Optimizing MapReduce based on Locality of K-V Pairs and Overlap between Shuffle and Local Reduce

Jianjiang Li*, Jie Wu†, Xiaolei Yang‡, and Shiqi Zhong§
*‡§*Department of Computer Science and Technology, University of Science and Technology Beijing, China*
†*Department of Computer and Information Sciences, Temple University, USA*
lijianjiang@ustb.edu.cn;†jiewu@temple.edu;‡ chinayangxiaolei@163.com; §zhongshiqi1991@163.com

***Abstract*—At present, MapReduce is the most popular programming model for big data processing. As a typical open source implementation of MapReduce, hadoop is divided into map, shuffle, and reduce. In the mapping phase, according to the principle moving computation towards data, the load is basically balanced and network traffic is relatively small. However, shuffle is likely to result in the outburst of network communication. At the same time, reduce without considering data skew will lead to an imbalanced load, and then performance degradation. This paper proposes a Locality-Enhanced Load Balance (LELB) algorithm, and then extends the execution flow of MapReduce to Map, Local reduce, Shuffle and final Reduce (MLSR), and proposes a corresponding MLSR algorithm. Use of the novel algorithms can share the computation of reduce and overlap with shuffle in order to take full advantage of CPU and I/O resources. The actual test results demonstrate that the execution performance using the LELB algorithm and the MLSR algorithm outperforms the execution performance using hadoop by up to 9.2% (for Merge Sort) and 14.4% (for WordCount).**

***Keywords*-MapReduce; hadoop; locality; overlap; cloud computing**

I. Introduction

Currently, as an open source implementation of Google's cloud computing system, hadoop[1][2] has been applied in big data analysis and processing by many famous companies such as eBay, Amazon, Google, Facebook, Adobe, Yahoo, and so on. MapReduce[3][4] is one of the most important parts of hadoop and Google's cloud computing system. It has been used in many fields, because it has the following advantages: (1) It can be used to deal with big data, and can hide many tedious details such as automatic parallelization, load balancing, and disaster management, which will greatly reduce the development effort of the programmer. (2) Its scalability is so good that it can support tens of thousands of MapReduce nodes.

The normal hadoop jobs will be divided into 3 phases: map, shuffle, and reduce. During the mapping phase, the data set of the input are processed by distributed tasks in the clusters of computers. Then the MapReduce framework writes the mapping output locally at each machine and aggregates the relevant records at each Reducer by remotely reading from the Mappers. This process of transferring data is called the Shuffling phase. All reduce tasks must start after the shuffle part of the map phase has been completed. Therefore, it is obviously that the implementation of MapReduce in hadoop still has plenty of shortcomings, as follows. Firstly, the mapping output will be directly shuffled without considering the locality of k-v pairs. At the same time, each reducer uses a FIFO queue to deal with records instead of k-v pairs [5]. Secondly, the granularities of Map and Reduce are coarse, which is inefficient for overlapping [6]. Finally, the concentrated outbreak of communication in the Shuffling phase [6] will result in obvious communication competition and latency.

Based on the above disadvantages, the main factors which result in performance degradation of MapReduce are communication cost in the shuffling phase and load imbalance in the reduction phase. Therefore, there are currently several kinds of methods proposed, as follows. Firstly, we consider balancing the load in the reduction phase: Based on the key distribution of intermediate pairs, [7–9] can schedule the MapReduce workload. Secondly, we consider reducing the communication costs in the shuffling phase: Through overlapping mapping and shuffling, [10] hides communication costs in the shuffling phase. Through repartitioning the input datasets and optimizing the distribution of keys in the mapping phase, [11] increases the data locality in the reduction phase. [5] develops a method to break the barrier in MapReduce in a way that improves efficiency, and [6] proposes MapReduce with communication overlap to improve its execution performance. By using locality, [7, 8] can decrease the amount of communication in the shuffling phase.

Recognizing the reason leading to the performance decrease, we develop a novel algorithm, Locality-Enhanced Load Balance (LELB) algorithm, which considers the ratio of the *k*th key on the *n*th node while considering the ratio of the *k*th key on the all nodes. The algorithm takes into account which key has the best 'internal locality', as well as which node has the better 'node locality'; this is done in order to choose the main key whose locality is the maximum. Therefore, we can organize which keys are to be executed on which nodes in the most efficient way possible. Besides, we extend the steps of MapReduce to include the Map, Local reduce, Shuffle, and final Reduce (MLSR). In the MLSR,

local reduce tasks can be executed concurrently so that the computation amount of the final reduction task is reduced. The MLSR overlaps the computation of local reduce and shuffle to target better use of the CPU and I/O resources. At the same time, a decentralized rather than concentrated outbreak of shuffle relieves the pressure on the network transmission. To quantify the performance of the LELB algorithm and the MLSR algorithm, we conduct a comprehensive study which applies them to the MapReduce implementation of Merge Sort and WordCount. Our experiment results demonstrate that the execution performance using the LELB algorithm and the MLSR algorithm outperforms the execution performance using traditional hadoop by up to 9.2% (for Merge Sort) and 14.4% (for WordCount).

The remainder of this paper is organized as follows. Section 2 introduces related work. Section 3 describes the basic components of the LELB algorithm and the MLSR algorithm in detail. The performance evaluation and analysis of the LELB algorithm and the MLSR algorithm are discussed in Section 4. Finally, we make a summary in Section 5.

II. Related Work

According to the discussion in Section 1, we could see that MapReduce still has some shortcomings, and one of its major drawbacks is that its execution performance is not as good as people anticipated. The main factors which result in performance degradation of MapReduce are communication cost in the shuffling phase, and load imbalance in the reduction phase. In our paper, we refer to such imbalanced situations as map-skew and reduce-skew, respectively. There are many situations that result in the skew. [12] has described lots of reasons for skew, and gives examples to avoid some situations. Additionally, there are many studies on decreasing skew and load imbalance.

