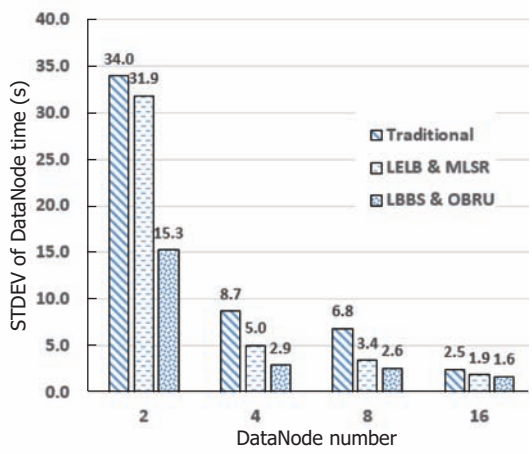

Fig. 5 STDEV group by DataNode number (4.0*x* input scale).

On the whole, LELB and MLSR exhibit better load balancing than the traditional MapReduce, but exceptions arise in the case of running two DataNodes with 1.5*x* and 2.0*x* input scales. In Figs. 3 and 4, we can see that the traditional MapReduce exhibits better load balancing than LELB and MLSR and is closer to the load balancing of LBBS and OBRU. This is due to the default scheme used to distribute partitions in the traditional MapReduce. When there are only two DataNodes, applying this scheme will produce a random schedule plan, which, in this case, is better than that generated by LELB and MLSR. For LELB and MLSR, the threshold value is the key to generating a balanced plan. We set the threshold value to be 10% of the average workload, and applied this value throughout the experiment. However, in some cases, this threshold can be too small to generate a plan, as in the case of using just two DataNodes. For these exceptions, the threshold must be set as 15% of the average workload, which can result in a less balanced scheduling plan. Using LBBS, the threshold is abandoned and the plan is thus naturally more balanced.

As more DataNodes are applied, the standard deviation of the schedule execution time decreases. When the input scale is too small to consume computing and network resources, there is little difference between the performances of the different scheduling algorithms. When sixteen DataNodes are used with each input scale, the standard deviations of the schedule execution times are almost the same.

4.2 Job performance

The job performance for each algorithm is reflected by the job time recorded on the NameNode side. Recording starts when the WordCount job is submitted and ends when the

job is marked as complete on the NameNode. Job time includes the times for the map, shuffle, and reduce phases as well as for the necessary network communications. Since the time for the map phase is almost the same for same input scale, the differences in job time actually reflect differences in the schedule execution times. In our analysis, we first grouped the test data by the DataNode number. Then, we compared the job times for each algorithm with respect to each input scale, as shown in Figs. 6–9.

In general, we can see in the figures that job time increases with increases in the input scale and decreases as more DataNodes are utilized. However, when sixteen DataNodes are used, the job time for 1.5x is almost the same as, or even less than, the job time for 1.0x. This is because the input scale is relatively small, too small to effectively consume the computing and network resources of sixteen DataNodes.

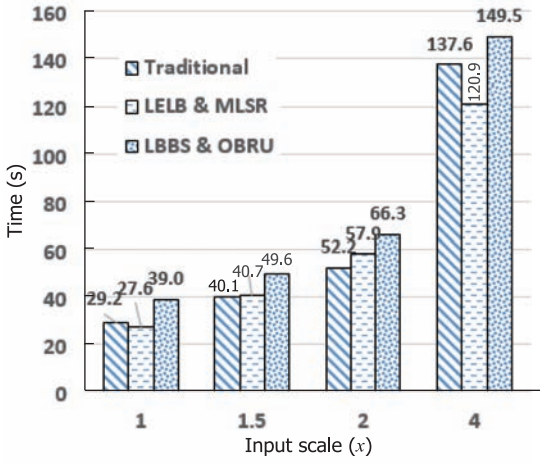

Fig. 6 Job time when two DataNodes are used.

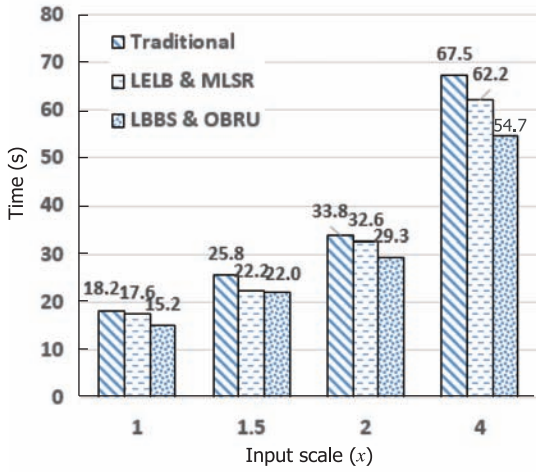

Fig. 7 Job time when four DataNodes are used.

