International Journal of Database Theory and Application
Vol.8, No.1 (2015), pp.1-10
http://dx.doi.org/10.14257/ijdta.2015.8.1.01

A Reduce Task Scheduler for MapReduce with Minimum Transmission Cost Based on Sampling Evaluation

Xia Tang [1], Lijun Wang [2*] and Zhiqiang Geng [1*]

[1]*College of Information Science and Technology, Beijing University of Chemical Technology, Beijing, 10029, China*
[2]*Network Research Center, Tsinghua University, Beijing 100084, China*
2012210217@grad.buct.edu.cn, hanym@mail.buct.edu.cn, gengzhiqiang@mail.buct.edu.cn, wanglijun@cernet.edu.cn

Abstract

MapReduce is a popular framework for processing large datasets in parallel over a cluster. It has gained wide attention for its high scalability, reliability and low cost. However, its performance may be degraded by excessive network traffic when processing jobs, for such two problems as data locality in reduce task scheduling and partitioning skew. We propose a Minimum Transmission Cost Reduce task Scheduler (MTCRS) based on sampling evaluation to solve the two problems. The MTCRS takes the waiting time of each reduce task and the transmission cost set as indicators to decide appropriate launching locations for Reduce tasks. The transmission cost set is computed by a mathematical model, in which the parameters are the sizes and the locations of intermediate data partitions generated by Average Reservoir Sampling (ARS) algorithm. The experiments show that the MTCRS reduces network traffic by 8.4% compared with Fair scheduler.

Keywords: *MapReduce, task scheduling, data locality, partitioning skew, transmission cost*

1. Introduction

MapReduce [1] has become a popular framework for processing and generating large datasets in parallel over a cluster. Hadoop [2], as a popular open source implementation of MapReduce, is successfully applied in many applications such as web indexing, report generation, data mining, log file analysis. The MapReduce system runs on the top of the Hadoop Distributed File System (HDFS) [3], in which data is loaded and partitioned into chunks, with each chunk replicated across multiple machines.

As a new abstraction, MapReduce hides the messy details of parallelization, fault-tolerance, data distribution and load balancing to automatically process and analyze large datasets, which allows users to express simple computations in the form of user-defined map functions and reduce functions. As an open source implementation of MapReduce, Hadoop has been widely accepted by the industry for its scalability, low cost and reliability, but its performance is still far behind the parallel database management systems [4]. In recent years, the performance optimization of Hadoop has been widely studied, and generally it can be divided into three aspects: 1) developing efficient applications, 2) adjusting parameters in configuration files of Hadoop [5], 3) modifying internal mechanisms of Hadoop to fix original defects [6]. The last approach is difficult but very promising, which is also our focus. In aspect of optimizing internal mechanisms

* To whom correspondence should be addressed: E-mail: gengzhiqiang@mail.buct.edu.cn, Tel:+86-10-64426960, Fax: +86-10-64437805. Email: wanglijun@cernet.edu.cn, Tel: +86-10-62603017, Fax: +86-10-62785822.

ISSN: 2005-4270 IJDTA

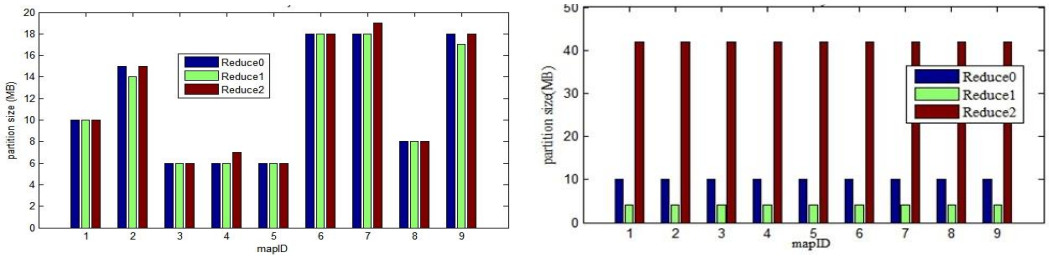
of Hadoop, scheduling optimization gets extensive attention. Nowadays there are three commonly used schedulers: default FIFO scheduler, Fair scheduler [7] and Capacity scheduler [8], and Fair scheduler is the most widely applied one. But all of them ignore the problem of data locality in scheduling Reduce tasks [9] and partitioning skew [10].

