 CMC, vol.55, no.3, pp.405-417, 2018

A Spark Scheduling Strategy for Heterogeneous Cluster

Xuewen Zhang[1], Zhonghao Li[1] , Gongshen Liu[1, *], Jiajun Xu[1], Tiankai Xie[2] and Jan Pan Nees[1]

Abstract: As a main distributed computing system, Spark has been used to solve problems with more and more complex tasks. However, the native scheduling strategy of Spark assumes it works on a homogenized cluster, which is not so effective when it comes to heterogeneous cluster. The aim of this study is looking for a more effective strategy to schedule tasks and adding it to the source code of Spark. After investigating Spark scheduling principles and mechanisms, we developed a stratifying algorithm and a node scheduling algorithm is proposed in this paper to optimize the native scheduling strategy of Spark. In this new strategy, the static level of nodes is calculated, the dynamic factors such as the length of running tasks, and CPU usage of work nodes are considered comprehensively. And through a series of comparative experiments in alienation cluster, the new strategy costs less running time and lower CPU usage rate than the original Spark strategy, which verifies that the new schedule strategy is more effective one.

Keywords: Spark, optimize scheduling, stratifying algorithm, performance optimization.

1 Introduction

The born of Spark [Zaharia, Chowdhur, Franklin et al. (2010)] has given possibility to upgrading distributed computing on computer clusters, improving obviously in memory R/W and implement efficiency compared with earlier Hadoop [Zaharia, Chowdhur, Das et al. (2011a, 2012b)]. But, as a general platform, Spark still has some problems in producing process, waiting for optimization.

For performance, in dealing with the underlying task scheduling, Spark does not consider two special impact factors hidden in actual production environment. One is the complexity of the task itself, the other one is heterogeneity of the cluster environment. The former will lead to bottleneck of a single point, resulting in localized distribution of tasks with high complexity, which will result in restrictions of overall efficiency of the cluster by single point blocking, making smooth completion of hydration operations unfeasible. The latter will cause that the cluster cannot fully play the potential of all computing nodes, that is, limited by the cluster platform, the advantage of high computing performance nodes is not able to be played, resulting in wasting of the efficiency upgrades brought by hardware. The above two, one is the inefficient

[1] School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai, 200240, P.R.C.

[2] Eberly College of Science, Pennsylvania State University, Old Main, State College, PA 16801, U.S.A.

[*] Corresponding Author: Gongshen Liu. Email: lgshen@sjtu.edu.cn.

performance of Spark in the time dimension, the other is the limited performance in the hardware dimension. To solve problems above, in this paper, a new scheduling strategy is proposed for Heterogeneous cluster in Spark, looking forward to improve efficient of the whole system.

In this paper, we will reveal some analyses of original Spark scheduling strategy with summary of the existing researches in distributed cluster heterogeneity in Section 2. In Section 3, our new strategy will be described in detail. Section 4 presents the feasibility and evaluation of the strategy by some experimental verification. In Section 5, we conclude the program and propose several future research directions.

2 Related works

Because Hadoop technology is more mature and wider and longer used in the enterprise, the optimization schemes for cluster heterogeneity of distributed computing platform are mainly focused on Hadoop platform. Many enterprises seek ways to use big data efficiently and effectively, which needs some new optimization algorithm proposed. In 2005, Nightingale et al. [Nightingale, Chen and Flinn (2005)] proposed the Speculative Task Execution Strategy (STES), the idea is that when free nodes appear in the cluster, they silently predict the implementation of the local copy of other nodes' tasks, thereby enhancing the utilization of the nodes to avoid unnecessary CPU-vacancy. And it saves snapshot before execution to keep track of recovery when the prediction fails. In 2008, Zaharia et al. [Zaharia, Konwinski, Joseph et al. (2008)] proposed Improved STES, which replaced task progress rate with expected completion time of the task as a scheduling basis. Zaharia et al. [Zaharia, Borthakur, Sen et al. (2010)] proposed the strategy of Delay Scheduling, and developed a multi-user FAIR Scheduler, which solves the problem of fairness of multi users' resource allocation and the problem of single queue blocking. The Spark source FAIR Scheduler implementation comes from this. In 2013, Tang et al. [Tang, Zhou, Li et al. (2013)] proposed the MapReduce Task Scheduler for Deadline (MTSD). MTSD first divides all the nodes in the heterogeneous cluster into multiple levels according to the computing power, and then estimates the duration of the tasks on different levels, distinguish Map tasks and Reduce tasks, and then make the completion time and task deadline as a basis for scheduling. In 2014, Xu et al. [Xu, Cao and Wang (2016)] proposed Adaptive Task Scheduling on Dynamic Workload Adjustment (ATSDWA), which deploys a monitoring module at each node to collect the node's resource usage and execution time. Each node initiates the maximum number of tasks according to the hardware configuration, and feeds back by Task Scheduler when each heartbeat packet transmits, then dynamically adjusts the maximum number of operable tasks of each node according to the real-time monitoring information, thus realizing dynamic adaptation of Hadoop Scheduling.

