2011 Fourth International Symposium on Parallel Architectures, Algorithms and Programming

Job Scheduling Optimization for Multi-User MapReduce Clusters

Yongcai Tao, Qing Zhang, Lei Shi
School of Information Engineering
Zhengzhou University
Zhengzhou, China
{ieyctao, ieqzhang, shilei}@zzu.edu.cn

Pinhua Chen
Department of Electronics and Information Engineering
Huazhong University of Science and Technology
Wuhan, China
xingkongsoft@163.com

***Abstract*—A shared MapReduce cluster is beneficial to build data warehouse which can be used by multiple users. FAIR scheduler gives each user the illusion of owning a private cluster. Moreover, it can dynamic redistribute capacity unused by some users to other users. However, when reassigning the slots, FAIR picks the most recently launched tasks to kill without considering the job character and data locality, which increases the network traffic while rescheduling the killed Map/Reduce tasks. The paper, based on FAIR scheduling, proposes an improved FAIR scheduling algorithm, which take into account the job character and data locality while killing tasks to make slots for new users. Performance evaluation results demonstrate that the improved FAIR decreases the data movement, speeds the execution of jobs, consequently improving the system performance.**

Keywords-MapReduce; job Scheduling; HDFS; Hadoop;

I. INTRODUCTION

MapReduce breaks a computation job into numerous small Map/Reduce tasks and lets them run on different resource nodes in parallel [1]. Hadoop is an open-source MapReduce runtime implementation [2]. The Hadoop framework is composed of a MapReduce execution runtime and a distributed file system called HDFS (Hadoop Distributed File System) [3]. Today, many software ecosystems have welled up around Hadoop and got widely used in web indexing, data mining, social networking, and scientific simulation [4, 5].

Nowadays, many clusters are deployed with Hadoop (called MapReduce clusters), and are shared by multiple users to run a mix of long batch jobs and short interactive queries over a common data set [6]. Sharing a MapReduce cluster enables users to multiplex computing and storage resources. Zaharia et al.[6] showed a representative use case at Facebook. Event logs from Facebook's website are imported into a Hadoop cluster every hour which runs a data warehouse. The system has been used by over 50 Facebook engineers and runs 3200 MapReduce jobs per day, including analyzing usage patterns to improve site design, detecting spam, data mining and ad optimization. The jobs' running time ranges from several hours to 1-2 minutes.

A shared cluster is beneficial to build data warehouse which can be used by multiple users. However, Native Hadoop's FIFO scheduler is unacceptable for production jobs and made interactive queries impossible among multiple users [2]. To address this problem, Zaharia et al.[6] designed and implemented a fair scheduler for Hadoop. It gives each user the illusion of owning a private cluster, letting users start jobs within seconds and run interactive queries. Moreover, it can dynamic redistribute capacity (slots) unused by some users (jobs) to other users (jobs) to guarantee that each user (job) can be assigned a virtual private cluster.

In practical, different kinds of jobs often simultaneously run in MapReduce framework. These different jobs make different workloads on MapReduce cluster, including I/O-bound and CPU-bound workloads. When reassigning the slots, FAIR picks the most recently launched tasks in over-scheduled jobs to kill to minimize wasted computation. This policy doesn't consider the data locality and job character, so causing large amount of data transferring and increasing network traffic while rescheduling killed Map/Reduce tasks. The paper, based on FAIR scheduling, proposes an improved FAIR scheduling algorithm, which take into account the job character and data locality while killing tasks to make slots for new users. The rational is that when killing tasks, it adopts different policies for the I/O-bound and CPU-bound jobs based on data locality. Performance evaluation results demonstrate that the improved FAIR decreases the data transferring, speeds the execution of jobs, consequently improving the system performance.

The rest of the paper is organized as follows. Section Ⅱ reviews the related work. Section Ⅲ introduces the backgrounds. The improved FAIR scheduling is designed in Section Ⅳ. Section Ⅴ conducts a performance evaluation. Finally, we conclude and give some future work in Section Ⅵ.