1.1. Data Locality in Reduce Tasks Scheduling

In distributed environment, data locality refers to moving computing tasks to the nodes where input data locate to reduce network traffic. When a user uploads files to HDFS, Files are divided into chunks ranging from 16MB to 64MB. When a user submits a job, current schedulers only consider data locality in scheduling Map tasks and neglect the data locality of Reduce tasks. The data locality of Reduce tasks is ignored for the following two reasons: Firstly, the number of Reduce tasks is always less than the number of Map tasks. Secondly, Reduce tasks are launched when certain percentage (0.05 by default) of Map tasks have been completed. At this moment, only part of intermediate data is generated. Therefore, it is impossible to decide which node is the best one to launch a Reduce task. However, if the Reduce tasks are placed at improper nodes, additional network traffic will be generated during transmitting intermediate data.

1.2. Partitioning Skew

After a Map task is launched, it reads the contents of the corresponding input split (one or more chunks) and parses them into key/value pairs and passes each pair to its user-defined Map function. The intermediate key/value pairs produced by this Map function are buffered in memory. Periodically, the buffered pairs are partitioned into R (R is the number of Reduce tasks) regions by the partitioning function and written to local disk. When a Reduce task is launched, it will copy its own partitions from nodes where Map tasks are placed. Data non-uniform distribution and improper partitioning function will lead to partitioning skew. Figure 1(a) shows the partitioning skew caused by the non-uniform distribution of data. The amount of data belongs to the same Reduce task exhibits a significant discrepancy on each Map node even though buffered pairs are evenly partitioned into R regions. Figure 1(b) shows improper user-defined partitioning function leads to partitioning skew even though the amount of data on each Map node are equal. In summary, if a Reduce task is placed on node which does not have much intermediate data, there will be much unnecessary network traffic.



1(a) Partitioning Skew by Data Distribution

1(b) Partitioning Skew by Partitioning Function

Figure 1. Two Partitioning Skew Scenarios

In order to solve the two problems mentioned above, we propose the Minimum Transmission Cost Reduce task Scheduler (MTCRS). Our experiments indicate that MTCRS can efficiently reduce network traffic. In this paper we make following contributions:

- We study the problem of data locality in scheduling Reduce tasks and partitioning

skew through several experiments, and explain reasonable Reduce Task locations can reduce network traffic during intermediate data transmission.

- We propose a sampling approach based on ARS algorithm to predict the distribution of overall partitions by computing distribution of the corresponding partitions for sampled data. Experiments show the accuracy and efficiency of this sampling approach.
- In order to compute best Reduce tasks locations, we build a transmission cost mathematical model whose parameters are the sizes and locations of partition of intermediate data generated by processing sampled data to minimize network traffic. And we implement a new scheduler based on this model MTCRS. Compared with Fair scheduler, MTCRS can reduce network traffic by 8.4%.

The rest of the paper is organized as follows. Section 2 presents the background of Hadoop and related work. The design details on MTCRS are described in Section 3. Section 4 is the evaluation of our approach. Finally, the conclusion and future work are given in Section 5.

2. Background and Related Work

2.1. The Process from Job Submission to Job Launching

Hadoop adopts the Master/Slave structure composed of several components: Client, JobTracker, TaskTracker and TaskScheduler. Figure 2 shows the entire detailed process of a job from submission to launching[11],which involves these components: 1) The Client submits a program to JobTracker by calling the function of job submitting; 2) After receiving the message from the Client, JobTracker notifies TaskScheduler to initialize the job, which decomposes the job into several Map tasks and Reduce tasks; 3) Every once in a while, TaskTracker sends heartbeat to JobTracker to report its resource information and whether it can accept a new task; 4) If a TaskTracker can accept a new task, JobTracker calls the assignTasks function of TaskScheduler to schedule a new task; 5) TaskScheduler chooses appropriate task list with certain strategy and return it to JobTracker; 6) JobTracker sends this list to corresponding TaskTracker; 7) TaskTracker receives the reply and launches tasks.