In Spark field, Yang et al. [Yang, Zheng and Wang (2016)] put forward Heterogeneous Spark Adaptive Task Scheduling (HSATS). In the similar case to ATSDWA, the monitoring module is added to the Worker nodes, and an algorithm to dynamically adjust weights of the nodes is added in TaskScheduler, to schedule nodes according to their real-time weights. And Kaur et al. proposed a fuzzy approach for the employee performance, which provides the idea of evaluating the node performance in this paper.

In summary, the developing process of solution for heterogeneous cluster scheduling is roughly from single queue to multi queue, static allocation to dynamic adjustment, sequential execution to forecast execution, and on this basis, scheduling feedback, residual time estimation, cluster partition and other ideas are constantly excavated and optimized.

3 A new spark scheduling engine

Before the researches are more concerned about how to achieve the balanced distribution of heterogeneous cluster resources, that is, through scheduling strategies to make heterogeneous clusters in the resource utilization level regarded as isomorphism. This paper hopes to make full use of cluster heterogeneity on this basis, in order to improve cluster efficiency.

Storage system often improve the efficiency of the overall system with multi-level cache, the characteristics of the cache are high efficiency of single point and small capacity. This is the usage of the storage media heterogeneity, to promote efficiency of the overall system through the introduction of high-performance heterogeneous media. Similarly, the current distributed cluster is still in a single-tier structure, and the calculating components in fact are similar to the storage components, whose performance continues to increase. As a result, the single-tier cluster structure is capable to be optimized. Therefore, the heterogeneous cluster optimization ideas in this paper are to stratify the cluster, to optimize overall efficiency. In the practical application scenario, it can be improvable to efficiency of the whole distributed system that a small number of high speed components are introduced into the cluster as high-performance nodes. The key problem to the strategy exists in how to layer the cluster and distribute the tasks with different complexity on demand to all levels of nodes.

3.1 Task identification

After releasing a Job, Spark will divide it into several Stages, and further divide the Stages into multiple Tasks, according to the dependencies between each other. Task is the basic unit of the final scheduling. This part will analyze the feasibility of task identification from a micro perspective.

First of all, when the submitted program calls Transformation or Action interface in Spark, SparkContext checks the incoming closures to ensure that they are transferred in serialization and executed in deserialization. The clean-up closure function is written into the created Resilient Distributed Datasets (RDD) [Zaharia and Matei (2011)] as the initialization parameter. At this point, Spark bounds RDD data and the function together. Because RDD is the main line of Spark's data stream, nether the implementation of Stage scheduling nor Task scheduling can affect the RDD binding relationship with its closure. When the Executor assigned to the Worker node begins to actually work, it will parse and call the executional function bounded on the RDD to iterate over the data.

So from the micro perspective, the closure operation can be traced back. As a result, it is available to analyze the complexity of closures written in the RDD when it is created, and write the complexity into the RDD as the reference indicators of the practical scheduling.

C_p $(p = 1, 2, \ldots, M)$ is the complexity of each task, and M is the total number of tasks.

The scheduling policy will determine the priority of the scheduling based on complexity of the tasks.

3.2 Cluster stratification

With hardware updating and high-performance hardware introduced, cluster heterogeneity will tend to be obvious. Currently Spark scheduling is only based on the number of CPU cores, this assumption is not "fair" for actual heterogeneous clusters. Because computing power of single cores are significantly different and GPU and other high-computing-ability hardware perform much better than CPU. Actually, distributing tasks by the number of cores ignores the heterogeneity of each core itself, thus scheduling granularity is not enough satisfying.

Therefore, we propose a scheduling scheme that stratifies the cluster according to their core power, which is served as one of the targets of the final scheduling strategy. The synoptic idea is to implement several benchmark works on each node and record every duration, then divide nodes according to the specified number of layers.