II. RLATED WORK

Scheduling on cloud computing systems has attracted many researches in recent years. The default Hadoop scheduler schedules jobs by FIFO where jobs are scheduled sequentially [2]. Ibrahim et al.[7] proposed a novel MapReduce framework on virtual machines, which executes local Reduce tasks within the physical node after all Map tasks in virtual machines have been finished to reduce the amount of outer data transferring. Sangwon et al.[8] proposed two optimization schemes, prefetching and pre-shuffling to improve MapReduce performance. Rafique et al.[9] used double-buffering and asynchronous I/O to support MapReduce functions in clusters with asymmetric components and addresses the I/O bottleneck issue among asymmetric multi-core processors (AMPs). Chao et al.[10] classified MapReduce workloads into three categories based on CPU and I/O utilization and designed a Triple-Queue

978-0-7695-4575-2/11 $26.00 © 2011 IEEE
DOI 10.1109/PAAP.2011.33

IEEE computer society

Scheduler in light of the dynamic MapReduce workload prediction mechanism. Jiahui et al.[11] proposed a heuristic task scheduling algorithm called Balance-Reduce(BAR). By taking a global view, BAR can adjust data locality dynamically according to network state and cluster workload. Hadoop on Demand(HOD) is a management system for provisioning virtual Hadoop clusters over a large physical cluster [12]. It is inefficient that Map tasks need to read input splits across two virtual clusters frequently. Zaharia et al. [13] proposed delay scheduling to enhance both fairness and data locality of jobs in a shared cluster. Zaharia et al. [6] designed and implemented a fair scheduler for Hadoop. It gives each user the illusion of owning a private cluster, letting users start jobs within seconds and run interactive queries. FAIR picks the most recently launched tasks in over-scheduled jobs to kill without considering the data locality and job character, so causing large amount of data transferring and increasing network traffic while rescheduling killed Map/Reduce tasks.

III. BACKGROUNDS

A. Hadoop

Hadoop's two fundamental subprojects are the HDFS and the MapReduce framework [2], its architecture is shown in Figure 1. The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. HDFS has a master/slave architecture. The master server, called NameNode, manages the file system namespace and regulates access to files by clients. In addition, there are a number of slaves, called DataNodes, manage storage attached to the nodes that they run on. The Map/Reduce framework runs on top of HDFS to process distributed large data sets on compute clusters. It consists of a single master JobTracker and one slave TaskTracker per cluster node. The JobTracker is responsible for scheduling the Map and Reduce tasks on the slaves, monitoring them and re-executing the failed tasks. The slaves execute the tasks as directed by the master. The TaskTracker does heartbeat in every interval or when a task in that node finishes.

B. Native Scheduling of Hadoop

Hadoop's scheduler exploits FIFO policy. The drawback of FIFO scheduling is poor response times for short jobs in the presence of large jobs. Hadoop exploits HOD (Hadoop On Demand) to solve the problem [12], which provides private MapReduce clusters over a large physical cluster. HOD lets users share a common filesystem while owning private MapReduce clusters. However, HOD had two problems. (1) Poor Locality: Each private cluster runs on a fixed set of nodes, but HDFS files are divided across all nodes. As a result, some Map tasks must read data over the network, degrading throughput and response time; (2) Poor Utilization: Because each private cluster has a static size, some nodes may be idle. As Hadoop jobs are "elastic," it is difficult to determine the appropriate size of private cluster.

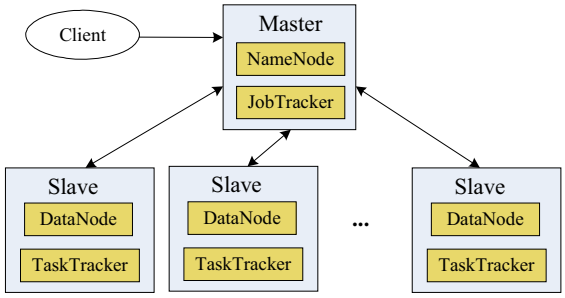

Figure 1. Hadoop Architecture

C. FAIR

To address the problems with HOD, Zaharia et al.[6] developed a fair scheduler (FAIR) for Hadoop. FAIR uses a two-level hierarchy and assigns resources to users in terms of "pool". At the top level, FAIR allocates task slots across pools, and at the second level, each pool allocates its slots among multiple jobs in the pool. Each pool *i* is given a minimum share (number of slots) m_i, which can be zero. The scheduler ensures that each pool will receive its minimum share, and the sum over the minimum shares of all pools does not exceed the system capacity. When a pool is not using its full minimum share, other pools are allowed to use its slots. FAIR exploits two policies to reassign the slots: (1) kill tasks of other pools' jobs, and (2) wait for tasks to finish.

