Similarity-based Node Distance Exploring and Locality-aware Shuffle Optimization for Hadoop MapReduce

Jihe Wang, Danghui Wang, Meng Zhang
School of Computer Science,
Northwestern Polytechnical University,
Xi'an, 710127, CHINA
{wangjihe, dhwang, zhangm}@nwpu.edu.cn

Meikang Qiu
Seidenberg School of Computer Science and Information Systems,
Pace University, Manhattan,
NY, 10038, USA.
mqiu@pace.edu

Bing Guo
Computer Science College,
Sichuan University,
Chengdu, 610064, CHINA.
guobing@scu.edu.cn

***Abstract*—To shorten the networking delay from MapTracker to ReduceTracker has attractive potential to gain high performance shuffle for MapReduce. As the original MapReduce shuffle has no locality-aware feature when assigning reduce-tasks over computing nodes, we plan to present a *similarity-based* distance in the proposed *Cloud Node Space* to evaluate distance between two computing nodes in data center. Then, we implement a centralized and statistic-based locating service prior networking shuffle to place reduce-tasks near their corresponding data. Experimental results show that, comparing with Hadoop version, this service can achieve 2.3X speedup on shuffle time and bandwidth budget decreases by 60%.**

***Index Terms*—Locality, network, MapReduce, data-center, node distance, shuffle, centralized.**

I. Introduction

MapReduce has merged as a popular and easy–to–use programming model for cloud computing [7]. With a MapReduce framework, user only needs to provide two basic functions (*map* and *reduce*) to build a parallel algorithm in a distributed system to process a large amount of data. As an open source implement of MapReduce and Hadoop Distributed File System (HDFS), Hadoop [1] gives a much more powerful framework to scale computing capability in data-center, and plenty of tunable parameters to drill the system performance. Enhanced by load balancing and fault tolerant policies, Hadoop has been used extensively for underlying systems to deal with large scale data sets by companies like Yahoo!, Facebook, etc [10].

The map and reduce stages are the entries where user loads the algorithm into Hadoop context. However, the shuffle stage is the heart of MapReduce and is where the "magic" happens. MapReduce can guarantee that the input of every ReduceTracker (IRs) has been sorted by key. Therefore, shuffle stage is responsible for 1) sorting IRs off-memory, and 2) tranferring IRs from MapTrackers to ReduceTrackers. And a shuffle stage can be divided into three sub–operations in data-center. Firstly, in the end of map stage, IRs are grouped by key and spilled into local disk as a set of HDFS files. Then, from different MapTrackers, the IR-data labeled with same key are transferred to one ReduceTracker with HTTP protocol through network, and are also spilled in the local disk of the ReduceTracker node as HDFS files. At last, before reduce stage starts, the IRs with same key are combined and sorted into single file on ReduceTracker, so that the IR-data can be fed to the reduce function iteratively.

Unfortunately, for this kind of distributed system, when there are a lot of data blocks to be copied to the disks on remote nodes, the data transferring is another time consuming phase becuase of the long path on network. To address the latter issue in Hadoop MapReduce framework, we put the data-center nodes in a proposed *Cloud Node Space* (CNS). In this space, any node has a relative distance to each of other nodes within a tree–type typology. Based on the distance evaluation, called *similarity*, a locality–aware task assignment algorithm coordinates both task–end and allocator–end, as shown in Section II, to achieve short–shuffle–path in running time. Several enhancements are introduced in this paper:

1) To design the CNS, we adopt *similarity* to express the node distance in a tree–type typology, which combines two types of information: network typology and link bandwidth. Within this space, it is easy to find the node pairs which take short network paths to exchange data in a hierarchical storage system.
2) In the task–end, a statistical–based method is proposed in the centralized JobTracker to generate the IRs transferring distribution between MapTrackers and ReduceTrackers. Based on the node distance in CNS, a candidate set of TaskTracker is constructed to provide reduce slots for a key, or key group.
3) In the allocator–end, a novel scheduling algorithm is presented to explore the whole system for proper reduce slots, and allocate those slots with their corresponding reduce tasks. This allocation can place a reduce task around the nodes where the related IRs are stored, which cuts down the transferring delay of shuffle in running time.