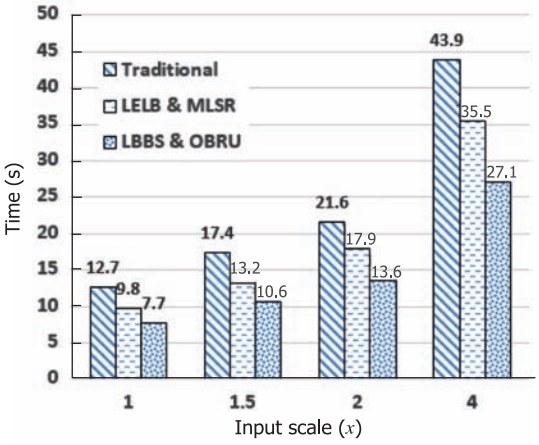

Fig. 8 Job time when eight DataNodes are used.

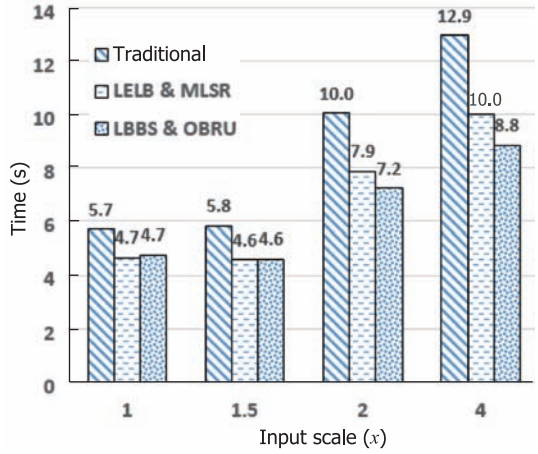

Fig. 9 Job time when sixteen DataNodes are used.

The job time for eight DataNodes clearly reflects the performances of the traditional MapReduce, LELB&MLSR, and LBBS&OBRU. As the input scale increases, applications running with the LBBS and OBRU algorithms take less time than those running with LELB and MLSR, or traditional MapReduce. Using LELB and MLSR is more efficient than using the traditional MapReduce. Specifically, the job time of applications running with LBBS and OBRU is 15% less than those running with LELB and MLSR, and 17% less than those running with the traditional MapReduce. As can be expected, when the input scale increases, performance improvement is more dramatic by using LBBS and OBRU.

Exceptions to this conclusion occur when only two DataNodes are used. In Fig. 6, the traditional MapReduce has the best performance of the three algorithms, and LBBS and OBRU do not perform as well as expected. Considering the implementation of the traditional MapReduce and the other two algorithms, we conclude that the local reduce step in MLSR and

OBRU uses a significant portion of job time. Moreover, the high expense of the scheduling and synchronization associated with LELB&MLSR and LBBS&OBRU is also a major factor. As the experiment proves, when the DataNode number is small, performing local reduce is not as necessary, and the traditional MapReduce with direct shuffle and final reduce is more effective.

In conclusion, when the input scale is large enough and a corresponding number of DataNodes are utilized in the cluster, it is more efficient to run WordCount-like applications with LBBS and OBRU than with either LELB&MLSR or the traditional MapReduce. In Hadoop, hundreds of machines are deployed in a cluster with terabytes of input data. This data could be processed more efficiently by using enhancements like those of the LBBS and OBRU algorithms.

5 Related Work

As discussed in Section 1, the traditional MapReduce has some major drawbacks that lead to an imbalanced load and performance degradation. A lot of research has been conducted to enhance Hadoop performance, most of which targets load balancing and overlapping. Other enhancement methods also have practical merits.

A number of studies have focused on reducing the data skew of MapReduce jobs. One comprehensive study[7] detailed the reasons for and outcomes of data skew during the map and reduce phases. This study also reported the best practices for alleviating data skew and suggested directions for future work. To mitigate skew problems, Kwon et al.[8] introduced a user-defined cost function to partition input data and optimize data distribution. Kwon et al.[9] proposed a framework that repartitions long tasks to make better use of cluster resources and alleviate data skew problems.