To ensure that each job gets its share, FAIR uses two timeouts, one for guaranteeing the minimum share (T_{min}), and another one for guaranteeing the fair share (T_{fair}), where $T_{min} < T_{fair}$. If a newly started job does not get its minimum share before T_{min} expires, FAIR kills other pools' tasks and re-allocates them to the job. Then, if the job has not achieved its fair share by T_{fair}, FAIR kills more tasks. When selecting tasks to kill, FAIR picks the most recently launched tasks in over-scheduled jobs. However, FAIR doesn't consider the data locality and job character when killing tasks, so causing large amount of data movement and increasing network traffic while rescheduling the killed Map/Reduce tasks.

IV. THE IMPROVED FAIR DESIGN

The improved FAIR scheduler aims to solve two issues. (1) How to select tasks to kill to make slots for new users while guaranteeing the data locality? (2) How to distribute the private data set in shared MapReduce cluster?

A. Classification of Workloads on MapReduce Cluster

In practical, different kinds of jobs often simultaneously run in MapReduce framework. These different jobs make different workloads on MapReduce cluster, including the I/O-bound and CPU-bound workloads. Classifying MapReduce workloads is beneficial to MapReduce scheduling as the usage of I/O and CPU are actually complementary. Based on the [10], the paper proposes a workload classification mechanism on MapReduce cluster. As the Map tasks are only killed while reassigning the slots, the paper only considers the Map task while classifying the

workloads. The Map task execution can be decomposed into three actions: 1) initializing input data; 2) computing Map task; 3) storing output result to local disk. The symbols used in the paper are listed in Table 1.

TABLE I. SYMBOL DEFINITION

Symbol	Meaning
MIN	*Map input data*
MOD	*Map output data*
λ	*ratio parameter*
MTCT	*Map task completion time*
DIOR	*disk I/O rate*
n	*number of Map tasks*
SN	*assigned number of slots*
NSmin	*minimum share number of slots*

The ratio (λ) of the amount of Map output data (MOD) and Map input data (MID) in a single Map task depends on the type of workload as shown in Formula 1.

$$MOD = \lambda * MID \qquad \lambda > 0 \tag{1}$$

Every node can be assigned *n* Map tasks which are synchronously running and share the disk I/O bandwidth when the system stably runs. So, if the data transferring time of MIN and MOD is larger than the completion time of Map tasks, this kind of job is I/O-bound. In this type of job, every Map task would generate lots of I/O operations, and when *n* Map tasks synchronously run on every node, there will be contention in I/O among different tasks. Formula 2 witnesses this conclusion.

$$\frac{n*(MID+MOD)}{DIOR} > MTCT \tag{2}$$

According to Formulas 1 and 2, we can obtain:

$$\frac{n*(1+\lambda)MID}{DIOR} > MTCT \tag{3}$$

The second class of workload is different with the former one. Its data transferring time of MIN and MOD is less than the completion time of Map tasks. As shown in Formula 3, this kind of job is CPU-bound.

$$\frac{n*(MID+MOD)}{DIOR} \le MTCT \tag{4}$$

According to Formulas 1 and 3, we can obtain:

$$\frac{n*(1+\lambda)MID}{DIOR} \le MTCT \tag{5}$$

In real environment, the MTCT may be infected by the nodes' workload runtime status. The workload on a node may make the MTCT longer than the ideal value. This means that if a job is defined as an I/O-bound class according to the above Formals, it is definitely I/O-bound. But if the job is defined as a CPU-bound class, it may be not CPU-bound class. A solution proposed in [10] is to run a benchmark job simultaneously together with the testing jobs, if the execution time of benchmark job doesn't get longer, it is confirmed that the testing job is CPU-bound.

