Partial Task Shuffle First Strategy for Spark

Tianlei Zhou[1, 2, a], Yuyang Wang[1, 2, b]

[1]School of computer science and technology, Chongqing University of Posts and Telecommunications, Chongqing, 400065, China

[2]Chongqing Engineering Research Center of Mobile Internet Data Application, Chongqing, 400065, China

[a]ztl_wangyi@163.com, [b]529504153@qq.com

Keywords: Big data, spark, shuffle, task

Abstract: Apache Spark is an in-memory distributed computing framework, which is more suitable for iterative jobs than MapReduce. However, the shuffle process needs to synchronize tasks between nodes, which may lead to waste the computing resources of the cluster and ultimately reduce the computing performance of the cluster. This is an important reason to limit the performance of Spark. In this paper, we proposes a Partial Task Shuffle First (PTSF) Strategy to dynamically generate Shuffle Write tasks and perform Shuffle operations on partial completed tasks. The strategy increases the parallel degrees of data calculation and transmission, lowering the peak of the Shuffle stage, allowing the cluster to be more balanced in the course of the operation. Finally, experiments show that the proposed strategy can improve Shuffle execution efficiency.

1. Introduction

Spark [1,2] solves the inefficiency of MapReduce [3] in dealing with iterative jobs through memory computation, is 100 times faster than MapReduce, and has been widely used by companies and organizations. For example, some shopping websites need to aggregated, analyze and mine the log data of their users' behaviors, and finally provide data to support their various recommendation systems and search systems. In this process, including ETL, SQL query analysis and machine learning, Spark is gradually replacing MapReduce as its mainstream computing engine for big data processing. In addition, Alibaba, Facebook, Tencent and other Internet companies have also applied Spark to their multiple business platforms.

Resilient Distributed Datasets [4] (RDD) is Spark's core abstract data structure. An RDD is simply a distributed collection of element. Resilient means that RDD allow data to be automatically switched between memory and disk. Distributed means that each RDD is split into multiple partitions [5], which may be computed on different nodes of the cluster. Spark provides two types of RDD operations: Transformation and Action. Transformations are operations on RDDs that return a new RDD, such as map and filter. Actions are operations that return a result to the driver program or write it to storage, and kick off a computation, such as count and first. It should be noted that RDD can only be created by reading input data or performing Action operations on existing RDDs. Therefore, during the calculation process, RDD will gradually form a chain dependent structure. In this chain-dependent structure, if the partition in the child RDD depends on each partition in the parent RDD, this dependency is called a wide dependency. Wide dependencies will trigger shuffle, which is a process of redistributing data across partitions (aka repartitioning) that may or may not cause moving data across JVM processes or even over the wire between executors on separate machines.

The Shuffle operation involves disk and network I/O. As we all know, disk and network I/O take much more time than in-memory computing, so the performance of the Shuffle directly affects the efficiency of the entire program. Therefore, to optimize the Shuffle process and reduce the time spent in the Shuffle is the key to improve the performance of the entire computing framework.

2. Shuffle Optimization

2.1. Shuffle Process

As a big data computing framework, Spark is not responsible for data storage and usually uses HDFS as the input source for its tasks. In HDFS [6], the file is split into blocks and stored in the DataNode. "Passing data is not as good as passing computation" is Spark's design philosophy, so when performing jobs, after reading input data through HDFS to create RDD, computing tasks are assigned to nodes where data blocks are located. As shown in Figure 1, solid line rounded corner box is RDD, solid line square corner box is a partition of RDD, and dotted line box is Stage. After the job is submitted, the input data is first read from HDFS and formatted as RDD (A, C, E), followed by GroupBy operation on A, Map operation on C, and Union operation on E. Where the GroupBy operation triggers a Shuffle.