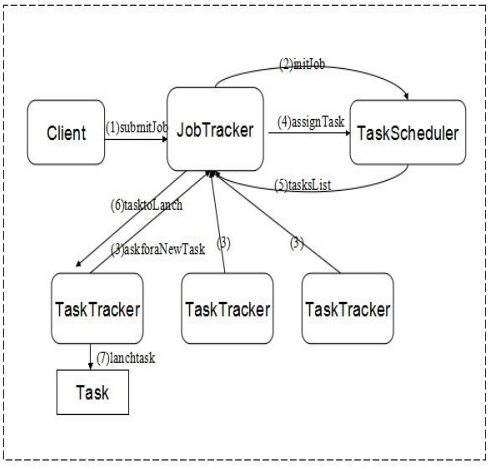


Figure 2. The Process Procedure of a Job

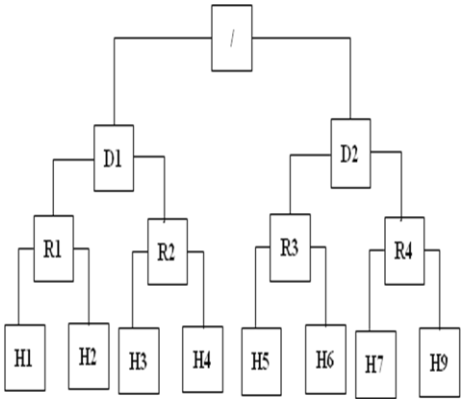


Figure 3. Network Topology of Hadoop Cluster

2.2. Typical Network Topology of Hadoop Cluster

In order to solve the problem of data locality, the network topology of a cluster should be provided. Figure 3 depicts a three layer network topology [10], where the root node represents the entire cluster and the nodes in each layer from top to bottom present data centers, racks (switches) and the actual physical nodes used to calculate and store data respectively. In practice, distance of two nodes is computed in the following metrics in Hadoop: the distance between a child node and its parent is 1, and we define the distance between two nodes as the sum of their distances to their common closest ancestor. For instance, the distance between H2 and H4 is 1 +1 +1 +1 = 4.

2.3. Research on Task Scheduling

There are lots of approaches against the data locality of Reduce tasks. Tan Jian et al.[9] have designed a resource-aware scheduler for Hadoop to mitigate job starvation problem and improved the overall data locality, which utilizes Wait Scheduling for Reduce Tasks and Random Peeking Scheduling for Map Tasks to optimize task placement. Mohammad Hammod *et al.*, [12, 13] have investigated the problems of data locality and partitioning skew in Hadoop and proposed CoGRS, a locality-aware and skew-aware reduce task scheduler to save network traffic. LaSA [14] is a locality-aware scheduling algorithm for MapReduce scheduler. A mathematical model based on the weight of data interference in scheduler is used to provide data locality-aware resource assignment. Seo *et al.*, [15] have designed two optimization schemes, Prefetching and Preshuffling, and implement them as a plug-in component called HPMR to improve the overall performance. Chen *et al.*, [16] have presented a grid-enabled MapReduce framework called “Ussop” to provides a set of C-language based MapReduce APIs and an efficient runtime system for exploiting the computing resources available on public-resource grids. Ussop introduces two novel task scheduling algorithms, VSMS and LARS, and achieves great performance.

As for partitioning skew, the current MapReduce implementations have overlooked the skew issue [17], which is a big challenge to achieve successful scale-up in parallel query systems [18]. Ibrahim *et al.*, [10] have investigated the problem of Partitioning Skew in MapReduce-based system and developed an algorithm, LEEN, for locality-aware and fairness-aware key partitioning in MapReduce. SkewReduce [19] is a system implemented on top of Hadoop for feature extraction analyses. The core of SkewReduce system is an optimizer, parameterized by user-defined cost functions, that determines how best to partition the input data to minimize computational skew. Kwon *et al.*, [20] described partitioning skew caused by various reasons and gave some suggestions to avoid its negative effects. Qiu *et al.*, [21] showed the partitioning skew in biomedical application and analyzed its effects on scheduling mechanism and gave detailed performance discussion. Finally, Lin [22] explained Zipfian distribution of intermediate data and proposed a theoretical model, describing the impact that distribution of input data impose on extracting parallelism.