B. The Improved FAIR Scheduling Algorithms

The improved FAIR scheduler takes into account the job character and data locality while killing tasks. The rational is that when killing tasks, it adopts different policies for I/O-bound and CPU-bound jobs based on data locality. (1) For I/O-bound jobs, the improved FAIR kills the tasks with data not closest to the slave (on a remote rack if possible, otherwise on the same rack, finally on the same node), trying to decrease the network traffic; (2) For CPU-bound jobs, the improved FAIR kills the most recently launched tasks in over-scheduled jobs to minimize the wasted computation.

The improved FAIR scheduling pseudocodes are shown in Algorithms 1 and 2. Algorithms 1 and 2 only describe the case that how to select tasks to kill if a newly started job does not get its minimum share before T_{min} expires. The case of getting job's fair share is similar to the one of getting minimum share and is omitted for the length of paper.

Algorithm 1: The improved FAIR scheduling for I/O-bound job
Input: scheduled Map tasks;
Output: killed Map tasks, scheduled data blocks;

```
1.  while (SN < NS_min) do
2.      for each scheduled Map tasks do
3.          if (its data ∈ other rack & its node ⊃ new
                Map data) do
4.              task → killed, SN++;
5.          elseif (its data ∈ local rack & its node ⊃
                new Map data) do
6.              task → killed, SN++;
7.          elseif (its data ∈ local node & its node ⊃
                new Map data ) do
8.              task → killed, SN++;
9.          elseif (its data ∈ other rack) do
10.             task → killed, SN++;
11.         elseif (its data ∈ local rack) do
12.             task → killed, SN++;
13.         elseif (its data ∈ local node) do
14.             task → killed, SN++;
15.         endif
16.     endfor
17. endwhile
18. if (user has private data set) do
19.     distribute the private data in shared
        MapReduce cluster based data-locality;
20. endif
```

As shown in Algorithm 1, for I/O-bound jobs, first, the improved FAIR kills the tasks with data on other rack and its node has the new Map task data (lines 3-4). Second, the scheduler kills the tasks with data on local rack and its node has the new Map task data (lines 5-6). Third, the scheduler kills the tasks with data on local node and its node has the new Map task data (lines 7-8). Lines 9-15 consider the case that the node has not the new Map data and its idea is similar to lines 3-8. Finally, if user has private data set, the scheduler distributes the private data in shared MapReduce cluster

based data-locality (lines 18-20). Namely, the scheduler distributes one replication of the data block to the killed node, and two replications to the other nodes. For I/O-bound jobs, the improved FAIR kills the tasks with data not closest to the slave (on a remote rack if possible, otherwise on the same rack, finally on the same node). The reason is that if the killed job's data is on local node, large amount of data will be transferred while rescheduling killed Map/Reduce tasks, which increases network traffic.

Algorithm 2 shows the improved FAIR scheduling policy for the CPU-bound job. First, the improved FAIR kills the most recently launched task and its data is on the other rack (lines 3-4). Second, the scheduler kills the most recently launched task and its data is on the local rack (lines 5-6). Third, the scheduler kills the most recently launched task and its data is on the local node (lines 7-8). Finally, if user has private data set, the scheduler distributes the private data in shared MapReduce cluster based data-locality (lines 12-14).

Algorithm 2: The improved FAIR scheduling for CPU-bound job
Input: scheduled Map tasks;
Output: killed Map tasks, scheduled data blocks;
1. **while** ($SN < NS_{min}$) **do**
2. | **for** each scheduled Map tasks **do**
3. | | **if** (∈ the most recently launched task & its data ∈ other rack) **do**
4. | | | task → killed, SN++;
5. | | **elseif** (∈ the most recently launched task & its data ∈ local rack) **do**
6. | | | task → killed, SN++;
7. | | **elseif** (∈ the most recently launched task & its data ∈ local node) **do**
8. | | | task → killed, SN++;
9. | | **endif**
10. | **endfor**
11. **endwhile**
12. **if** (user has private data set) **do**
13. | distribute the private data in shared MapReduce cluster based data-locality;
14. **endif**

V. Performance Evaluations

A. Experimental Environments

The experimental environment comprises three groups of nodes. The first is a cluster which contains 4 nodes (2 Intel Xeon Quad-Core 2.0GHz CPU, 4GB memory, 500GB disk). The second consists of 4 nodes (Intel Celeron dual-core 2.0GHz CPU, 2GB memory, 400GB disk). The third consists of 4 nodes (Intel Celeron dual-core 2.0GHz CPU, 2GB memory, 500GB disk). Since the nodes within the same group share the same switch, each group can be seen as one rack.