An investigation [13] is given, which demonstrates the goodness of scheduling multiple tasks simultaneously, based on fine data locality, and the authors also study how the different factors influence the performance of data locality. M. Zaharia et al. designed a new robust scheduler for a homogeneous environment called LATE to deal with selecting speculative tasks by estimating time left instead of progress rate, and achieved significant improvements on hadoop response times [14, 15]. [16] shows that coupling the map phase and reduce phase appropriately can also improve the computing performance, thus it comes up with a new model based on MapReduce's basic scheduling characteristics.

III. LELB and MLSR

The problem we study and try to solve in this paper is that in the case of giving a distribution of keys, how these keys are scheduled in order to minimize the execution time of all of MapReduce. Fig. 1 shows a simplified example of the distribution of keys. We could use it to illustrate the key problems. Here, there are 3 keys A, B and C on 3 nodes. Suppose that the sum of the workload for each key on 3 nodes is equal, that is $sum\_of\_workload(A) = sum\_of\_workload(B) = sum\_of\_workload(C)$. So, the key issues in this paper could be converted into: Which node will keys A, B and C be executed on? Which job will be executed first on every node? For each node, will it send/receive data using a network source, or will it reduce using the computing source?

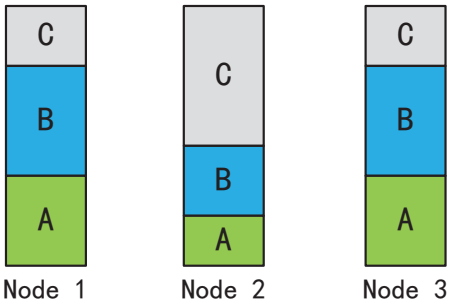

Figure 1. The distribution of keys (A simplified example).

This paper will follow four steps to solve these issues above.

(1) Based on the locality of keys, we can determine the order in which keys will be executed. (2) According to the rule of data movement, we can judge whether the key on other nodes needs to be immediately transferred to this node for reducing, or it must be reduced after locally reducing and moving this key to this node. If some keys arrived at this node, the number of the keys on this node needs to be modified.

(3) When the total workload of some node exceeds the average load, this node cannot execute a new workload.

(4) The next round of workload distribution will start from the node with minimal workload. We will introduce important parts, respectively, in detail.

A. Locality-Enhanced Load Balance (LELB) Algorithm

This step chooses the main keys to execute on each node in order to decrease the communication during the shuffling phase. By sampling the input data to Map nodes, we can get the key distribution on each Map node. Suppose key_n^k is the *k*th key on the *n*th Map node and $1 \leq k \leq numKeys$, $1 \leq n \leq numMapNodes$, then we define:

(1) $Locality1_n^k = key_n^k / \sum_{k=1}^{numKeys} key_n^k$ is the proportion of the *k*th key on the *n*th Map node.

(2) $Locality2_n^k = key_n^k / \sum_{n=1}^{numMapNodes} key_n^k$ is the proportion of the *k*th key on all the Map nodes. Obviously, $Locality1_n^k$ shows the key's internal locality of one Map node, $Locality2_n^k$ shows the key's locality among all the Map nodes.

As shown in Fig. 2, the length of the key indicates the proportion of the key: the longer the length is, the

Figure 2. The distribution of keys on 3 Map nodes.

bigger the proportion is. From this figure, we know: key 4 has the highest proportion of Map node 1, key 5 has the highest proportion of Map node 2, and key 3 has the highest proportion of Map node 3. Compared to Map node 1 and Map node 3, Map node 2 has the highest proportion of key1. So, according to the above definitions, on Map node 1, key 4 has the best internal locality ($Locality1_1^4$). On Map node 2, key 5 has the best internal locality ($Locality1_2^5$). On Map node 3, key 3 has the best internal locality ($Locality1_3^3$). Based on key1, compared to Map node 1 and Map node 3, Map node 2 has a better node locality ($Locality2_2^1$).

To a large extent, we can choose the *k*th key's maximum $Locality2_n^k$ to determine which node the *k*th key should be located on in order to achieve the minimum shuffle time. However, if we take the load balance of the reduce period into consideration, for example, when the load on one Map node is greater than its average load and beyond the threshold, then we possibly move this key to the other Map node, which has the second $Locality2_n^k$ for the reduce period. Based on this principle, we can finally locate all the keys to the Map nodes. Similarly, we can use $Locality1_n^k$ to determine the key's execution order on the same Map node during the reduce period. The key with the biggest $Locality1_n^k$ will be the first executed reduce task. Therefore, based on $Locality1_n^k$ and $Locality2_n^k$, we define:

$$
\begin{aligned}
Locality_n^k &= Locality1_n^k * Locality2_n^k \\
&= (key_n^k / \sum_{k=1}^{numKeys} key_n^k) * (key_n^k / \sum_{n=1}^{numMapNodes} key_n^k) \\
&= (key_n^k)^2 / (\sum_{k=1}^{numKeys} key_n^k * \sum_{n=1}^{numMapNodes} key_n^k)
\end{aligned}
$$

The above definitions of locality take into account both the internal node locality and locality between all the nodes, so we call this method Locality-Enhanced. Here we give our Locality-Enhanced Load Balance algorithm. The key steps in algorithm 1 are as follows:

- Line 1: The numMapNodes sets of potential reducers to schedule (R_n, where $1 \le n \le numMapNodes$) are initialized, that is $R_n = \emptyset$.
- Lines 2-4: By computing $Locality1_n^k$ and $Locality2_n^k$, the locality enhanced of the *k*th key on the *n*th Map node, which is named as $Locality_n^k$ can be obtained. Here, $1 \le k \le numKeys$ and $1 \le n \le numMapNodes$. $Locality_n^k$ takes into account both the internal node locality and the locality between all the nodes.
- Lines 5-6: The average load of the Map nodes is computed. The real load of every Map node is set to zero.
- Line 7: To search the maximum value of $Locality_n^k$ and record the corresponding *mk* (the sequence number of keys) and *mn* (the sequence number of Map nodes).
- Line 8: The load of the *mk*th key will be added to the load of the *mn*th Map node.
- Lines 9-12: If the absolute value of the difference between real load and average load of the *mn*th Map node is less than the predefined threshold value, then