978-1-5386-3684-8/17 $31.00 © 2017 IEEE
DOI 10.1109/SmartCloud.2017.23

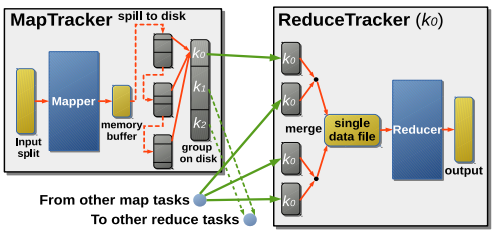

Fig. 1. The shuffle in MapReduce.

The rest of the paper is organized as follow. Section II illustrates the background and main idea of our approach. Section III shows the defines of the *similarity*-based CNS. The design of a new reduce task scheduler is presented in Section IV. Finally, Sections V and VI are the experimental results and conclusions, respectively.

II. Motivation and Methodology

The previous work on evaluating node distance models in [8], [4], [2] have the similar designing target that placing reduce tasks with lower overhead. However, their models treats the network bandwidth as constant values during distance calculation, which is a kind of concession since the available bandwidth between two nodes is not a fixed value, and data-center should avoid consuming bandwidth with exhaustive allocation. Another disadvantage of those work is that they use direct hop-count to evaluate the distances between two nodes, which ignore the fact that the current data-center takes fat-tree topology [3], where, the different depths of nodes in tree mark the different importance in distance evaluation.

Figure 1 shows the details of the shuffle phase in Hadoop MapReduce. In MapTracker, the mapper is fed with split data as input and writes IRs in a local memory buffer. When the contents of the buffer reach a certain threshold size, a background thread will sort the contents in-memory and spill them to local disk. After the last spill file is written to disk, all of the local spilled files are merged into single IRs partition, where, the IR-data with same key are grouped together. A ReduceTracker fetches all the IRs labeled with same key, k_0 in the example, from different MapTrackers and merges them into single data file as the input of reducer. In the Hadoop MapReduce implementation, there is no locality-aware service for the network-based data transfer within data-center, which results in that a lot of time is spent in the shuffle stage of Hadoop. Some studies show that there is nearly a third of time consumed by the shuffle phase when MapReduce process a large amount of data [5]. Furthermore, the number of random data accessing requests is huge from remote, however, the physical resources utilization of data-center is low during shuffle, especially for data-intensive Hadoop jobs [10].

To overcome those issues, we plan to add a locality-aware service to the centralized ReduceTracker allocator to cut down the overhead of networking-based remote IRs fetching during shuffle. The highlight of this service includes following two problems to be solved.

The first one is how to evaluate node distance with the tree-type architecture in data-center. In this work, we design a *similarity*-based *Cloud Node Space* which uses a direct method to calculate node distance. This method considers both network topology and bandwidth and provides the *similarity* of a node pair, which measures the network distance between the two nodes.

The second problem is how to let an allocator be aware of the most appropriate ReduceTracker slot for a given key. The original Hadoop MapReduce includes a centralized allocator for ReduceTracker, which gives a chance to analyze the distribution of the IR-data over network and select the near-IR-data node to shrink the workload budget of overall network. Thus, in this work, we provide an locality-aware allocator to arrange ReduceTrackers as near as possible to the corresponding IR-data.

III. Cloud Node Space

A. Definition of Similarity

Most of the current data-centers are tree-type organization, where, nodes are connected with undirected simple graph without cycles. Any two nodes of the tree are linked with an unique simple path, which is the shortest path between them, as shown in Figure 2. We consider a rooted tree that has a root node as the top-level switcher or router. The total number of nodes in the tree is donated by n, all nodes by $[v_1, v_2, \ldots, v_n]$; particularly, the root node by R. An ordered node sequence $P_{i,j} = [v^0_{i,j}, v^1_{i,j}, v^2_{i,j}, \ldots, \ldots, v^{h(i,j)}_{i,j}]$ is the unique path from v_i to v_j, where, $h(i,j)$ is the hop number of the path and $v^0_{i,j} = v_i$, $v^h_{i,j} = v_j$. The corresponding ordered edge sequence is $E_{i,j} = [e^1_{i,j}, e^2_{i,j}, \ldots, e^h_{i,j}]$, where, $e^k_{i,j} = (v^{k-1}_{i,j}, v^k_{i,j})$ and $k \leq h$. If $d(v_i, v_j)$ is the distance between the two nodes, a simple definition of the *similarity* could be as follow:

$$s_{i,j} = \frac{1}{1 + d(v_i, v_j)} \tag{1}$$

In our CNS model, two nodes are treated as a closer pair if they have a higher *similarity* value. In general, many types of monotonically decreasing functions could be used for this purpose.

However, this kind of definition is not useful for hierarchical topology like those current data-centers, because it makes no difference between similarities of node pairs located at different depth in a tree. Considering an example in Figure 2, based on Equation (1), the similarity between v_1 and v_2 should be equal to the similarity between v_9 and v_{10}, because they have the same hop length, $h(1,2) = h(9,10) = 1$. Unfortunately, in practice, the efficiency of data transmission on path $P_{1,2}$ can be higher than the efficiency on path $P_{9,10}$. In the hierarchical network, the edge $e_{9,12}$ is serving not only for the transmission between v_9 and v_{10}, but also for the transmission from nodes at lower level (v_6 and v_7) to the rest nodes in the tree. In this case, the available bandwidth on $e_{9,12}$ should be shared to many more connections of the three nodes (v_6, v_7 and v_9), rather than the single node (v_9). The reason

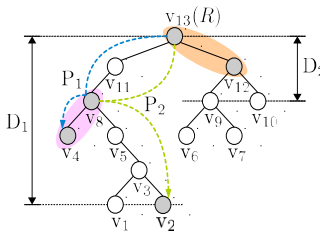

Table (a): Bandwidth of edges and root node R.

Edge	Bandwidth
(v2,v3)	0.5GB/s
(v3,v5)	2.0GB/s
(v5,v8)	1.0GB/s
(v4,v8)	1.0GB/s
(v8,v11)	1.5GB/s
(v11,v13)	1.0GB/s
(v12,v13)	1.0GB/s
R	0.8GB/s

Fig. 2. An example graph of tree-type topology in data-center.

of the problem comes from that the model in Equation (1) fails to consider the node depth in a tree. In Figure (2), the depth of v_1 and v_2, D_1, is 5, and the depth of v_9 and v_{10}, D_2, is 2. An intuitive statement is that a deeper node could have less number of bandwidth sharers in a hierarchical topology network, which would make the node enjoy better transmitting efficiency.

Another disadvantage of the similarity model in Equation(1) is that it treats all the edges as the link with the same bandwidth feature, which uses the hop number as distance of node pair. However, in practice, a data-center has the bandwidth limitation at different levels. For example, the speed of fetching IR-data from node's local disk cannot exceed the maximal disk I/O reading speed. Gigabyte-level switchers are widely used in the internal network of data-center, which also provides nonidentical bandwidth at different node levels. Furthermore, *Infrastructure-as-a-Service* (IaaS) could constrain the internal network bandwidth for a program to guarantee the performance of the overall system, called *budget*.

To overcome the drawback of the traditional *similarity* definition, we plan to replace it with a new one that considers both node depth and bandwidth variety.

Rather than using the hop number as node distance, we use the available bandwidth to represent the *resistance* through a path as follow:

$$r(v_i, v_j) = \sum_{e_{p,q} \in E_{i,j}} \frac{1}{b_{p,q}} \tag{2}$$

where, $b_{p,q}$ is the available bandwidth between two neighbor nodes on the path from v_p to v_q. The idea behind Equation (2) is that all the edges on a path can be treated as *resistance* to forward data. An edge is a "low-resistance" link if it features a high bandwidth, which could provide a fast pass to the next node. Thus, the *resistance* from v_i to v_j is the sum of all the segmented *resistances* on the path.