B. Benchmarks

We use a testing program to get the DIOR value. This program is simply to read and write on the disk. In experiment, the DIOR is almost 32.4 MB/S and the Map/Reduce slots are set to be 8. So, the n of Formula is 8. The experiments use three widely used benchmarks, "TeraSort", "GrepCount" and "WordCount", which belong to the three different workload types. (1) TeraSort: It is a famous total order sort benchmark and a sequential I/O-bound job. Its MID is 64M, MOD is almost 64M in average. The MTCT is 9 seconds. (2) GrepCount: The GrepCount program accepts a regular expression and the files as the input parameters. It outputs the occupation number of the lines matching the input regular expression in the documents rather than all lines matching the regular expression. In our test case, we use (*.) as the regular expression. Its MID is 64M and MOD is almost 0.82M in average. The MTCT is 102 seconds. According to Formula 3, GrepCount belongs to the CPU-bound job. (3) WordCount: It counts the frequency of occurrence for each word in a set of files. Its MID is 64M and MOD is almost 64M in average. The MTCT is 46 seconds. According to Formula 3, WordCount belongs to the Sway job.

C. Experimental Results

We choose the input data set as 10GB for each job to simulate the situation in a real product environment. Each block is 64M. First, we submit TeraSort and GrepCount jobs which occupy all nodes' slots and each job can be assigned 48 slots. Then, we submit WordCount job. The improved FAIR scheduler assigns it the fair share slots (32).

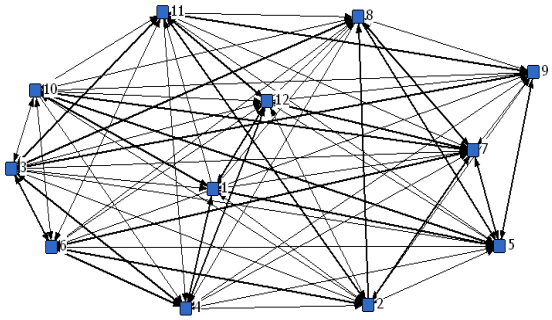

Figure 2. Data Movement in FAIR

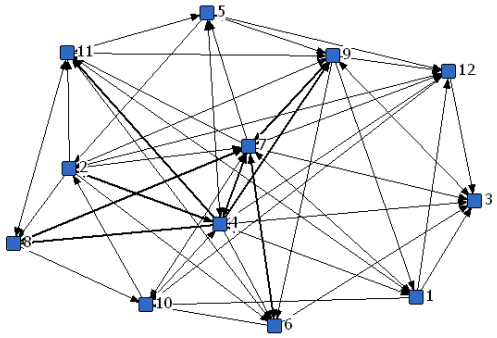

Figure 3. Data Movement in the Improved FAIR

Figures 2 and 3 show the comparison of data movement of above three jobs in FAIR and the improved FAIR

schedulers. It can be seen from the Figures 2 and 3 that the data movement of FAIR is larger than the one of the improved FAIR. The reason is that native FAIR picks the most recently launched tasks to kill without considering the job character and data locality. The improved FAIR, while killing tasks, adopts different policies for I/O-bound and CPU-bound jobs based on data locality. For I/O-bound job, it tries to kill the tasks with data not closest to the slave and its nodes have the new job's data, which decreases the data transferring. Figures 4 and 5 show the comparison of job completion time. It can be concluded that the job completion time in the improved FAIR is better than the one of FAIR. This is because that compared with FAIR, the improved FAIR fully considers the job character while killing the tasks. The improved FAIR not only decreases the data transferring and speeds up job execution, but also minimizes the wasted computation.