Algorithm 1 LELB Algorithm

Input:

key^k: the *k*th key
key_n^k: the number of *k*th keys on the *n*th Map node,
Where,$1 \le k \le numKeys$,
$1 \le n \le numMapNodes$
$numKeys$: the number of keys
$numMapNodes$: the number of Map nodes
M=$\{key^k, 1 \le k \le numKeys\}$
LTV: the threshold value

Output:

load balance scheduling scheme during reduce phase

1: initialize *numMapNodes* sets of potential reducers to schedule,$R_n = \emptyset, 1 \le n \le numMapNodes$
2: **for all** $1 \le k \le numKeys$ **do**
3: **for all** $1 \le n \le numMapNodes$ **do**
4: $Locality1_n^k \Leftarrow key_n^k / \sum_{k=1}^{numKeys} key_n^k$
$Locality2_n^k \Leftarrow key_n^k / \sum_{n=1}^{numMapNodes} key_n^k$
$Locality_n^k \Leftarrow Locality1_n^k * Locality2_n^k$
5: averageload $\Leftarrow \sum_{n=1}^{numMapNodes} \sum_{k=1}^{numKeys} key_n^k$ /*numMapNodes*
6: $Load_n \Leftarrow 0, 1 \le n \le numMapNodes$
7: calculate maximum-value
maxLocality=max$\{Locality_n^k , key^k \in M\}$
$mk \Leftarrow k$ *and* $mn \Leftarrow n$
8: $Load_{mn} \Leftarrow Load_{mn} + \sum_{n=1}^{numMapNodes} key_n^{mk}$
9: **if** $|Load_{mn}$ - averageload $| \le LTV$ **then**
10: add key^{mk} to R_{mn}, key^{mk} will be executed reduce task on the *mn*th Map node; delete key^{mk} from *M*
11: **else**
12: $Load_{mn} \Leftarrow Load_{mn} - \sum_{n=1}^{numMapNodes} key_n^{mk}$
13: delete $Locality_{mn}^{mk}$ from $\{Locality_n^m, 1 \le k \le numKeys, 1 \le n \le numMapNodes\}$
14: **if** *M* is not empty **then**
15: go to Line 7
16: **else**
17: return $R_n, 1 \le n \le numMapNodes$

key^{mk} will be added to the set of R_{mn}. The *mk*th key will be executed to reduce the task on the *mn*th Map node, meaning that other Map node's data of the same key will be sent to this Map node, and key^{mk} will be deleted from the set of *M*.

- Line 13: $Locality_{mn}^{mk}$ is deleted from $\{Locality_n^k, 1 \leq k \leq numKeys, 1 \leq n \leq numMapNodes\}$
- Lines 14-17: If the set of *M* is not empty, it will be executed repeatedly. Otherwise, return the $R_n, 1 \leq n \leq numMapNodes$ and the LELB algorithm finishes.

Below, we will briefly explain the key selection of the LELB algorithm under some different situations.

(1) Suppose that the sum of every key on all nodes is the same, so it can be treated as a constant number and Locality can be redefined as:

$$Locality_n^k = (key_n^k)^2 / \sum_{k=1}^{numKeys} key_n^k$$

We choose the main key whose Locality is maximized to execute on some nodes.

For example, the keys distribution is as follows. There are 3 nodes. Node1 {key1:50, key2:100, key3:50, key4:40, key5:60, key6:80} means that there are 50 key1, 100 key2, 50 key3, 40 key4, 60 key5 and 80 key6 on node1. The situation of node2 and node 3 are in the Tables I. Using the LELB algorithm, we choose key1 and key4 to execute on node3, key5 and key3 to execute on node2, and key2 and key6 to execute on node1. Table II show the locality of different keys on every node in this situation. Additionally, the sequence numbers (①②③④⑤⑥) show the order of the scheme according to the LELB algorithm.

(2) Suppose that the number of keys on every node is almost the same, so it can be treated as a constant number and Locality can be redefined as:

$$Locality_n^k = (key_n^k)^2 / \sum_{n=1}^{numMapNodes} key_n^k$$

We choose the main key whose Locality is maximized to execute on some node.

For example, the keys distribution is as follows. There are 3 nodes. Tables III show the number of different keys on every node in this situation. Using the LELB algorithm, we choose key2 and key5 to execute on node2, key6 and key3 to execute on node1, and key1 and key4 to execute on node3. Table IV show the locality of different keys on every node in this situation. The sequence numbers (①②③④⑤⑥) show the order of the scheme according to the LELB algorithm.

(3) In general, we choose the main key whose Locality is maximized to execute on some nodes. Here, the Locality is:

$$\begin{aligned} Locality_n^k &= Locality1_n^k * Locality2_n^k \\ &= (key_n^k / \sum_{k=1}^{numKeys} key_n^k) * (key_n^k / \sum_{n=1}^{numMapNodes} key_n^k) \\ &= (key_n^k)^2 / (\sum_{k=1}^{numKeys} key_n^k * \sum_{n=1}^{numMapNodes} key_n^k) \end{aligned}$$

Table I
THE NUMBER OF DIFFERENT KEYS ON EVERY NODE IN SITUATION(1)

	key1	key2	key3	key4	key5	key6
node1	50	100	50	40	60	80
node2	30	80	100	70	130	50
node3	120	20	50	90	10	70