Some recent work focused on overlapping different phases of MapReduce to enhance job performance. Li et al.[4] targeted the load balancing and resource utilization of MapReduce jobs (also the basis for this paper). They defined a combined locality that serves as a measure of workload for establishing schedule plans for reduce tasks. The shuffle and local reduce tasks are then overlapped to enhance job performance. Guo et al.[6] proposed iShuffle, a framework that overlaps the map and shuffle phases by predicting the distribution of partitions. MapReduce jobs using iShuffle perform well in a multi-task environment. In Ref. [10], the authors overlapped the communications of the reduce and shuffle phases, but found that the reduce phase may start prematurely, which results in an incomplete reduce task and introduces extra overheads.

A lot of research has focused on improving task scheduling algorithms. In Refs. [11, 12], the authors improved MapReduce job performance in heterogeneous environments. Tian et al.[11] proposed a scheduler that made scheduling decisions dynamically, whereas Zaharia et al.[12] made use of data locality during the map phase and emphasized the fairness of generated plans. Tao et al.[13] proposed a simple load feedback-based resource scheduling scheme that balances workload and performance even under heavily loaded cloud systems. Wolf et al.[14] proposed a scheduling algorithm that also emphasized fairness and proposes different metrics for optimizing the performance of MapReduce jobs.

Other than Ref. [4], many researchers have proposed optimizations on the basis of analyzing the data locality. In Ref. [15], the authors conducted a comprehensive investigation of the data locality of Hadoop jobs and analyzed its impact on job performance. The authors in Ref. [16] optimized MapReduce jobs based on data locality.

In addition to the studies above, there have been many other perspectives presented for optimizing MapReduce. The authors of Refs. [17–19] made improvements to the HDFS. In Ref. [17], the authors analyzed the limitations of HDFS operations and platform potentials that arose from delays in scheduling and probability issues. The authors of Refs. [18, 19] concentrated on improving jobs that operate small files. In Ref. [18], the authors identified a solution to the small files problem based on the merits of the Hadoop archives and sequence files. The authors in Ref. [19] proposed a different solution by enhancing the HDFS I/O feature.

Using regression methods, Song et al.[20] evaluated the performance of Hadoop jobs using a job analyzer and a prediction module. Zhu and Chen[21] presented and compared two mechanisms for failure detection via HeartBeat information, demonstrated the merits of both strategies, and made suggestions for usage under different circumstances. Wang et al.[22] introduced a merge algorithm to avoid data repetition and disk access and also proposed the use of a pipeline with which to overlap the shuffle, merge, and reduce phases.

6 Conclusion

In this paper, we considered the LELB and MLSR algorithms based on the locality and overlapping. We

then proposed enhanced algorithms, LBBS and OBRU, to produce a more balanced workload and better allocate computing and network resources. First, We improved the LELB by collecting partition information during the map phase rather than sampling information before running the job. Then, we optimized for the logic for selecting partitions as well as the data structure of the LELB. With regard to the MLSR, we introduced the OBRU algorithm, which is divided into three phases, local reduce, shuffle, and final reduce, and used it to allocate computing and network resources more effectively. Our experimental results confirm that LBBS and OBRU perform better in terms of both load balancing and job execution time. Specifically, when running with the LBBS and OBRU algorithms using eight DataNodes, WordCount applications consume at least 15% less than they do running with LELB and MLSR.

Acknowledgment

This work was supported by the National Key R&D Program of China (Nos. 2017YFB0202104 and 2017YFB0202003).

References

[1] H. A. Hadoop, http://hadoop.apache.org, 2009.

[2] J. Dean and S. Ghemawat, MapReduce: Simplified data processing on large clusters, *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] T. White, *Hadoop*: *The Definitive Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc, 2012.

[4] J. J. Li, J. Wu, X. L. Yang, and S. Q. Zhong, Optimizing mapreduce based on locality of k-v pairs and overlap between shuffle and local reduce, presented at the 44$^{\text{th}}$ Int. Conf. on Parallel Processing, Beijing, China, 2015.

[5] L. Z. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Y. Chen, and D. Chen, G-Hadoop: MapReduce across distributed data centers for data-intensive computing, *Future Generation Computer Systems*, vol. 29, no. 3, pp. 739–750, 2013.

[6] Y. F. Guo, J. Rao, D. Z. Cheng, and X. B. Zhou, Ishuffle: Improving hadoop performance with shuffle-on-write, *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1649–1662, 2017.

[7] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, A study of skew in MapReduce applications, in *5th Open Cirrus Summit*, 2011.

[8] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, Skew-resistant parallel processing of feature-extracting scientific user-defined functions, in *Proc. 1st Int. ACM Symp. on Cloud Computing*, 2010, pp. 75–86.

[9] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, SkewTune in action: Mitigating skew in MapReduce applications, *Proceedings of the VLDB Endowment Hompage Archive*, vol. 5, no. 12, pp. 1934–1937, 2012.