3. The Design of the New Reduce Task Scheduler

3.1. ARS Sampling Algorithm

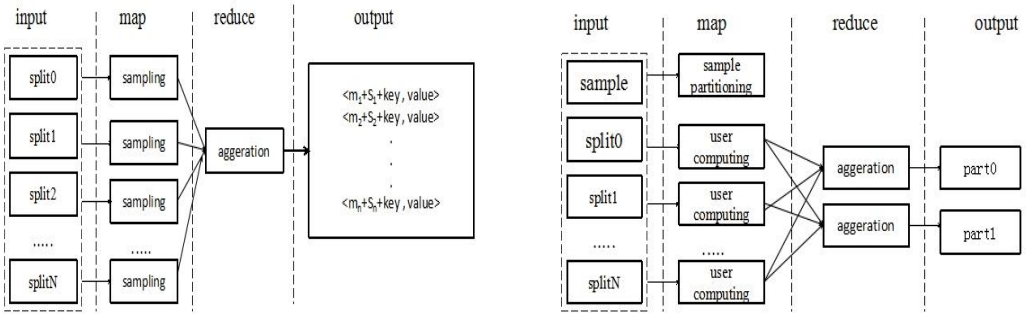
Sampling methods are widely applied in data stream approximate aggregate queries and streaming data analysis. In this paper, we decide to treat sampled data as the input data of Map tasks to estimate the partitions of all intermediate data. So we need to run an extra sampling job to obtain the sampled data. As is shown in Figure 4(a), the map function is used for sampling key/value pairs from input split, and the reduce function is to merge these pairs together and store them to one file. In our design, sampling process is from the Reservoir Sampling algorithm [23], which is free from the number of key/value pairs. However, due to the randomness of RS's sampling results, we propose Average Reservoir Sampling (ARS) algorithm, the main idea of ARS is as follows:

Step1: We build 5 reservoirs for each split and sample K elements from it. All key/value pairs in the split are scanned and the first K elements are stored in each reservoir.

Step 2: For a key/value pair whose line number is larger than K, we replace stored elements with it in a certain probability. This process is executed for each reservoir.

Step 3: Each sampled key/value pair is marked with its split ID S_i and the reservoir ID m_i;

Step 4: All the sampled key/value pairs are summarized together and stored into a file by reduce function.



4(a) Sampling Job 4(b) User Job

Figure 4. The Process of Sampling Job and User Job

3.2. Transmission Cost Mathematical Model

As is shown in Figure 4(b), when the sampling process is completed, both sampled data and original input data are passed to map function and computed in different ways. Firstly, for each sampled key/value pair, the split ID and reservoir ID marked above are extracted. And then the Reduce task ID that each pair belongs to it is calculated by partitioning function. Then the number of bytes of key/value pairs are calculated that belongs to the j^{th} Reduce task from the i^{th} Map task in the r^{th} reservoir, $D_r(i,j)$. Finally, the partition proportion $P(i,j)$ is calculated according to Formula 1, in which M and R represent the number of Map tasks and the number of Reduce tasks respectively, and S represents the number of reservoirs. At last, $P(i,j)$ is stored in heartbeat and then sent to JobTracker for further use. JobTrakcer stores the received partition proportion in the corresponding self-defined data structure of JobInProgress.

$$P(i,j) = \left(\sum_{r=1}^{r=S} (D_r(i,j) / \sum_{s=1}^{s=M} \sum_{t=1}^{t=R} D_r(s,t))\right) / S \quad (1)$$

When all these values of one job are saved, the transmission cost mathematical model is used to compute the best Reduce task locations. Assume $T_r = (N,E)$ denotes the network topology of Hadoop discussed in Section 2, where N and E represent the node set and the edge set of this tree respectively, and $Dis(u,v)$ is the hop distance between the node u and node v. U represents the set of nodes containing intermediate data for a job. We define N_r to be the set of nodes that have available reduce slots. When the j^{th} Reduce task is launched on node $v(v \in N_r)$, we can compute its transmission cost $C_j(U,v)$ as formula 2.

$$C_j(U,v) = \sum_{i=1}^{i=|U|} P(i,j) Dis(U_i, v) \quad (2)$$

For each node $v(v \in N_r)$, we calculate the corresponding transmission cost $C_j(U,v)$ for the j^{th} Reduce task and store the Cost/Node pair to a list CNList[j] in ascending order of the cost value. Obviously the node at the first place of CNList[j] (CNList[j] ∈ CNList) is the best location to launch the j^{th} Reduce task.