Table II
THE LOCALITY OF DIFFERENT KEYS ON EVERY NODE IN SITUATION(1)

	key1	key2	key3	key4	key5	key6
node1	6.58	26.32③	6.58	4.21	9.47	16.84④
node2	1.96	13.91	21.74⑤	10.65	36.74②	5.43
node3	40.00①	1.11	6.94	22.50⑥	0.28	13.61

Table III
THE NUMBER OF DIFFERENT KEYS ON EVERY NODE IN SITUATION(2)

	key1	key2	key3	key4	key5	key6
node1	50	100	70	40	50	90
node2	20	130	100	60	80	10
node3	90	50	80	80	60	40

Table IV
THE LOCALITY OF DIFFERENT KEYS ON EVERY NODE IN SITUATION(2)

	key1	key2	key3	key4	key5	key6
node1	15.63	35.71	19.60⑤	8.89	13.16	57.86②
node2	2.50	60.36①	40.00	20.00	33.68⑥	0.71
node3	50.63③	8.93	25.60	35.56④	18.95	11.43

B. The Extended Execution Flow of MapReduce

This paper extends the execution flow of MapReduce to Map, Local reduce, Shuffle and final Reduce (MLSR) as shown in Fig. 3. Local reduce can share the computation of reduce and overlap with shuffle in order to take full advantage of CPU and I/O resources, compared with the traditional MapReduce framework. LReduce in Fig. 3 means local reduce and the number of mappings on a machine can be larger than 1. The amount of data does not reduce during the local reducing phase, but the performance is better than when using original MapReduce, according to the experiments in Section 4.

This paper will deduce the circumstances under which the key on other nodes executes local reduce, shuffle, and final reduce, without needing to be immediately transferred to this node (traditional shuffling). Suppose that the time complexity of computation for n keys is $f_c(n)$

Table V
THE RELATIONSHIP BETWEEN TIME COMPLEXITY OF REDUCE AND THE SCOPE OF α

Time Complexity	The upper bound of α	Scope of α
$f_c(n) = O(1)$	0	0
$f_c(n) = O(n)$	$1 - max_{i=1}^{m}\{O(\beta_i)\}$	$[0,\ 1 - 1/m]$
$f_c(n) = O(n^2)$	$1 - max_{i=1}^{m}\{O(\beta_i^2)\}$	$[0,\ 1 - 1/m^2]$
$f_c(n) = O(n^3)$	$1 - max_{i=1}^{m}\{O(\beta_i^3)\}$	$[0,\ 1 - 1/m^3]$
$f_c(n) = O(n^k), k \geq 1$	$1 - max_{i=1}^{m}\{O(\beta_i^k)\}$	$[0, 1 - 1/m^k]$
$f_c(n) = O(logn)$	$-max_{i=1}^{m}\{O(log(\beta_i))\}/O(logN)$	$[0,\ log_N m]$
$f_c(n) = O(nlogn)$	$1 - max_{i=1}^{m}\{O(\beta_i)\} - max_{i=1}^{m}\{O(\beta_i log\beta_i)\}/O(logN)$	$[0,\ 1 - 1/m + log_N m/m]$

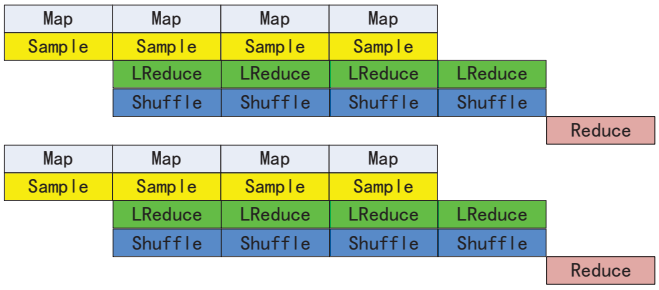

Figure 3. The execution flow of MLSR.

and the time complexity of communication for n keys is $f_T(n)$, then the cost of executing local reduce, shuffle, and final reduce is: $Cost1 = max\{f_c(n_1), ..., f_c(n_m)\} + max\{f_{Ti}(n_1), ..., f_{Ti}(n_m)\} + \alpha f_c(n_1 + ... + n_m)$, where $0 \leq \alpha \leq 1$. α is the ratio of the time of executing 'final reduce' in MLSR and 'reduce' in traditional MapReduce. The cost of executing traditional shuffle and reduce is: $Cost2 = max\{f_{Ti}(n_1), ..., f_{Ti}(n_m)\} + f_c(n_1 + ... + n_m)$.

Suppose that $N = \sum_{i=1}^{m} n_i$, i=1, 2, ..., m, where m is the number of nodes being used to map and N is the total of some key. Let us define $\beta_i = \frac{n_i}{N}$, where $i = 1, 2, ..., m$. Note that $\sum_{i=1}^{m} \beta_i = 1$, $0 \leq \beta_i \leq 1$. Therefore, $Cost2 - Cost1 = (1 - \alpha) f_c(N) - max_{i=1}^{m}\{f_c(\beta_i N)\}$.

If $\alpha \leq 1 - max_{i=1}^{m}\{f_c(\beta_i N)\}/f_c(N)$, then $Cost1$ is smaller than $Cost2$, that is, the scheme of 'local reduce, shuffle, and final reduce' will be applied. Otherwise, the scheme of 'traditional shuffle and reduce' will be applied.

Here, we will analyze how different time complexities of computation for n keys ($f_c(n)$) will impact the scheme of 'local reduce, shuffle and final reduce' and 'traditional shuffle and reduce'. Then, the situation with which the MLSR has better execution performance will be easily seen.

(1) $f_c(n) = O(1)$

$$\begin{aligned}\alpha \leq & 1 - max_{i=1}^{m}\{f_c(\beta_i N)\}/f_c(N)\\ = & 1 - max_{i=1}^{m}\{O(1)\}/O(1)\\ = & 1 - 1 = 0\end{aligned}$$

That is, the scheme of 'traditional shuffle and reduce' will be applied. $Cost2 - Cost1 = (1 - \alpha)O(1) - O(1) = -\alpha O(1)$, so the scheme of 'traditional shuffle and reduce' will be applied.