P_i $(i = 1, 2, \dots, N)$ is index of node performance, N is the number of nodes. The index represents computing performance of CPU of corresponding node. The larger the index is, the more powerful the CPU is.

K is the total of layers, L_j is the level of the computing performance of node j. In the formula, α is a constant greater than 1, defined as hierarchical exponent, so that the final stratification result obeys an exponential function with the hierarchical index, which means, there are less nodes in higher level, and nodes in lower levels exist more. This result meets the expectations of stratification.

$$L_j = \begin{cases} L_{j-1} \times \alpha, & 1 \le j < K \\ 1, & j = 0 \end{cases} \tag{1}$$

Based on the two definitions above, we can get the stratifying algorithm of nodes in cluster.

Algorithm 1 Nodes stratifying
Input: workers in the cluster
Output: L
1: Traverse all Workers in Cluster, run the same set of benchmark tasks separately;
2: Count the executing time t of each Worker, and get array T=$1/t$;
3: Sort array T in ascending order;
4: normalize array T according to T [0];
5: Sign the first node as L_1;
6: for i=2, 3…T. size -1 do
7: get the corresponding layered result of each node according to Eq. (1) and save in array L;
8: **return** L;

According to this algorithm, a set of nodes at each level are available, then we can define P_i for the corresponding L_j, that is, nodes in the same level have the same performing index, the main purpose of the dividing policy is to eliminate the haphazard of scheduling. In addition, various benchmark tasks in Algorithm 1 result in multiple grading results, then we can create a grading matrix, whose rows represent nodes and columns represent tasks. For every node, the majority of grades from all benchmark tasks can be referred to as the final stratifying consequence.

3.3 Node detection

In the above, we discuss task scheduling of Spark based on the number of cores, which is actually a static way of analysis to the system. Cores themselves may be busy or free, furthermore, busy cores may keep different task queue. So it is necessary to monitor the usage and task queue of each CPU dynamically, in order to optimize the task scheduling method.

Spark cluster is a master-slave structure, Master and Worker communicate using RPC mechanism, so each Worker node can be added with a detecting module. To ensure the connection of Master and Worker, there will be information Interaction called *Heartbeat* between the two nodes, so that we can take advantage of the opportunity of each heartbeat to synchronize the current resource usage, Master preserves a resource utilization table of Workers in the local as the basis for the next resource allocation, to achieve the dynamic assignment of resources.

Specific parameters to be detected are queue length, CPU usage and node-performance index. Length of the queue on a single node will affect the completing time of the node. Generally, the longer the queue is, the longer the expected executing time is, tasks scheduled for it should be reduced. There are dependences when Spark execute tasks. If the queue is too long and tasks on other nodes depend on a task in the queue, the block of entire operation is possible, so it is significant to detect the length of the queue to avoid too-long task queue during scheduling. CPU usage directly reflects whether the core is busy or not, if CPU occupancy is too high, it indicates that the node lacks available computing resources, and redistributing tasks may need to wait for the release of computing resources, resulting in delays, so the scheduling should balance the CPU usage of each node. Node-performance index, as described in Definition 2 above, reflects the capacity of the node, scheduling should reasonably distribute task according to the capacity of each node. In addition, tasks with high complexity should be allocated to the nodes with high capacity, to reduce load of low-performing nodes and thus improve overall efficiency. Fig. 1 shows the scheduling structure of Spark.

After starting *Job*, *Driver* will create the corresponding TaskScheduler as the task scheduler of entire work. When tasks need for allocation, *Driver* requires resources from *Cluster Manager* of *Master* according to the task demand. At this time, *Cluster Manager* will get available *Executor*s' information on the *Worker* as the assignable resource pool for its task. When the task is assigned to an *Executor*, the Executor registers to the Driver and keeps connection through Heartbeat message. Therefore, it can synchronize the CPU status to TaskScheduler with a newly-created Heartbeat message, and TaskScheduler records and saves it as its scheduling metric.

The message interaction in detail is:

- When Executor is assigned to the Driver by the Master for Job, it sends registration information to Driver, so that Driver obtains the basic information of Executor;
- With instantiation and initialization starting, Executor starts a Heartbeater, as shown in Fig. 2, while Heartbeater will be bend with a HeartbeatTask, which means the task performed at each heartbeat;
- At each heartbeat, Executor creates a Heartbeat message that wraps data needed to be transferred to Driver. The message can be reconstructed to transfer customized information;
- The sent message will be received and analyzed by the Heartbeat Receiver of Driver. Obtained parameters are stored in Driver for subsequent use.