Algorithm 1: MTCRS

```
1: static values: min_cost=Integer.MAX_VALUE,
                  Vcost=0,vnode=null,WaitSet=null
                  nodemax=numNodes, waitTimeThreshold=T.
2: Input: RRT: remaining Reduce tasks, TT: TaskTracker,
          RWait: wait time set of elements in RRT,
          CNList: the list of CNList[j].
3: output: Tip: the selected reduce task.
4: if CNList.size()=numOfReduceTasks then
5:    for j=1 to RRT.size() do
6:       if RWait[j] > T then
7:          WaitSet.add(RWait[j])
8:       end if
9:    end for
10:   for j=1 to WaitSet.size() do
11:      if Mincost > CNList[WaitsSet[j].RID].get(vnode) then
12:         Mincost = cost; Tip = WaitsSet[j].RID
13:      end if
14:   end for
15:   if Tip != null then
16:      RRT.remove(Tip); return Tip
17:   end If
18:   for j=1 to RRT.size() do
19:      if CNList[j].get(0) == vnode then
20:         Tip=RRT[j] ; RRT.remove(Tip); return Tip
21:      else RWait[j]++;
22:    end if
23:   end for
24: end if
25: return null
```

3.3. MTCRS

In this paper, we replace the mechanism of scheduling Reduce tasks in Fair scheduler with a new Reduce Task scheduler named MTCRS and its scheduling process is as follows.

After a certain number of Map tasks (the default percentage is 5%) are completed and the transmission cost list of all tasks are calculated, Reduce tasks get the opportunity to be scheduled. When JobTracker receives a request heartbeat for a Reduce task from a TaskTracker, MTCRS is called. Firstly, MTCRS checks whether there exist Reduce tasks whose waiting time ($t \in$ RWait) are over threshold T (the number of slave nodes). If there exist, store them in a set named WaitSet and choose the Reduce task whose CNList[j] (CNList[j] ∈ CNList) claims the transmission cost of the node where the current TaskTracker locates is the lowest. Otherwise, check whether there exists a Reduce task with its CNList ranking the node where the current TaskTracker locates first. If there exists, assign that Reduce task to the current TaskTracker and remove its CNList and waiting time. Otherwise, increment the waiting time of all the remaining Reduce tasks. The Reduce task assignment process is described by Algorithm 1.

4. Experiments and Evaluation

4.1. Environment and Datasets

We conduct extensive experiments to evaluate the approaches proposed in this paper. Our cluster contains 1 master node and 9 slave nodes. Each node is a 64-bit Intel Core machine with one four-core CPU 2.0 Ghz CPU, 4GB physical memory and 250GB disk, and runs CentOS release 6.4. The version of Hadoop is Apache Hadoop 1.2.0. Each node is configured with 4 map slots and 2 reduce slots. As for input data, we use Random Writer to generate Input01, Input02 and Input03 whose sizes are 1G, 10G and 1.8G respectively. Furthermore, we download a real world dataset, containing 1472 English novels and named as Input04 for convenience and it is merged into one 445M file.

Each line in Input01 and Input02 is a sentence composed of random words. And the contents of Input01 and Input02 both obey uniform distribution. Each line in Input03 is in the form of key/value pair, and the contents of Input03 and Input04 do not obey uniform distribution.