(2) $f_c(n) = O(n^k)$

$$\begin{aligned}\alpha \leq & 1 - max_{i=1}^{m}\{f_c(\beta_i N)\}/f_c(N)\\ = & 1 - max_{i=1}^{m}\{O(\beta_i^2 N^k)\}/O(N^k)\\ = & 1 - max_{i=1}^{m}\{O(\beta_i^k)\}\end{aligned}$$

So, when $f_c(n) = O(n^k)$, $k \geq 1$, if $\alpha \leq 1 - max_{i=1}^{m}\{O(\beta_i^k)\}$ then the scheme of 'local reduce, shuffle, and final reduce' will be applied. Otherwise, the scheme of 'traditional shuffle and reduce' will be applied.

(3) $f_c(n) = O(logn)$

$$\begin{aligned}\alpha \leq & 1 - (max_{i=1}^{m}\{f_c(\beta_i N))/(f_c(N))\\ = & 1 - max_{i=1}^{m}\{O(log(\beta_i N))\}/O(logN)\\ = & - max_{i=1}^{m}\{O(log(\beta_i))\}/O(logN)\end{aligned}$$

(4) $f_c(n) = O(nlogn)$

$$\begin{aligned}\alpha \leq & 1 - max_{i=1}^{m}\{f_c(\beta_i N))/(f_c(N)\\ = & 1 - max_{i=1}^{m}\{O(\beta_i N log(\beta_i N))\}/O(NlogN)\\ = & 1 - max_{i=1}^{m}\{O(\beta_i)\}\\ & - max_{i=1}^{m}\{O(\beta_i log\beta_i)\}/O(logN)\end{aligned}$$

Table V shows the relationship between time complexity and α.

C. Map, Local reduce, Shuffle, and final Reduce (MLSR) Algorithm

- When the number of nodes being used to map is 1, and the value of α is always zero, this indicates that direct use of traditional shuffle and reduce.
- When $f_c(n) = O(n^k), k \geq 1$, with the increase of m, the value of α tends to be 1, which means the execution performance using the MLSR is nearly always superior to the execution performance using traditional shuffle and reduce.
- When $f_c(n) = O(n)$ and $f_c(n) = O(nlogn)$, the values of their α are similar. That is, the execution

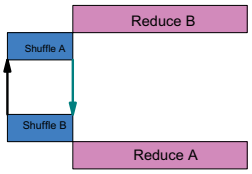

Figure 4. the execution flow of shuffle and reduce in traditional MapReduce (Case 1).

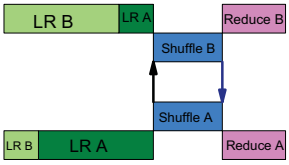

Figure 5. Case 2.

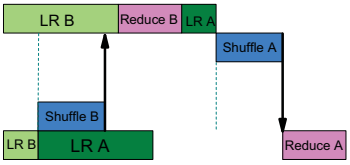

Figure 6. Case 3 & Case 4 (1).

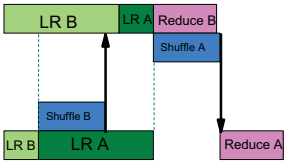

Figure 7. Case 3 & Case 4 (2).

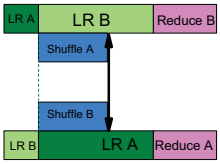

Figure 8. Case 3 & Case 4 (3).

performance using the MLSR is nearly always superior to the execution performance using traditional shuffle and reduce.

- In general, m is smaller than N, so when $f_c(n) = O(logn)$, the value of α is very small, indicating direct use of traditional shuffle and reduce.

Here, we use four different cases to show how the MLSR could make full use of the CPU and I/O resources to have better execution performance.

Suppose that key A and key B are distributed on M1 and M2 (different Map/Reduce nodes) after the mapping phase. There are four cases as follows:

Case 1: If key B on M1 must wait for key B on M2 to reduce on M1, at the same time, key A on M2 must wait for key A on M1 to reduce on M2. Then while moving key A to M2 and moving key B to M1, key B on M1 and key A on M2 cannot be executed. In the shuffling phase, only the I/O resource is used in Case 1.

Case 2: If key B will be executed on M1 and may be locally reduced on M2, at the same time, key A will execute on M2 and may be locally reduced on M1. Then during execution, key A and key B cannot be shuffled. During the shuffle phase, only I/O resources are used in Case 2.

Case 3: If key B will be executed on M1 and may be locally reduced on M2, at the same time, key A on M2 must wait for key A on M1 to reduce on M2, then during moving key A from M1 to M2, key B will be locally reduced on M2. During the shuffle phase, both CPU and I/O resources are used in Case 3.

Case 4: If key A will be executed on M2 and may be locally reduced on M1, at the same time, key B on M1 must wait for key B on M2 to reduce on M1, then while moving key B from M2 to M1, key A will be locally reduced on M1. During the shuffle phase, both CPU and I/O resources are used in Case 4.

Fig. 4 shows the execution flow of shuffle and reduce in traditional MapReduce. Fig. 5, Fig. 6, Fig. 7 and Fig. 8 show the execution flow of Map, Local reduce, Shuffle and final Reduce in extended MapReduce (MLSR). LRs in Fig. 5, Fig. 6, Fig. 7, and Fig.8 mean Local Reduce. Obviously, Fig. 8 has the best execution performance. Algorithm 2 describes the Map, Local reduce, Shuffle, and final Reduce (MLSR) algorithm, which has the best execution performance as shown in Fig. 8.