We define the lowest common ancestor node of v_i and v_j is $lca_{i,j}$. For example, in Figure 2, $lca_{2,4} = v_8$, and $lca_{2,12} = R$. To evaluate the depth of $P_{i,j}$, we use the depth of the highest node on the path to represent the path's depth. Actually, the highest node is the same node of the lowest common ancestor node $lca_{i,j}$. Then, the fraction of $P_{i,j}$ *resistance* to the depth path (from R to $lca_{i,j}$) resistance can locate the $P_{i,j}$ to a depth in tree, which meets the depth feature as mentioned before, as shown in Equation (3).

$$r_d(v_i, v_j) = \frac{r(v_i, lca_{i,j}) + r(lca_{i,j}, v_j)}{r_R + r(lca_{i,j}, R)} \tag{3}$$

where, r_R is the *resistance* on root switcher (or router). From Equation (3), a deeper $lca_{i,j}$ from root would results in a smaller $r_d(v_i, v_j)$ for identical paths, which means that the path has a faster data-pass than a "higher" path.

By replacing $d(v_i, v_j)$ with the $r_d(v_i, v_j)$ in Equation (3), we obtain a new definition of *similarity* between two nodes like Equation (1):

$$s_{i,j} = \frac{r_R + r(lca_{i,j}, R)}{r(v_i, lca_{i,j}) + r(lca_{i,j}, v_j) + r_R + r(lca_{i,j}, R)}. \tag{4}$$

This *similarity* is symmetric for any two nodes, which means $s_{i,j} = s_{j,i}$.

B. Node Similarity in data-center

The bandwidth within data-center is one of the scarce resources, which should be allocated to concurrent tasks, or virtual machines, in the form of budget. The work in [11] shows that the variety of inner optical bandwidth budgets ranges from 1Gbps to 40Gbps, and some less efficient services even cannot achieve 1Gbps. Furthermore, the switchers at different levels also feature very different physical bandwidths in a data-center. For example, for designing a server room Ethernet LAN, several 1Gbps Ethernet ports combine to a 10Gbps port as a switcher connecting to other servers [6].

TABLE I
THE SIMILARITY VALUES AMONG NODES v_2, v_4, v_8, v_{12}, AND v_{13} IN FIGURE 2.

Node	v_4	v_8	v_{12}	v_{13}
v_2	0.393	0.455	0.167	0.195
v_4		0.745	0.255	0.392
v_8			0.319	0.429
v_{12}				0.556

In Figure 2, we provide an example of the bandwidth in the tree-type network. As an example, we focus on the *similarity* among five nodes: v_2, v_4, v_8, v_{12}, and v_{13}. The relevant available bandwidths are listed in Table (a) of Figure 2. And Table I is their *similarity* values. With Table I, a *similarity*-based space is expended. For example, the *similarity* between node v_2 and v_4 is not only the function of the *resistance* of $P_{2,4}$, but also the function of their lowest common ancestor node v_8. The common section of P_1 and P_2 ($v_{13} - v_8$), as shown in Figure 2, scales the *similarity* of the node pair.

An interesting phenomenon is that the similarity value of (v_4, v_8) pair is unequal to the one of the (v_{12}, v_{13}) pair ($s_{4,8} = 0.745$, $s_{12,13} = 0.556$), though they have same available bandwidth ($b_{4,8} = b_{12,13} = 1.0Gbps$). The reason is that the two common network structures locate at different depths of the tree. Thus, the bandwidth between v_{12} and v_{13} has more sharers, such as those nodes under v_{12}, which results in less available bandwidth used by the "pure" connection between v_{12} and v_{13}. Therefore, in our CNS, (v_4, v_8) pair has higher *similarity* value than (v_{12}, v_{13}) pair.