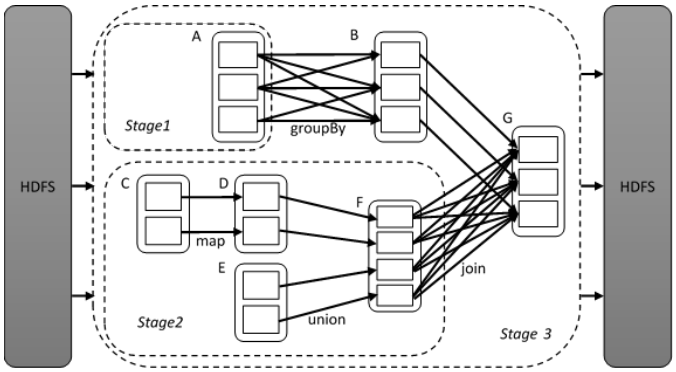

Figure 1. The job execution process in spark.

Shuffle is the reorganization of data. Figure 2 shows the shuffle process. In Spark, it is divided into two phases: Shuffle Write and Shuffle Read. In earlier versions, each Map task created R bucket caches, R was the number of Reduce tasks. In the Shuffle Write stage, the Map task is first read to output data. Then the Partitioner is used to calculate the location of each data partition. According to the partition location, the data is written to the corresponding Bucket cache and finally the data cached by the Bucket is written to the disk. In the Shuffle Read phase, the Reduce task uses BlockStoreShuffleFetcher to get the data information, and then pulls the data through the BlockManager. The pulled data is first put into the memory, and if the memory is not enough, it is put into the disk. However, this method will generate M*R Block files. For example, when the number of Map task is 128 in a node and the number of Reduce task is 128, the node will generate 16,384 Block files. A large number of random I/O will exert great pressure on the file system and eventually Reduce spark execution efficiency. Therefore, in later versions, Spark introduced a file merging mechanism. Executor merges the data of Map tasks running on the same core, reducing the number of files and reducing the pressure of random reading and writing.

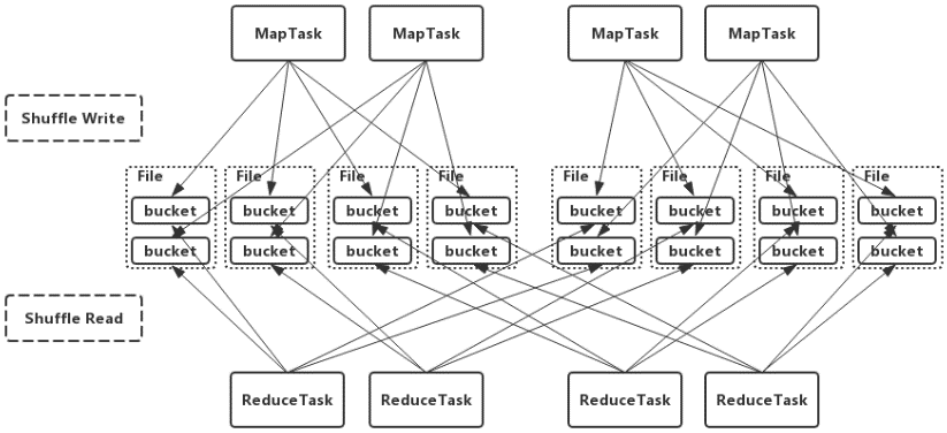

Figure 2. The shuffle process in spark.

2.2. Problem Analysis

As described in the previous section, the second phase of the Shuffle process, the Shuffle Read process, requires pulling data from all the nodes in the cluster. It should be noted that the data pull operation of the Reduce task needs to wait for all Map tasks to complete before it starts. In other words, the Shuffle Read process involves synchronizing the computing tasks of each node in the cluster.