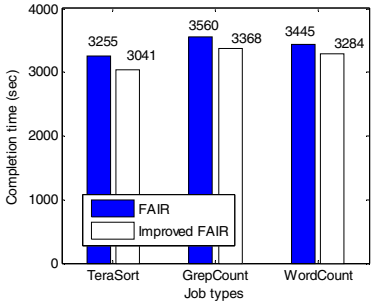

Figure 4. Comparison of Job Completion Time (input data set is 10GB)

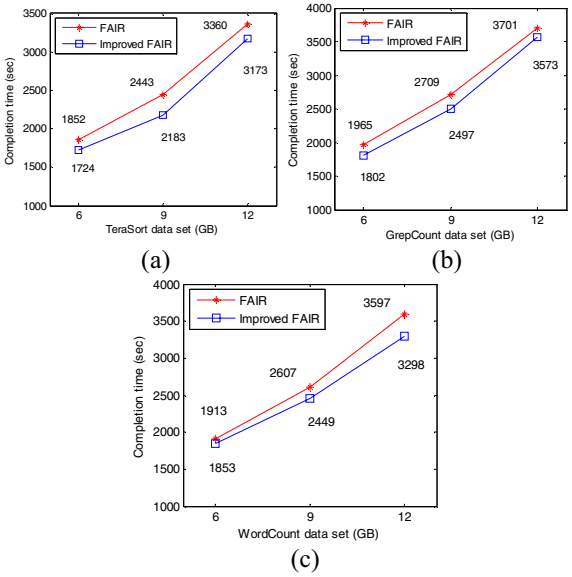

Figure 5. Comparison of Job Completion Time

VI. Conclusion and Future Work

Network bandwidth is a relatively scarce resource in cloud. To conserve network bandwidth, the improved FAIR proposed in the paper takes into account the job character and data locality while killing tasks to make slots for new users. Performance evaluation results demonstrate that the improved FAIR decreases the data movement, speeds the execution of jobs, so improving the system performance. Current and future research efforts include further bettering the improved FAIR scheduler in reliability, fault-tolerance ability and manageability.

References

[1] J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Communications of the ACM, 2008, 51(1):1-13.

[2] Hadoop, http://hadoop.apache.org/.

[3] D. Borthakur, The Hadoop Distributed File System: Architecture and Design. The Apache Software Foundation, 2007.

[4] Facebook, http://www.facebook.com

[5] Amazon Simple Storage Service, http://aws.amazon.com/s3/

[6] M. Zaharia, D. Borthakur, J. Sarma, et al., "Job scheduling for multi-user mapreduce clusters," EECS Department University of California Berkeley Tech Rep UCBEECS200955 Apr, 2009–55. EECS Department, University of California, Berkeley. Retrieved from http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-55.pdf

[7] S. Ibrahim, H. Jin, B. Cheng, et al., "Cloudlet: Towards MapReduce implementation on Virtual machines," In Proc. of 18th ACM International Symposium on High Performance Distributed Computing, USA, 2009, pp.65–66..

[8] S. Sangwon, J. Ingook, W. Kyungchang, et al., "Hpmr: Prefetching and pre-shuffling in shared mapreduce computation environment," In Proc. of the 2009 IEEE Cluster Computing, USA, 2009, pp.1-8.

[9] M. Rafique, B. Rose, A. Butt, et al., "Supporting mapreduce on large-scale asymmetric multi-core clusters," In ACM Operating Systems Review, 2009, 43(2):25–34.

[10] T. Chao, H. Zhou, Y. He, et al., "A Dynamic MapReduce Scheduler for Heterogeneous Workloads," In Proc. of the 8th International Conference on Grid and Cooperative Computing, China, 2009, pp.218-224.

[11] J. Jiahui, L. Junzhou, S. Aibo, et al., "Bar: an efficient data locality driven task scheduling algorithm for cloud computing," In Proc. Of 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), USA, 2011, pp.295-304.

[12] Hadoop on demand. [Online]. Available: http://hadoop.apache.org/common/docs/r0.18.3/hod.html.

[13] M. Zaharia, D. Borthakur, J. Sen Sarma, et al., "Delay Scheduling : A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling," In Proc. Of the 5th European Conference on Computer Systems(EuroSys), USA, 2010, pp.265-278.