4.2. ARS

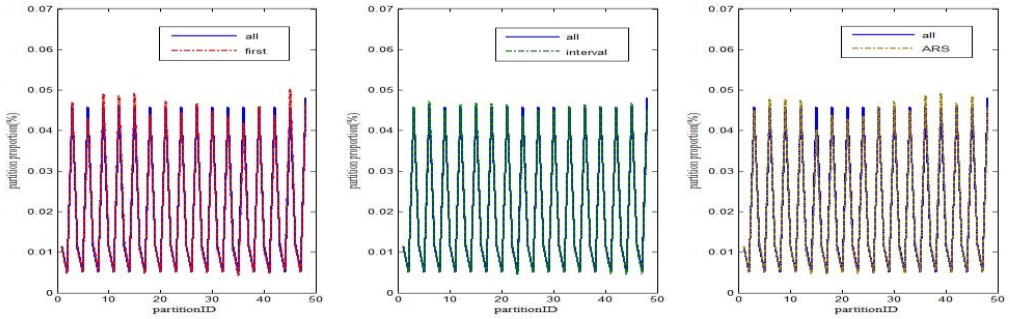
In order to verify the effectiveness of ARS, we use other two sampling algorithms, FirstSample and IntervalSample to conduct contrastive experiments. FirstSample (FS) chooses the first *k* key/value pair as sampled pairs, while IntervalSample (IS) samples *k* key/value pairs at fixed intervals. We submit 6 jobs with different input data and sampling algorithms and generate 6 different sample files, the information of jobs is shown in Table 1. From Table 1, we can see sampling jobs with different algorithms runs fast and sample files are small .We find it works well when k is 100 in practice.

Table 1. Jobs with Different Sampling Methods

JobID	Job Name	Time(s)	# Map	Sample File	Sample Size
1	FsJob*Input01	16	16	Fsample01	1.2M
2	ISJob*Input01	17	16	Isample01	1.3 M
3	ARSJob*Input01	16	16	Asample01	5 M
4	FSJob*Input04	9	7	Fsampel04	566kB
5	ISJob*Input04	10	7	Isample04	284kB
6	ARSJob*Input04	9	7	Asample04	1.5M

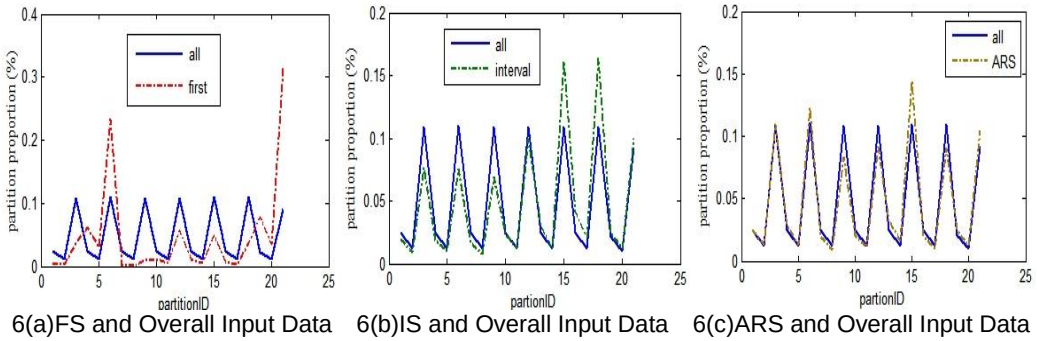
In order to evaluate the effectiveness of FS, IS and ARS when sampling from Input01, the sampled data called Fsample01, Isample01 and Asample01 respectively are processed by user Job(with user-defined partitioning function) to get partition proportion. The result is shown in Figure 5, where overall data is drawn in blue, the y axis represents partition proportion for sampled data. We can see lines are highly overlapped for all sampling algorithm, which indicates that they can all represent the distribution of overall data well when the input data obeys uniform distribution.

After sampling from Input04 with these three algorithms, the sampled data called Fsample04, Isample04 and Asample04 respectively are processed by user Job (with default partitioning function). We find that the results of different sampling algorithms differ greatly. Figure 6(a) shows that there exists large discrepancy between the partition proportion distribution of all and FS. The similarity reaches as low as 48.3%. And the sampling result of IS is better than FS whose similarity reaches 81.32% as is shown in Figure 6(b). Furthermore, ARS has the best result as the similarity reaches as high as 92% in Figure 6(c).



5(a)FS and Overall Input Data 5(b)IS and Overall Input Data 5(c)ARS and Overall Input Data

Figure 5. The Distribution Comparisons of Different Sampling Results and Uniform Data



6(a)FS and Overall Input Data 6(b)IS and Overall Input Data 6(c)ARS and Overall Input Data

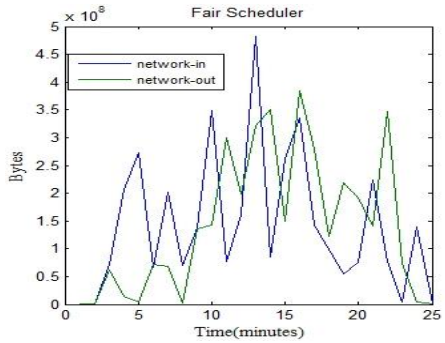
Figure 6. The Distribution Comparisons of Different Sampling Results and Non-Uniform Data