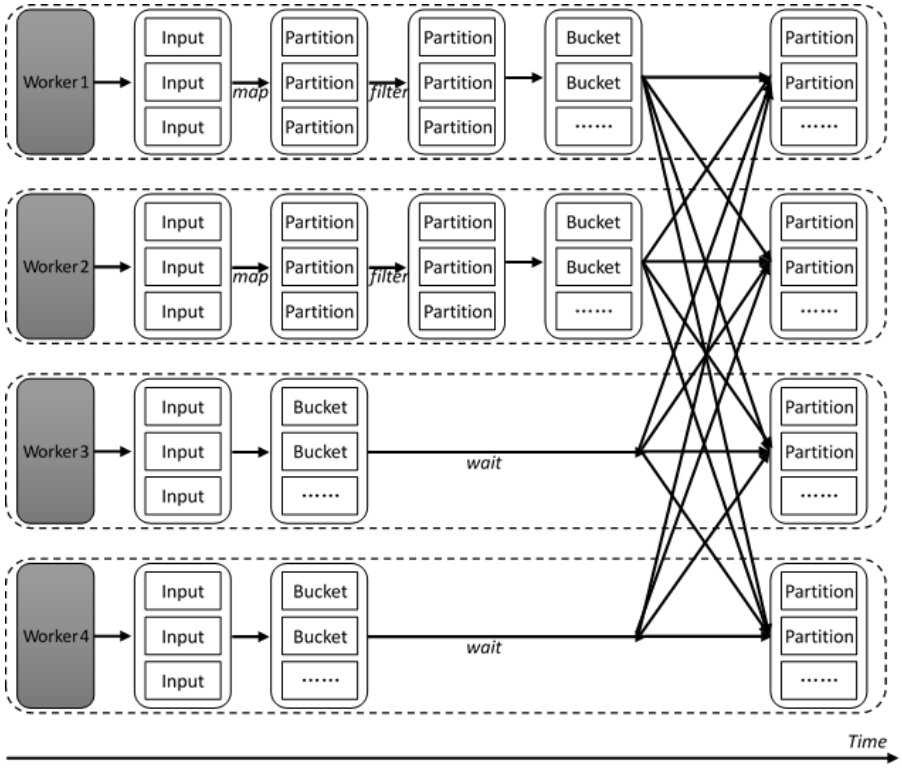

Figure 3. Task allocation of original spark

As shown in figure 3, each node performs different task jobs. Worker1 and Worker2 first create RDD by reading input data, then perform Map and Filter calculations, and finally perform Shuffle Write; while Worker3 and Worker4 only need to create RDD by reading the input dataset to perform Shuffle Write. Therefore, before Worker1 and Worker2 are written to the Bucket, Worker3 and Worker4 are idle pending due to the small number of computation tasks. It can be concluded from the above analysis that in spark, computing process and data transmission process are processed serially, which will lead to the waste of cluster resources. In addition, if Shuffle Read is performed simultaneously after all nodes are synchronized, a lot of data transmission is concentrated at the same time point, which will bring greater pressure on the network load. Therefore, properly adjusting the time when the Reduce task pulls data can improve Shuffle efficiency.

2.3. Partial Task Shuffle First Strategy

In response to the above problems, this section proposes a Partial Task Shuffle First strategy. Different from the original Shuffle process, a Map task does not perform a Shuffle Write immediately after execution. Instead, it processes data of multiple Map tasks and achieves the effect of merging Block files, and finally outputs a group of Block files. Since the intermediate data transmission does not need to consume computing resources, once the partial data Shuffle Write is executed, the output intermediate data is pushed to other nodes of the cluster, thereby increasing the parallelism between data transmission and calculation.

First, we need to start the monitoring module of each Worker node in the computing cluster, which is responsible for monitoring the running state of the task on its own node, and is used to predict the time required for the completion of the Map task on this node and the time required by the data Shuffle Write.

Table.1. Monitoring module acquires data content

Symbol	description
SM_{input_i}	Input data size of the i-th Map task
TM_i	Execution time of the i-th Map task
SM_{output_i}	Output data size of the i-th Map task
TS_k	Execution time of the k-th completed Shuffle Write task
SS_{input_k}	Input data size of the k-th completed Shuffle Write task
$TotalCore$	Executor Core Count
$MapCore$	Executor Map Core Count
$ShuffleCore$	Executor Shuffle Core Count

Second, PTSF needs to initialize the following data:

Un-shuffled Set. It is used to store the output result of the completed Map task, that is, the data set without Shuffle Write. Executor puts the task output into the pending result set if it performs a Map task. If a Shuffle Write task starts, it will read the collection and empty the collection. The initial state is empty.