[10] F. Ahmad, S. Lee, M. Thottethodi, and T. N. Vijaykumar, MapReduce with Communication Overlap (MaRCO), *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 608–620, 2013.

[11] C. Tian, H. J. Zhou, Y. Q. He, and L. Zha, A dynamic mapreduce scheduler for heterogeneous workloads, presented at the 2009 Eighth Int. Conf. on Grid and Cooperative Computing, Lanzhou, China, 2009.

[12] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, Improving MapReduce performance in heterogeneous environments, in *Proc. 8th USENIX Conf. Operating Systems Design and Implementation*, San Diego, CA, USA, 2008, pp. 29–42.

[13] D. Tao, Z. W. Lin, and B. X. Wang, Load feedback-based resource scheduling and dynamic migration-based data locality for virtual hadoop clusters in openstack-based clouds, *Tsinghua Science and Technology*, vol. 22, no. 2, pp. 149–159, 2017.

[14] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K. L. Wu, and A. Balmin, Flex: A slot allocation scheduling optimizer for mapreduce workloads, in *Proc. ACM/IFIP/USENIX 11th Int. Conf. Middleware*, Bangalore, India, 2010, pp. 1–20.

[15] Z. H. Guo, G. Fox, and M. Zhou, Investigation of data locality in mapreduce, in *Proc. 12th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing*, Ottawa, ON, Canada, 2012, pp. 419–426.

[16] Y. R. Zhao, W. P. Wang, D. Meng, X. F. Yang, S. B. Zhang, J. Li, and G. Guan, A data locality optimization algorithm for large-scale data processing in Hadoop, presented at the 2012 IEEE Symp. on Computers and Communications (ISCC), Cappadocia, Turkey, 2012.

[17] J. Shafer, S. Rixner, and A. L. Cox, The hadoop distributed filesystem: Balancing portability and performance, presented at the 2010 IEEE Int. Symp. on Performance Analysis of Systems & Software, White Plains, NY, USA, 2010.

[18] N. Mohandas and S. M. Thampi, Improving Hadoop performance in handling small files, in *Advances in Computing and Communications*, A. Abraham, J. L. Mauri, J. F. Buford, J. Suzuki, S. M.Thampi, eds. Springer, Berlin, Heidelberg, 2011.

[19] J. Liu, B. Li, and M. N. Song, THE optimization of HDFS based on small files, presented at the 2010 3$^{\text{rd}}$ IEEE Int. Conf. on Broadband Network and Multimedia Technology (IC-BNMT), Beijing, China, 2010.

[20] G. Song, Z. D. Meng, F. Huet, F. Magoules, L. Yu, and X. L. Lin, A Hadoop Mapreduce performance prediction method, presented at the 2013 IEEE 10$^{\text{th}}$ Int. Conf. on High Performance Computing and Communications & 2013 IEEE Int. Conf. on Embedded and Ubiquitous

Computing, Zhangjiajie, China, 2013.

[21] H. Zhu and H. P. Chen, Adaptive failure detection via heartbeat under Hadoop, presented at the 2011 IEEE Asia-Pacific Services Computing Conf., Jeju Island, R. Korea, 2012.

[22] Y. D. Wang, X. Y. Que, W. K. Yu, D. Goldenberg, and D. Sehgal, Hadoop acceleration through network levitated merge, in *Proc. 2011 Int. Conf. High PERFORMANCE Computing*, *Networking*, *Storage and Analysis*, New York, NY, USA, 2011.

Jianjiang Li is currently an associate professor at University of Science and Technology Beijing, China. He received the PhD degree in computer science and technology from Tsinghua University in 2005. He was a visiting scholar at Temple University from Jan. 2014 to Jan. 2015. His current research interests include parallel computing, cloud computing, and parallel compilation.

Jie Wang is currently a master student in University of Science and Technology Beijing, China. She received the BS degree from Tangshan Normal University in 2015. Her current research interests include cloud computing and parallel computing.

Bin Lyu is currently a master student in University of Southern California, USA. He received the BS degree from University of Science and Technology Beijing in 2017. His current research interests include cloud computing and parallel computing.

Jie Wu is the Associate Vice Provost for International Affairs at Temple University and IEEE Fellow. He also serves as Director of Center for Networked Computing and Laura H. Carnell professor. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. He received the PhD degree from Florida Atlantic University in 1989.

Xiaolei Yang received the master degree from University of Science and Technology Beijing, China, in 2016. His current research interests include parallel computing, cloud computing, and recommender systems.