4.3. MTCRS

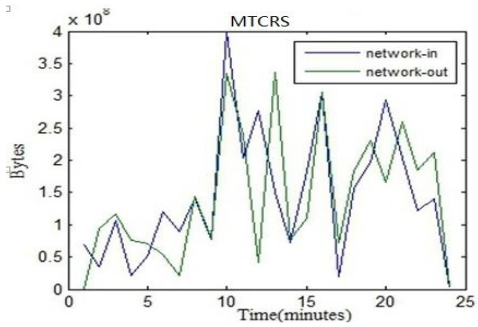
In this paper, we use network traffic and execution time as indicators to evaluate the performance of MTCRS compared with FairScheduler, which is the most widely used.

4.3.1 Network Traffic: The WordCount benchmark without combiner function is executed for 5 times with Input01 to evaluate the performance of MTCRS. The network traffic is monitored by Wireshark [24]. Figure 7(a) shows the input network traffic and output network traffic on whole nodes during shuffling intermediate data with Fair scheduler, and Figure 7(b) is with our scheduler MTCRS. As we can see, MTCRS scheduler produces 8.4% less network traffic than Fair Scheduler for exactly the same job.

4.3.2 Execution Time: The performance of Fair Scheduler and MTCRS is evaluated by comparing their own execution time for three jobs: WordCount, Terasort, Sort with Input01, Input03, Input03 as input data respectively. They are all executed for 5 times. We plot the average, minimum and maximum time for the three jobs scheduled by Fair Scheduler and our MTCRS Scheduler respectively. As Figure 8 shows, when applied to Terasort and Sort, the job execution time of MTCRS is less than the time of Fair Scheduler. However, the job execution time for Wordcount of MTCRS is 2.8% more than the time of Fair Scheduler.



7(a) Network Traffic of Fair Scheduler



7(b) Network Traffic of MTCRS Scheduler

Figure 7. The Comparison Results of Network Traffic with Fair Schedulers and MTCRS

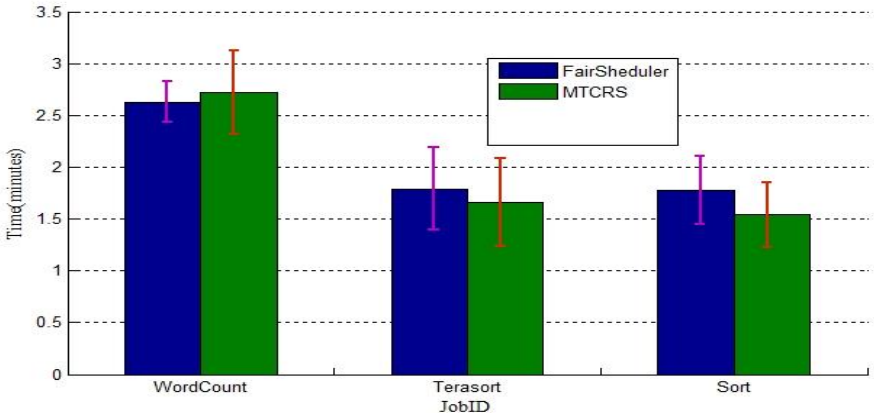


Figure 8. The avg, max and min Execution Time for Jobs with Fair Scheduler and MTCRS

5. Conclusion and Future Work

In this paper, we propose a new scheduler MTCRS based on sampling evaluation to solve the two problems of data locality in scheduling Reduce tasks and partitioning skew. First of all, we prove the effectiveness of Average Reservoir Sampling algorithm with control experiments of FS and IS. And the extensive experiments show that MTCRS can reduce network traffic by 8.4% compared with Fair Scheduler. However, the time performance of MTCRS does not stand out especially when submitting a job whose Map task number is more than the amount of available slots. The reason is that when there is not enough slot available to launch Map tasks, some Map tasks have to wait until being scheduled which in turn leads to the delay of knowing the locations of Map tasks, and it further postpones calculating transmission cost. In the future work, we will try to take more factors into account, such as available slots, the locations and size of intermediate data, to build a more comprehensive model. During experiments, we also find Hadoop costs too much time for IO operations, and decide to improve its performance by designing a more reasonable and compact input format in the future work.