Map Set. The remaining tasks are predicted by the completed task information. In the initial state, there are fewer completed Map tasks, and the prediction bias may be large. Therefore, it is necessary to store the completed Map task operation information, which is used to dynamically adjust the prediction result, reduce the error, and the initial state is empty.

Shuffled Set. Similar to the completed Map task collection. Stores the information of the completed Shuffle Write task, and the initial state is empty.

We need to calculate the execution speed of the node Map task to predict the execution time required by the Map task, as shown in equation 1.

$$V_{map} = \frac{1}{|\alpha|}\sum_{i}^{i\in\alpha}\frac{SM_{input_i}}{TM_i} \tag{1}$$

In equation 1, α is the Map Set. Similarly, the execution speed of the node Shuffle Write task can be calculated, as shown in equation 2.

$$\mathrm{V_{shuffle}} = \frac{1}{|\beta|}\sum_{\mathrm{k}}^{\mathrm{k}\in\beta}\frac{\mathrm{SS_{iutput_k}}}{\mathrm{TS_k}} \tag{2}$$

In equation 2,β is the Shuffled Set. Further, the execution time required for the unfinished Map tasks can be predicted by Equation 3

$$T_{map\_remain} = \frac{\sum_{i}^{i\notin\alpha} SM_{input_i}}{V_{map}} \tag{3}$$

The time required to perform a Shuffle Write operation on the data output by the Map task can be obtained by Equation 4

$$T_{shuffle\_remain} = \frac{\sum_{i}^{i\in\alpha} SM_{output_i} + \lambda\times\sum_{i}^{i\notin\alpha} SM_{input_i} - \sum_{k}^{k\in\beta} SS_{iutput_k}}{V_{shuffle}} \tag{4}$$

Where λ is the input-output conversion factor, which can be calculated by equation (5)

$$\lambda = \frac{\sum_{i}^{i\in\alpha} SM_{output_i}}{\sum_{i}^{i\in\alpha} SM_{input_i}} \tag{5}$$

Finally, we perform the shuffle process through the following algorithm.

Algorithm1: PTSF algorithm

Input: MapCore; ShuffleCore; Un-shuffled Set; Map Set; Shuffled Set

Output: Blocks

1: maps= List< Map>

2: while maps not empty do

```
3:    if map in maps is finished then
4:        maps.remove(map)
5:        MapQueue.add(map,TM_i, SM_input_i)
6:        unshuffleSet.add(M_output_i)
7:        T_map_remain=compute(MapQueue)
8:        T_shuffle_remain=compute(ShuffleQueue)
9:        if T_shuffle_remain/T_map_remain>ShuffleCore/MapCore than
10:           dataset=unshuffleSet
11:           clear(unshuffleSet)
12:           shuffleWrite(dataset)
13:           ShuffleQueue.add(shuffleWrite, TS_k, S_input_k)
14:       else
15:           waitOtherTask( )
16:       end if
17:   end if
```

Obtain all the Map tasks on the node, and when one of the Map tasks is executed, record the running information and output result information of the task. Then, the completed task is used to predict the time that the incomplete Map task and the Shuffle Write task will be consumed, and finally the computing resources are dynamically allocated for the two tasks. It should be noted that the initial state of the Shuffle Write task set is empty, that is, there is no completed Shuffle Write task information, so it is necessary to start extracting a small portion of the Map task output data as a sample, and run a smaller Shuffle Write task instance. The execution effect of the instance is used as the initial data for prediction.

3. Experiment

In order to verify the performance of PTSF, the cluster built in this paper uses 7 servers, one server is the master node, and the other six servers are the worker nodes. Each server configuration is as described in Table 2:

Table.2. Experiment Configurations

Parameters	Values
CPU	Intel Xeon/16Core 2.8Ghz
RAM	8GB
DISK	150G
OS	CentOS 6.9
JDK	JDK1.8.0_102
SCALA	Scala-2.11.8
Hadoop	Apache Hadoop 2.7.3
Spark	Apache Spark 2.1.0

Experimental data set to select the SNAP (Stanford Network Analysis Project) provides four large Network data set, take a PageRank algorithm in each data set and PageRank contains several rounds of iteration, each iteration involves two Shuffle (Join and ReduceByKey) operation, and thus more conducive to verify the result of the experiment.