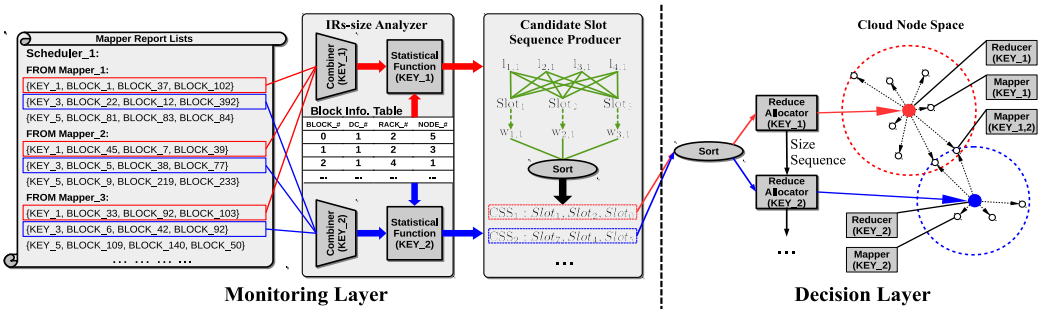

Fig. 3. Overall process of locality-aware ReduceTracker scheduler.

IV. Locality-Aware ReduceTracker Scheduler

A. Overview: Centralized Shuffle Optimizing

The optimal solution of allocating reduce tasks over nodes needs a global selection in all available reduce slots on data-center. We design a centralized service in master node, which collects locality reports about IR-data from all MapTrackers after those mappers have finished their map-side shuffles. Based on the locality reports, our service is able to calculate and compare the *weighted similarities* between those MapTrackers and available ReduceTracker slots, and exploring the best potential slot as the target ReduceTracker for each key label. For the convenience of discussion, we assume that each slot only contains one reduce task serving for constant number of keys, and workload balancing feature can be blended into our service easily. Figure 3 illustrates the procedure of the centralized reduce tack scheduling.

The input of the service is a *mapper report list* which contains the locality information of all the IR-data blocks over data-center, as shown in Figure 3. Each item in the list records the blocks that contain the corresponding key's IR-data in a MapTracker. For example, the first record in the list of Figure 3 shows that, in MapTracker 1, key_1-labeled IR-data is stored in blocks 1, 37, and 102. In our implementation, block number is statically assigned to each MapTracker, which avoids the name conflicting in the distributed storage system.

The output of the service is a set of key-slot pairs which guides the scheduler to assign a reduce task to a specific available slot, which features lower network overhead. We as well assume that a key is assigned to only one reduce slot, however, a reduce slot can served for no more than K_{max} keys. For the example in Figure 3, the *Cloud Node Space* in right side shows the optimal task assignment, where, key_1-task is scheduled to the slot on the red node which has the maximal *weighted similarity* to those MapTrackers with key_1-labeled IR-data. The details of the *weighted similarity* is in Section IV-B.

To achieve the optimal task assignment over data-center, the locality-aware scheduling contains two parts: the *monitoring layer* and *decision layer*. The *monitoring layer* analyzes the *Mapper Report List* (MRL) and extracts a ordered *Candidate Slot Sequence* (CSS) for each key with *parallel pattern*. The CSS lists the available slots with better-to-worse order to advise the further decision. The *decision layer* refers both CSS and IR-data size of keys to generate a feasible allocating order with *sequential pattern*. The methodology behind the allocation is to assign a key with a higher priority if it has bigger IR-data size, which could get more benefit on networking optimization.

B. Monitoring Layer: Statistic-based Location Exploring

Monitoring layer aims to provide a candidate slot sequence to the decision layer, which includes two sub-modules: *IRs-size analyzer* and *candidate-slot-sequence producer* (*CSS producer*), as shown in Figure 3. For a key_i, the *IRs-size analyzer* applies a statistic procedure to MRL to generate a distribution of the key_i-labeled IRs over data-center. Then, *CSS producer* calculates the similarity between each reduce slot node and the mapper nodes with the key_i-labeled IRs. The key_i available slot nodes are sorted in SC_i with "similarity" order.