Acknowledgements

This work was supported partially by National natural science foundation of China (No.61374166) and Doctoral Fund of Ministry of Education of China (No.20120010110010).

References

[1] J. Dean and S. Ghemawat, COMMUN. ACM, vol. 1, no. 51, **(2008)**.
[2] Hadoop. http://hadoop. apache.org, Apache Software Foundation.
[3] D. Borthakur, "HDFS architecture guide", HADOOP APACHE PROJECT http://hadoop.apache. org/common/docs/current/hdfs design. Pdf. **(2008)**.
[4] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo and A. Ra, Communications of the ACM, vol. 1, no. 53, **(2010)**.
[5] S. Babu, "Towards automatic optimization of MapReduce programs", Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, **(2010)** June 10.
[6] J. Dittrich and J. A. Quiané-Ruiz, Proceedings of the VLDB Endowment, vol. 12, no. 5, **(2012)**.
[7] FairScheduler, http://hadoop. apache. org/mapreduce/docs/r0. 21. 0/fair scheduler. html.
[8] CapacityScheduler, http://hadoop. apache. org/mapreduce/docs/r0. 21. 0/capacity scheduler. Html.
[9] J. Tan, X. Meng and L. Zhang, "Coupling task progress for mapreduce resource-aware scheduling", INFOCOM, 2013 Proceedings IEEE, Turin, Italy, **(2013)** April.
[10] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He and L. Qi, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud", Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference, Indianapolis, IN, USA, **(2010)** November, pp. 17-24.
[11] X. Dong, "Hadoop Internals: in-depth Study of MapReduce", China Machine Press, China, **(2013)**.
[12] M. Hammoud, M. S. Rehman and M. F. Sakr, "Center-of-gravity reduce task scheduling to lower mapreduce network traffic", Cloud Computing (CLOUD), 2012 IEEE 5th International Conference, Honolulu, HI, pp. 49-58, **(2012)** June.
[13] M. Hammoud and M. Sakr, "Locality-aware reduce task scheduling for MapReduce", Proceedings of 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom), Athens, Greece, **(2011)**.
[14] T. Y. Chen, H. W. Wei, M. F. Wei, Y. J. Chen, T. S. Hsu and W. K. Shih, "LaSA: A locality-aware scheduling algorithm for Hadoop-MapReduce resource assignment", Collaboration Technologies and Systems (CTS), 2013 International Conference, San Diego, CA, **(2013)** May, pp. 342-346.
[15] S. Seo, I. Jang, K. Woo, I. Kim, J, S. Kim and S. Maeng, "HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment", Cluster Computing and Workshops, CLUSTER'09. IEEE International Conference, New Orleans, LA, **(2009)** August, pp. 1-8.
[16] P. C. Chen, Y. L. Su, J. B. Chang and C. K. Shieh, "Variable-Sized map and locality-aware reduce on public-resource grids", Advances in Grid and Pervasive Computing, Springer Berlin Heidelberg, **(2010)**.
[17] D. DeWitt and M. Stonebraker, "The Database Column", vol. 1, **(2008)**.
[18] D. DeWitt and J. Gray, COMMUN. ACM., vol. 6, no. 35, **(1992)**.
[19] Y. Kwon, M. Balazinska, B. Howe and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions", Proceedings of the 1st ACM symposium on Cloud computing, Indianapolis, Indiana, USA, **(2010)** June.
[20] Y. Kwon, M. Balazinska, B. Howe and J. Rolia, "Open Cirrus Summit", **(2011)**.
[21] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga and D. Gannon, "Cloud technologies for bioinformatics applications", Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers, Oregon, USA, **(2009)** November.
[22] J. Lin, "The curse of zipf and limits to parallelization: a look at the stragglers problem in MapReduce", 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, Boston, USA, **(2009)** July.
[23] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters", Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, Montana, USA, **(2009)** October.
[24] http://blog. csdn. net/hackbuteer1/article/details/7971328.
[25] http://blog. csdn. net/cumirror/article/details/7054496.