Here we give the MLSR algorithm. The key steps are explained as follows:

- Lines 1-6: If the *k*th key does not belong to R_n and the Cost(*Local Reduce, Shuffle, and final Reduce*) of key^k

Algorithm 2 Map, Local reduce, Shuffle, and final Reduce (MLSR) Algorithm

Input:

key^k: the *k*th key
$numKeys$: the number of keys
$numMapNodes$: the number of Map nodes
R_n, $1 \leq n \leq numMapNodes$: load balance scheduling scheme during the reduce phase generated by the LELB algorithm

Output:

scheduling scheme generated by the MLSR algorithm

1: **for all** $1 \leq n \leq numMapNodes$ **do**
2: **for all** $1 \leq k \leq numKeys$ **do**
3: **if** $key^k \notin R_n$ **then**
4: **if** $Cost(Local\ reduce,\ Shuffle,\ and\ final\ Reduce)$ of key^k is less than $Cost(traditional\ \ Shuffle\ \ and\ \ Reduce)$ of key^k **then**
5: local reduce for key^k on the *n*th node
6: shuffle for key^k
7: **for all** $1 \leq n \leq numMapNodes$ **do**
8: **for all** $1 \leq k \leq numKeys$ **do**
9: **if** $key^k \in R_n$ **then**
10: local reduce for key^k on the *n*th node
11: find reduce for key^k, $1 \leq k \leq numKeys$

is less than the Cost(*traditional Shuffle and Reduce*) of key^k , key^k on the *n*th node will execute local reduce. On the other hand, if the *k*th key does not belong to R_n and the Cost(*Local Reduce, Shuffle, and final Reduce*) of key^k is not less than the Cost(*traditional Shuffle and Reduce*) of key^k, key^k on the *n*th node will execute shuffle directly.

- Lines 7-10: If the *k*th key belongs to R_n, key^k on the *n*th node will execute local reduce.
- Line 11: Finally, all keys will execute reduce.

It is noted that in order to highly overlap the shuffle and local reduce, some light loads need to be passed away immediately or conducted after local reducing, because light loads may be largely sent to other nodes to execute final reducing.

Rather, some heavy loads will largely execute ultimate reducing on their own nodes. Therefore, while heavy loads are executing local reducing, they receive corresponding keys from other Map nodes so that the shuffle and local reduce can be overlapped to a high degree.

IV. Performance Evaluation And Analysis

In this paper, we use different experiments to test the performance of the LELB algorithm and the MLSR algorithm. We apply the LELB algorithm and the MLSR algorithm to the MapReduce implementation of Merge Sort and WordCount. What's more, we use different sizes of data and different numbers of map tasks to discuss how the data scale and map tasks impact the computing performance. In the end, we give the explanation according to all the test results. The hardware and software test environment is shown in Tables VI and VII.

Table VI
THE HARDWARE TEST ENVIRONMENT

NameNode	DataNode
1 Intel multi-core server	3 SMP Intel Servers
4-way 4-core Intel Xeon 2.13 GHz	2-core Intel Xeon 3.0GHz
2 x 2M L2 Cache	1M L2 Cache
2GB Memory	
36GB Hard Disk	
2 x Intel EtherExpress/1000 network cards	

Table VII
THE SOFTWARE TEST ENVIRONMENT

NameNode	DataNode
Redhat Enterprise Linux Server Release 5.2	Fedora 3
hadoop: 0.20.2	
Eclipse: Europa 3.3	

A. Merge Sort

We compare the execution performance of Merge Sort between using classical MapReduce in hadoop and using the LELB algorithm and the MLSR algorithm proposed in this paper. On each DataNode, we use a basic class 'RandomWriter' to generate a specified size of unordered data set, which is the data source processed by Merge Sort.

Merge Sort uses classical MapReduce in hadoop: The initial data is the unsorted data. After the map phase, the value of the key in the pairs equals 0, and the value of the $value$ in the pairs is equal to the data. Then the reduce task receives the output of the map task, and runs the Merge Sort algorithm to generate the sorted result.

Merge Sort using the LELB algorithm and the MLSR algorithm as proposed in this paper: The initial data is the unsorted data. In the first place, the data set is divided into the map task. In our experiments, we set a public variable-named $counter$ in the configuration of the MapReduce job. The initial $counter$ is equal to 0. During the division of the initial data, we assign the $counter$ to each map task. During each map task, we use $counter$ and postfix in an increment of 1. For example, map task 3 firstly finished reading the data, using a $counter$ to be the output value of the keys in this map. The second map task using a $counter$ is map task 4; the $counter$ after postfix increment becomes 1. So the output value of the keys in the map 4 is 1. This operation means using counters to tag all data in each map to the subsequent operation. During the map phase, we set the value of the key in pairs equal to the $counter$, and the value of the $value$ in pairs is equal to data. Then we conduct the Local reduce operation: we sort the pairs according to the $value$. After forming new pairs in which the key stays the same but the $value$ becomes the sorted $value\text{-}list$, the $value\text{-}list$ is in ascending order by the values of the data. After the above operations, the reduce task receives all the output of the map task. We use the key's value as the tag to distinguish the $value\text{-}list$ and merge them. After several recursion merges, we finally generate the sorted data.

The Merge Sort algorithm consists of two parts: division and merge. We use map task to divide the initial data and conduct parallel sort, then we use the reduce task to merge the output of the map task. To improve the computing performance, we set a public variable named $counter$ and set all the values of the key in the pairs equal to the $counter$ if they come from the same map task. After parallel sorting the initial data, we could get the sorted $value\text{-}list$ which is equal to the local reduce operation. The local reduce operation will decrease the computing time of the reduce phase, and in this way we could improve the computing performance when faced with a large scale data set.