A *IRs-size analyzer* scans the MRL file to search the locality of each IR-data group. We use a triple $(Mapper_i, key_j, blocks_{i,j})$ to represent a line of MRL, where the $blocks_{i,j}$ is the list of block labeled with key_j in $Mapper_i$. For the example in Figure 3, $(Mapper_1, key_1, [1, 37, 102])$ conveys the first record in $Mapper_1$. A combiner is assigned to each key to collect those records. Then, a statistical function $l_{i,j}$, as shown in Equation (5), accumulates the total size of key_j-labeled IR-data stored in $Mapper_i$.

$$l_{i,j} = \sum_{node \in blocks_{i,j}} size(node, key_j) \tag{5}$$

Actually, if the total MapTracker number is M_{max}, the vector $L_j : [l_{1,j}, l_{2,j}, \cdot, l_{M_{max},j}]$ is the size distribution of key_j-labeled IR-data over the data-center. The *CSS producer* will start from the size distribution to exploring the appropriate ReduceTracker slots. Furthermore, a block information table is passed to *CSS producer* by *IRs-size analyzer*, which includes the locality of blocks in tree-type architecture, such as the No. of data-center, rack, and node, to support *similarity* calculation.

The *similarity* defined in III-A is based on the static bandwidth budget and network topology, which cannot provide a real-time workload evaluation on network. Based on the vector L_j from *IRs-size analyzer*, in Equation (6), we define the "weighted similarity" $w_{i,j}$ between a set of MapTrackers

Algorithm 1: ReduceTracker assigning algorithm.

Data: $[L_1, L_2, \ldots, L_{kn}]$ and $[CSS_1, CSS_2, \ldots, CSS_{kn}]$.
Result: An sequence of $(key_j, Slot_i)$.

```
1  for j ∈ [1, kn] do
2      key_length_j = Σ_{k=1}^{M_max} l_{k,j} ;
3      css_list.append(CSS_j, key_length_j) ;
4  sort css_list with key_length_j by decreasing order;
5  while scc_i ← pop(css_list) do
6      while slot_i ← pop(css_i.css) do
7          if slot_i.used < K_max then
8              alloc_list.append(css_i.key, slot_i);
9              slot_i.used = slot_i.used + 1;
10             break;
11 return alloc_list;
```

and one ReduceTracker to evaluate the similarity between distributed key_j-labeled IR-data and a specific slot-node $Slot_i$.

$$w_{i,j} = \sum_{k \leq M_{max}} s_{k,i} l_{k,i} \tag{6}$$

where, k iterates each MapTracker, $s_{k,i}$ is the *similarity* between $Mapper_k$ and $Slot_i$ as defined in Equation (4). This *weighted similarity* can distinguish two keys by different IR-data distribution on data-center, which assigns a higher similarity value to a key if most of its IR-data is resident in those near nodes from $Slot_i$, though the paths are all identical. Since there is no computing dependence between any two slot-nodes, parallel way is applied to accelerate producing the vector $CSS_j : [w_{1,j}, w_{2,j}, \ldots, w_{S_{max},j}]$, where, S_{max} is the total number of available slot-nodes. After CSS_j has been produced, a sort operation is applied to it with decreasing order, where, a front element corresponds to a high-priority slot-node to be allocated to key_j. The degree of parallelism with this procedure is kn if there are total kn keys. When the CSS producer finishes producing CSS vectors for all keys (CSS_1 to CSS_{kn}), those vectors are passed to the *Decision Layer* as the recommending solution.

C. Decision Layer: ReduceTracker Assigning Algorithm

The *Decision Layer* refers the CSS vectors to assign reduce tasks to those available reduce slots. A set of tuple (L_j, CSS_j) for all $j \leq kn$ is the input of the this layer, and ReduceTracker assigning method is shown in Algorithm 1.