Table.3. Experiment Dataset

Dataset	Nodes	Edges	Description
Amazon0601	403394	3387388	Amazon product co-purchasing network from June 1 2003
wiki-Talk	2394385	5021410	Wikipedia talk (communication) network
soc-pokec-relationships	1632803	30622564	Pokec online social network
soc-LiveJournal1	4847571	68993773	LiveJournal online social network

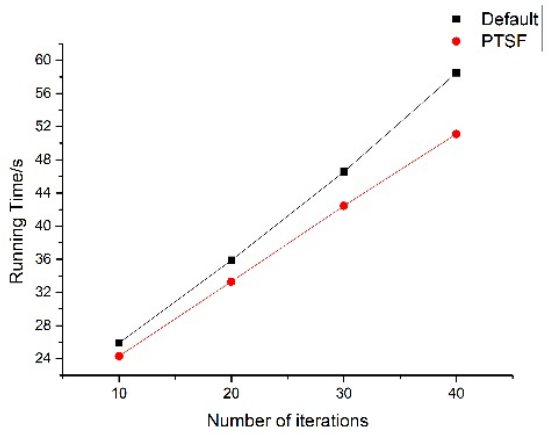

Figure 4. Amazon0601

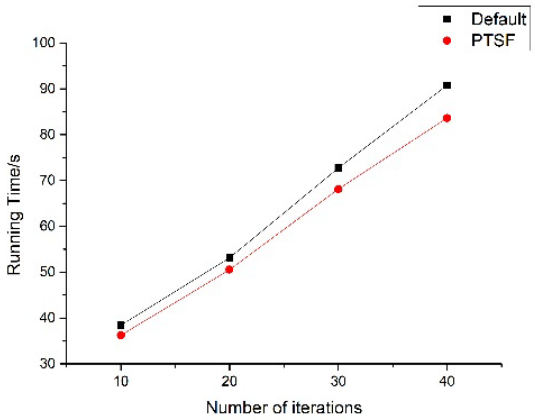

Figure 5. wiki-Talk

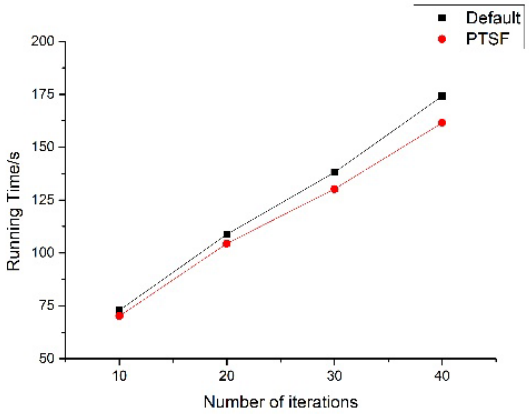

Figure 6. Soc-pokec-relationships

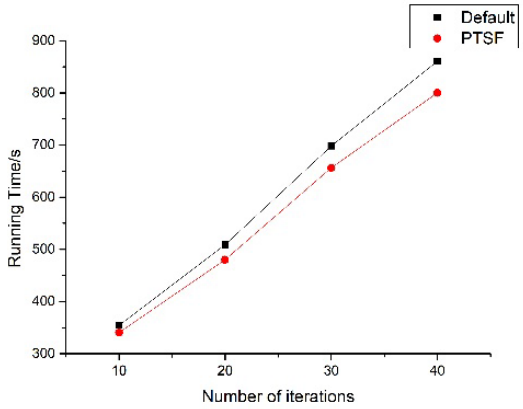

Figure 7. Soc-LiveJournal1

First, we use PageRank iterations 10 to 40 times on these four datasets under the native Spark platform and optimized platform. In the course of the experiment, in order to reduce the error, the results of each experiment are the average of the results of multiple experiments. As can be seen from the results shown in Figures 4 to 7, the job execution time is increasing because the number of iterations is increased (the number of Shuffles is increased). On the other hand, under the four data sets, the Shuffle efficiency has improved, which also reflects the applicability of the strategy in different data scenarios.