Fig. 9 shows the execution time when the map tasks is 10. When the data size is 1GB, the execution performance of Merge Sort using classical MapReduce in hadoop is better than that using the LELB algorithm and the MLSR algorithm, as proposed, because compared with Merge Sort using the LELB algorithm and the MLSR algorithm, Merge Sort using classical MapReduce in hadoop has little computing during the map phase. Merge Sort using the LELB algorithm and the MLSR algorithm takes more time to compute k-v pairs. As a result, the increasing time is longer than the time

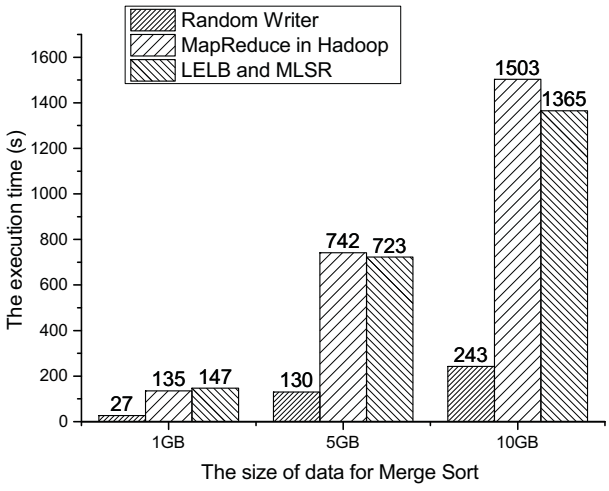

Figure 9. The relationship between the computing performance and the size of data for Merge Sort.

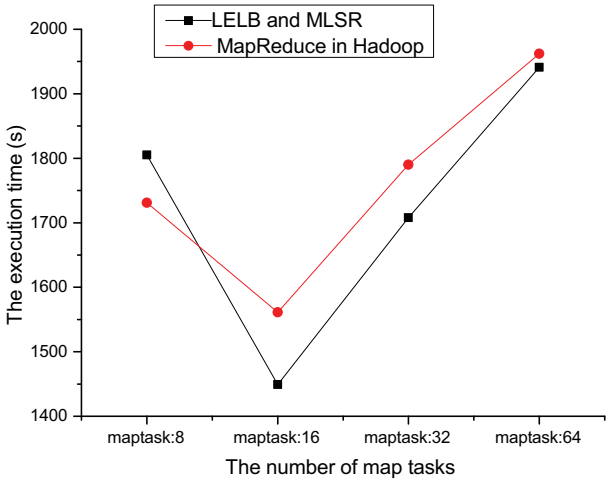

Figure 10. The relationship between the computing performance and the number of map tasks for Merge Sort.

which the local reduce operation saves, when the size of the data set is relatively small. However, when the data set comes to be 5GB, Merge Sort using the LELB algorithm and the MLSR algorithm is faster than Merge Sort using classical MapReduce in hadoop, by up to 2.5%. When the size increases to 10GB, the advantage ascends to 9.2%.

The experiment results demonstrate that as the size of the data set becomes bigger, Merge Sort using the LELB algorithm and the MLSR algorithm proposed in this paper has better execution efficiency than Merge Sort using classical MapReduce in hadoop, especially when faced with large-scale data.

Fig. 10 demonstrates the relationship between the computing performance and the map task number for Merge Sort. According to the data, when the number of the map tasks is 16, the sort algorithm has the best performance. If the number of map tasks is too little, each map has to process too much data. The lower the level of parallelism for DataNode, the weaker the performance. On the other hand, too many map tasks will bring too many disk I/O jobs. In particular, while the number of map tasks is far larger than the physical cores, the performance degradation is quite obvious.

B. WordCount

We compare the execution performance of WordCount between classical MapReduce in hadoop and using the LELB algorithm and the MLSR algorithm as proposed in this paper.

The test data sets (1GB,5GB,10GB) were generated by random word lengths varying from 3 to 10. In order to test the proposed algorithm in the paper, we test the WordCount program in 3 different ways:(1) WordCount without combiner using MapReduce in hadoop.(2) WordCount with combiner using MapReduce in hadoop.(3) WordCount using the LELB algorithm and the MLSR algorithm proposed in this paper.

WordCount without using classical MapReduce in hadoop: The 'No Combiner' WordCount is the classic example among the hadoop examples. Without combiner, the intermediate data will increase exponentially. This will challenge the network bandwidth.

WordCount using classical MapReduce in hadoop: The 'Combiner' WordCount is the default program given by the hadoop example. We test the data set by setting the $job.setCombinerClass(IntSumReducer.class)$. With this combiner, we can reduce the amount of intermediate data.

WordCount using the LELB algorithm and the MLSR algorithm as proposed in this paper: Firstly, the data set is equally divided to all the map tasks. In each map task, we firstly use our 'Duplicate Removal' method to get an ArrayList to save all of the words of map tasks: we call this ArrayList '$KeyList$'. In this KeyList, every word appears only once, so we call these keys 'LocalKeys'. Then we use an array called '$LocalSum[\,]$' to get and save the numbers of each of the LocalKeys of this map task. Finally, we write the context with a loop: we write the context for $KeyList.size()$ times, and each time we write the context with one of the LocalKeys and its number in this map task.

Fig. 11 shows the relationship between the computing performance and the size of data for WordCount. From Fig. 11, we learn that: When the amount of map tasks is 16 (The suitable maptask amount for this application), the performance improvement of 'the LELB algorithm and the MLSR algorithm' plan grows with the size of data set.

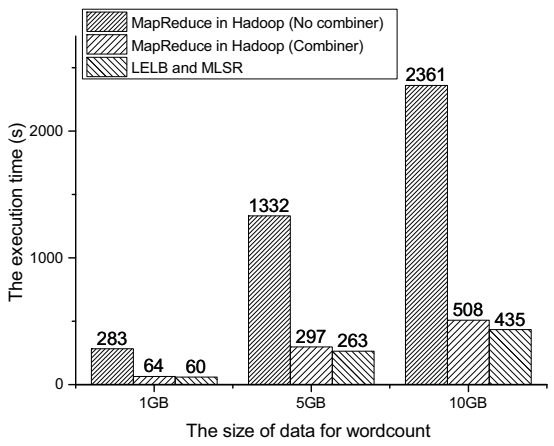

Figure 11. The relationship between the computing performance and the size of data for WordCount.