Lines 1 to 3 are the loop to statistic the IR-data size of each key. After the key_length_j obtained in line 2, it is tagged to CSS_j in line 3 to form a new CSS list (css_list). Then, in line 4, the css_list is sorted with decreasing key_length_j order, which makes the scheduler allocate slots for those keys with bigger size of IR-data. The following external loop (from lines 5 to 10) pops each key with the IR-data-size order from css_list to a iterating variable css_i. In the internal loop (from lines 6 to 10), the scheduler searches an available reduce slot in a key's CSS vector. Since each CSS vector arranges the slot node with a decreasing *similarity* order, the searching can find the appropriate slot node quickly. The kernel idea is that key_j is scheduled to $slot_i$ if the keys on the slot node haven't achieved the maximal capability (K_{max} defined in Section IV-A). Then, the $(key_j, slot_i)$ is appended to the allocating list in line 8. The number of key on $slot_i$ increases by 1 in line 9. Finally, in line 11, the allocating list is returned to the background thread on master node which refers the list to deliver reduce tasks over data-center.

As shown in Figure 3, the *Decision Layer* sorts keys with IR-data-size order, which results in that key_1 has a higher priority than key_2 to be assigned with an available reduce slot. In the CNS, the reduce slot has the higher *weighted similarity* than any other available slot for the IR shuffle between the set of MapTrackers and a single ReduceTracker. Then, the background thread on master node notices the ReduceTracker on the slot to start fetching the corresponding key-labeled IR-data from the relevant MapTrackers.

V. Experimental Results

A. Platform Setup

The experimental platform is a Hadoop data-center including 8 PC nodes and 7 switchers to form a full binary tree. There are 4 racks, each of which contains 2 PC nodes and 1 switcher with the bandwidth of 10Gbps. Every two racks are connected together with a switcher (10Gbps) as a rack group, and there are 2 rack groups that are connected by a top switcher with the maximal capability of 100Gbps. To balance the workload of the mapping and reducing, six of the PCs are statically configured as MapTrackers, and the rest two are only assigned with ReduceTracker jobs. Each PC node runs at most 12 map task ($M_{max} = 72$) or 4 reduce task ($S_{max} = 8$). The memory cards work on $1,333$MHz and has 17.4GB/s reading speed and 3.2GB/s writing speed. The local disks has the maximal 1.6GB/s reading speed and 95.6MB/s writing speed. As the architecture is symmetrical for all PC nodes, we randomly choose a PC node ($node\_1$) as the mater node of Hadoop to run the background thread of proposed scheduler.

We selected representative programs used in different domains: text analytics (String Match, Inverted Index, and Word Count), mathematical operation (Matrix Multiplication), web searching (Similarity Score, Page View Count, and Page View Rank), and machine learning (K-Means) [12], [9]. The spilled files are stored within *ext4* file system on the local disks of each node. The spill buffer in memory for each MapTracker is set to 64M. The sorting function is the build-in python function which default uses quick sorting pattern on IR-data.

B. Speedup on Shuffle

Figure 4 (a) is the time comparison between the original Hadoop random shuffle and the proposed locality-aware shuffle. Among the set of the benchmark programs, the K-Means obtains the maximal shuffle speedup rate about 3.1X. On average, our locality-aware scheduler can achieve 2.6X speedup to the original shuffle. Actually, the accelerating rate

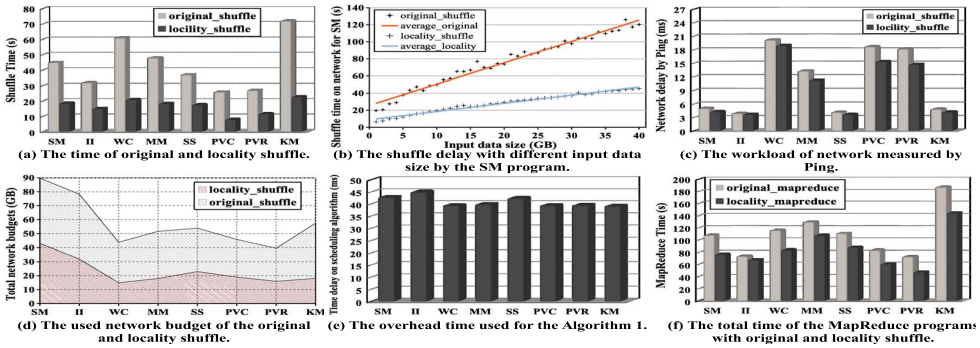

Fig. 4. The results of the locality-aware task scheduler.

could be higher than that if the node number of the data-center is scaled up because there will be much more candidate slot nodes to increase the *similarity* values.