PTSF can use the waiting time of node synchronization to exchange data to reduce the network congestion when synchronizing Shuffle under the native policy. As the number of shuffles in PageRank is more, it is not conducive to the analysis and comparison of network state. Therefore, the word frequency statistical algorithm (WordCount) containing only a Shuffle operation was selected in the second experiment, and 1G,2G and 4G data sets were used for verification respectively. The experimental results are shown in figure 8.

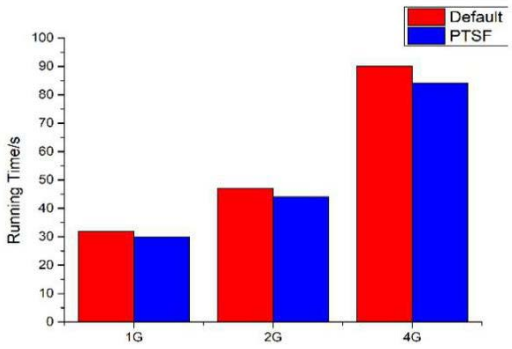

Figure 8. Word Count

As shown in figure 8, PTSF has a certain improvement in execution efficiency compared with the native policy under different data set sizes. In the calculation, further monitoring the network traffic through the nload tool.

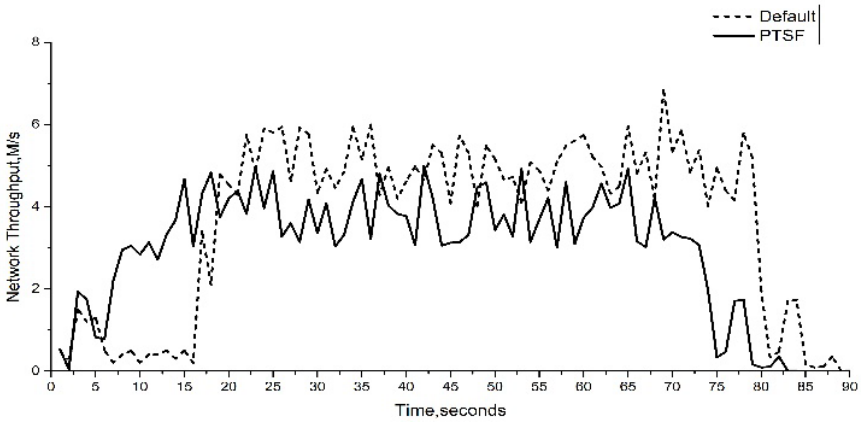

Figure 9. Network throughput of 4G Data

As shown in figure 9, when native Spark performs jobs, the load on the cluster network in the beginning and end stages is low, while the load on the Shuffle phase is high. PTSF makes the network load more balanced.

4. Conclusion

PTSF proposed in this paper can improve the parallelism of data calculation and network transmission. It can also reduce the peak value of network transmission in the Shuffle phase by utilizing the synchronous waiting period between nodes, so that the overall network load is balanced. Experiments show that the strategy proposed in this paper can better improve the efficiency of Shuffle execution. Further, in the heterogeneous Spark cluster, due to the differences in node performance, slow nodes may slow down the overall progress of the job, which can also be continued in the future.

References

[1] Zaharia M, Xin R S, Wendell P, et al. Apache Spark: a unified engine for big data processing [J]. Communications of the Acm, 2016, 59 (11):56-65.

[2] Apache Spark [EB/OL]. http://spark.apache.org/

[3] Dean J, Ghemawat S. MapReduce: A Flexible Data Processing Tool [J]. Communications of the Acm, 2010, 53 (1):72-77.

[4] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing [C]// Usenix Conference on Networked Systems Design & Implementation. 2012.

[5] Ramakrishnan S R, Swart G, Urmanov A. Balancing reducer skew in MapReduce workloads using progressive sampling [C]// Acm Symposium on Cloud Computing. 2012.

[6] Shvachko K, Kuang H, Radia S, et al. The Hadoop Distributed File System [C]// IEEE Symposium on Mass Storage Systems & Technologies. 2010.