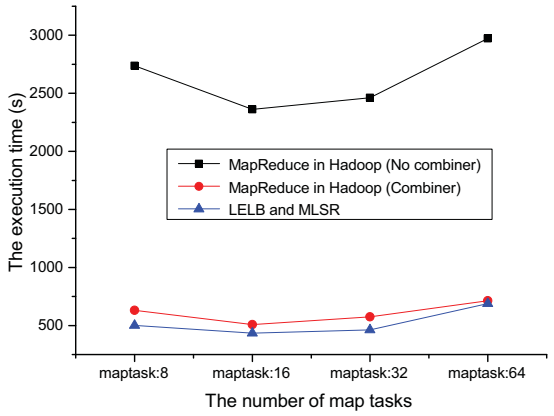

Figure 12. The relationship between the computing performance and the number of map tasks for WordCount.

When the data set is 1GB, 'the LELB algorithm and the MLSR algorithm' is faster than the Combiner plan by up to 6.7%. When the data set is 5GB, the improvement is 11.4%. Meanwhile, when the data set is 10GB, the improvement grows to 14.4%. Also from Fig. 11, we see that there will be a great performance degradation if the WordCount program runs without combiner. The execution time will increase by nearly 5 times.

Fig. 12 shows the relationship between the computing performance and the numbers of the maptask. Without a combiner, WordCount using MapReduce in hadoop runs slowly. The combiner is of great importance to the WordCount because it can significantly reduce the volume of data during the shuffle phase. According to Fig. 12, the suitable number of map tasks for this test is 16. When the maptask is 16, WordCount using the LELB algorithm and the MLSR algorithm proposed in this paper has the best permformance. This situation is faster than WordCount with a combiner using MapReduce in hadoop by up to 14.4%. As with the 'Combiner' WordCount, 'the LELB algorithm and the MLSR algorithm' WordCount can reduce the amount of intermediate data through the local reduce phrase. What is more, for the 'the LELB algorithm and the MLSR algorithm' WordCount, the volume of output data from the map task can be reduced significantly (due to the 'Duplicate Removal' method); thus the time of disk spinning is reduced. So the 'the LELB algorithm and the MLSR algorithm' WordCount is even faster than the 'Combiner' WordCount. We could see that it is more obvious compared to our Merge Sort test (which is 9.2%). The key reason is that by using the LELB algorithm and the MLSR algorithm proposed in this paper, the volume of intermediate data during the shuffle phrase can be reduced significantly (Each word has its own local sum value instead of 1) while the amount of intermediate data in the Merge Sort test remains unchanged (The only change is the order of the data). This is why it can save the time of intermediate data being transferred.

V. Conclusion

In this paper, we studied the performance optimization of MapReduce by considering the locality of keys and overlapping between shuffle and local reduce. This paper proposes a Locality-Enhanced Load Balance (LELB) algorithm, where Locality-Enhanced takes into account both the internal node locality and locality between all the nodes. Besides, this paper extends the execution flow of MapReduce to Map, Local reduce, Shuffle and final Reduce (MLSR), where local reduce tasks can be executed concurrently to decrease the computation amount of the final reduce task; at the same time, a decentralized outbreak of shuffle relieves the pressure on the network transmission and can be hidden by overlapping between shuffling and local reduce. Therefore, local reduce can share the computation of reduce and overlap with shuffle, in order to take full advantage of CPU and I/O resources. The actual test results show that when the size of the data set is 10GB, the execution using the LELB algorithm and the MLSR algorithm proposed in this paper is faster than the execution using classical MapReduce in hadoop, by up to 9.2% (for Merge Sort) and 14.4% (for WordCount).

Acknowledgment

This work was supported in part by the National High Technology Research and Development Program of China (No.2015AA01A303), Beijing Key Subject Development Project (XK10080537), NSF grants CNS 149860, CNS 1461932, CNS 1460971, CNS 1439672, CNS 1301774, ECCS 1231461, ECCS 1128209, and CNS 1138963.

REFERENCES

[1] "Hadoop," http://hadoop.apache.org.

[2] "Hadoop tutorial," http://developer.yahoo.com/hadoop/tutorial/.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *To appear in OSDI*, p. 1, 2004.

[4] R. Lämmel, "Googles mapreduce programming model‖revisited," *Science of computer programming*, vol. 70, no. 1, pp. 1–30, 2008.

[5] A. Verma, B. Cho, N. Zea, I. Gupta, and R. H. Campbell, "Breaking the mapreduce stage barrier," *Cluster computing*, vol. 16, pp. 191–206, 2013.

[6] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Mapreduce with communication overlap (marco)," 2007.

[7] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 17–24.

[8] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Handling partitioning skew in mapreduce using leen," *Peer-to-Peer Networking and Applications*, vol. 6, no. 4, pp. 409–424, 2013.

[9] L. Fan, B. Gao, X. Sun, F. Zhang, and Z. Liu, "Improving the load balance of mapreduce operations based on the key distribution of pairs," *arXiv preprint arXiv:1401.0355*, 2014.

[10] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Performance Evaluation*, vol. 70, no. 10, pp. 720–735, 2013.

[11] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, and P. Valduriez, "Data partitioning for minimizing transferred data in mapreduce," in *Data Management in Cloud, Grid and P2P Systems*. Springer, 2013, pp. 1–12.

[12] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications," *Open Cirrus Summit*, 2011.

[13] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society, 2012, pp. 419–426.

[14] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *OSDI*, vol. 8, 2008, p. 7.

[15] A. MateiZaharia, A. Joseph, and I. RandyKatz, "Improving mapreduce performance in heterogeneous environments," 2010.

[16] J. Tan, X. Meng, and L. Zhang, "Performance analysis of coupling scheduler for mapreduce/hadoop," in *INFOCOM, 2012 Proceedings IEEE*. IEEE, 2012, pp. 2586–2590.