Figure 4 (b) is an example of String Match to show the relationship of input data size and the speedup of shuffle. The figure plots the time used by shuffle with different input data from 1GB to 40GB. We find a linear increasing of the time without network congestion since a budget service can allocate enough network bandwidth to the program. The results show that our locality-aware scheduler can always accelerate the shuffle time by 2.5X for the String Match program.

To monitor the network status during shuffle, we use a script to automatically ping random remote nodes from each node. Figure 4 (c) shows that the locality-aware method has a little function to improve the network delay, however, the improvement (about 1.15X) is not that high as that of the shuffle time in Figure 4 (a). To obtain the real bandwidth budget used by the locality-aware shuffle, we record each remote IR-data fetching and add those data size together with hop factor. Fortunately, the locality-aware scheduler can help the budget service to save 60% bandwidth, as shown in Figure 4 (d). We believe that a new scheduler could become a potential technique to relieve the pressure of internal network on data-center.

Though there is 2.6X time saving gained by the locality-aware scheduler, the computing overhead of the scheduler is negligible, as shown in Figure 4 (e). The original Hadoop scheduler choose a random node as the ReduceTracker. We assume that the original scheduler is an instant one without any time delay. The new scheduler completes the slot allocating within 45ms in our platform, which impacts little on the shuffle performance. Figure 4 (f) is the speedup of the whole MapReduce framework gained by the new scheduler. Since the shuffle stage contributes nearly 40% processing time of MapReduce on the platform, our optimization on shuffle can achieve about 1.32X speedup on average.

In summary, the locality-aware scheduler is able to shrink the shuffle time and keep the network responding time, while, the computing overhead is negligible.

VI. Conclusions

This paper proposes a new *similarity*-based method to measure the distance between two nodes in a tree-type architecture. In the MapReduce and data-center context, a *cloud node space* is expanded to help the exploring the best reduce slot in data-center. Based on the CNS, a new locality-aware scheduler is designed to allocate reduce tasks to the available slots on data-center with lowest *similarity* values.

Acknowledgments

This work was supported in part by NSFC No. 61472322, 61772352, 61472050, 61332001.

References

[1] Apache hadoop project. http://hadoop.apache.org/.
[2] E. Arslan, M. Shekhar, and T. Kosar. Locality and network-aware reduce task scheduling for data-intensive applications. In *DataCloud '14*, pages 17–24, 2014.
[3] M. Bradonjić, I. Saniee, and I. Widjaja. Scaling of capacity and reliability in data center networks. *ACM Sigmetrics Performance Evaluation Review*, 42(2):46–48, Sept. 2014.
[4] X. Bu, J. Rao, and C.-z. Xu. Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In *HPDC'13*, pages 227–238, 2013.
[5] F. Chen, M. Kodialam, and T. Lakshman. Joint scheduling of processing and shuffle phases in mapreduce systems. In *INFOCOM'12*, pages 1143–1151, March 2012.
[6] Cisco. Server room technology design guide. *The Cisco Validated Design Guides*, Aug. 2013.
[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04*, pages 1–10, Berkeley, CA, USA, 2004.
[8] M. Hammoud and M. F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 570–576, 2011.
[9] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *PACT '08*, pages 260–269, 2008.
[10] J. Li, X. Lin, X. Cui, and Y. Ye. Improving the shuffle of hadoop mapreduce. In *CLOUDCOM '13*, pages 266–273, Washington, DC, USA, 2013.
[11] A. Mahimkar, A. Chiu, R. Doverspike, M. D. Feuer, P. Magill, E. Mavrogiorgis, and et al. Bandwidth on demand for inter-data center communication. In *HotNets-X'11*, pages 24:1–24:6, 2011.
[12] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *HPCA '07*, pages 13–24, Washington, DC, USA, 2007.
[13] W. Yu, Y. Wang, and X. Que. Design and evaluation of network-levitated merge for hadoop acceleration. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):602–611, Mar. 2014.