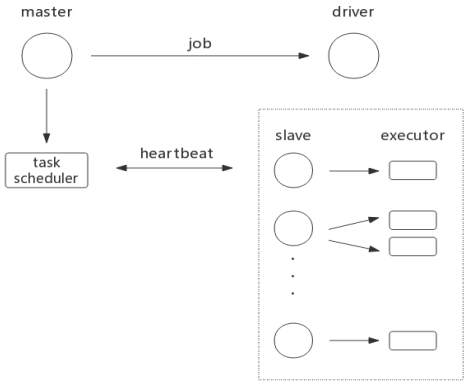

Figure 1: Spark cluster scheduling structure

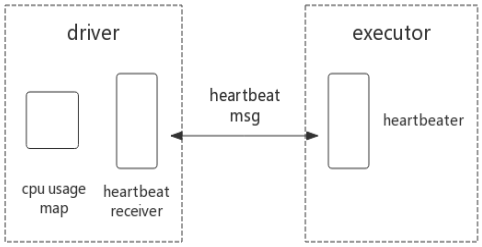

Figure 2: Spark cluster scheduling structure

3.4 Stratification scheduling

After separately analyzing Spark features from task, cluster and node gradations, this section will combine the three factors to achieve an integral scheduling strategy.

Firstly, before task is executed, nodes stratifying are pre-implemented to analyze the current cluster environment. As performance of all nodes, the analysis is written to the configuration of Spark system, which will be read as parameter and stored as SparkEnv during initialization. The pre-implementation can be triggered by the following event:

- Hardware changes;
- Stratification data is empty or changes;

- Duration is too long since last updating.

When creating RDD, Spark parses operating complexity of the incoming closure and records as C_p on the RDD. Through DAGScheduler [Mo, Yang and Cui (2014)] and TaskScheduler, Job will remain the following parameters before resource allocation:

- C_p for Task;
- $(P_i,\ runningTasks.len,\ CPUusage)$ for Executor to be distributed, representing separately node performance of the Executor, the length of its task queue, and CPU usage of the nodes recorded in TaskScheduler.

Related to the task to be scheduled, scores of each Executor are calculated by a module introduced in TaskScheduler.

- Evaluation function. For Executor q, define len_q as the length of its task queue, U_q as its CPU usage, $f(len_q, U_q)$ as an index of its current executing power.

  $$f(len_q, U_q) = \theta \times len_q + \mu \times U_q \tag{2}$$

  where θ, μ are corresponding coefficient. Smaller the function equals, more powerful the Executor is.

- Complexity conversion function. Define $g(C_p)$ as a function to converse complexity to the threshold of node-performance index.

  $$g(C_p) = \frac{C_p}{C_{max}} \times K \tag{3}$$

  where C_{max} is the peak of complexity. The result is the best node-performance index for the task.

Based on the equations and analysis above, we can get the scheduling algorithm of nodes in cluster.

Algorithm 2 Node scheduling

Input: Executor array G, length, U_{th}

Output: Selected executor S

1: foreach Executor q:

2: exclude all Executors whose queue length or CPU usage exceeds the threshold length and U_{th}.

3: calculate the evaluation function of each Executor according to Eq. 2.

4: sort the array G in descending order according to its level.

5: Get $g(C_p)$ according to Eq. 3.

6: for i=1, 2 …G.size do

7: if there is an Executor greater than $g(C_p)$ in queue G, select the one with the smallest evaluation function;

8: else select the executor S with the smallest evaluation function which level is closest to $g(C_p)$

9: Select randomly if step 3 and 4 both have no result;

10: **return** S

4 Evaluation

We conduct experiments from simulation in theory and real cluster condition to evaluate the algorithm. Indicators of experiments are mainly operation time and whether the distribution of nodes is balanced.

4.1 Simulation

- Node: parameters are performance index, CPU usage and task queue, created by Gaussian distribution;
- Task: parameters are complexity and CPU demand, created by Gaussian distribution. Task generation obeys Poisson distribution.
- Execution: tasks are allocated to nodes and added to their task queues, while CPU demand is appended to usage of nodes. Execution time is proportional to complexity of the task and inversely proportional to performance index of the node.
- Scheduling: original theory of Spark is considered as random algorithm.

With some fundamental conception above, this simulation experiment will statistic run several times. The mean of the running time is considered as the indicator of implementing efficiency. Keeping configuration of nodes constant, we execute ten different groups of tasks with the two algorithms. As the result, operating times of new algorithm are always shorter than the traditional one, as shown in Fig. 3. When we solely change the number of tasks, we get the result integrated to Fig. 4, which indicates that operating time grows by the task load increases, while operations with the new strategy are always efficient than the random method.

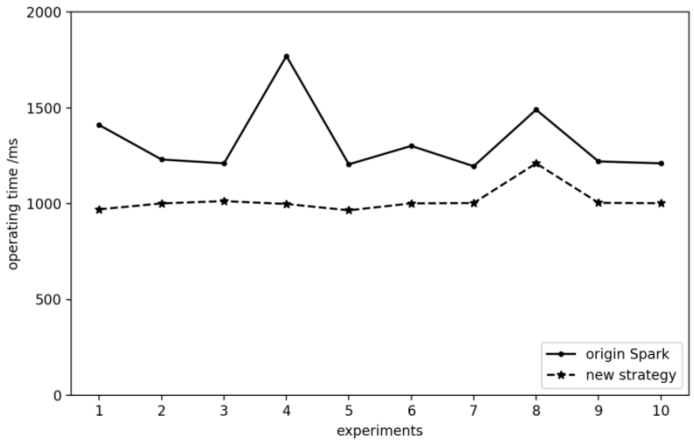

Figure 3: Time comparison of multiple experiments with the same task number

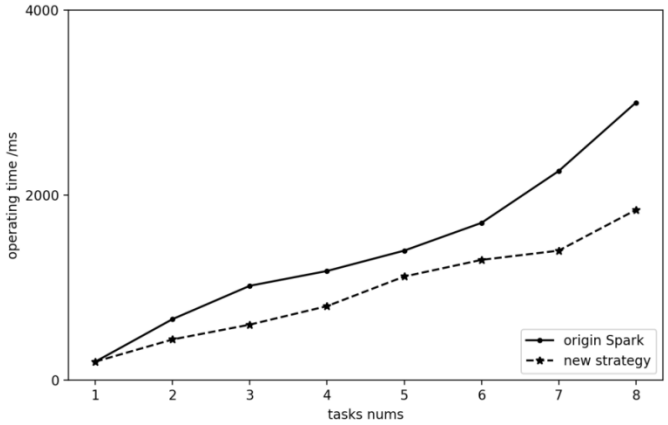

Figure 4: Time comparison of multiple experiments with different task number

4.2 Experiments

To evaluate the algorithm in hardware platform, we prepared four programs to run in Spark system with separately original method and our algorithm, such as SparkWordCount, SparkKmeans, SparkPageRank and SparkLR (Spark Logistic Regression).

Three machines from Ali Cloud Server Elastic Compute Service (ECS) are deployed to construct a simple heterogeneous cluster. One is both Master and Worker and the other two are Workers. The configurations are shown as following:

Table 1: Configuration of nodes

Hardware	Master Node	Worker Node
CPU	Two cores	Single core
memory	4 G DDR4	2 G DDR4
OS	Ubuntu 64 bit	Ubuntu 64 bit
HDD	40 G	40 G
bandwidth	1 Mbps	1 Mbps

We executed the four programs above on the micro-cluster to process data with multiple scales of samples. In terms of operating efficiency, operating time of tasks is an appropriate criterion. For SparkWordCount, SparkPageRank and SparkKmeans, we change the size of data set. For SparkLR, the number of sample points is 100,000 and dimension is configured to 10, and independent variable is the number of iterations. Dependent variables of all experiments are operating time. The results shown in Figs. 5-8 indicate that our algorithm has an improvement of operation time. In terms of nodes, usage of CPU and balance of nodes both need to be considered. For a single worker node, we monitored its CPU usage when executing the same task with two different strategies. As shown in Fig. 9, at most time CPU loads less with our algorithm than traditional Spark method. Operating the same task with the same size of data for several times, Fig. 10

compared average task distribution for the three nodes in the cluster. The indicator is occupancy rate of task queue on each node. We made presentation of percentage of every node with each strategy. As calculated, variance of the new strategy is 6.08, compared with 9.25 using original Spark. Then it is absolute that the distribution of our algorithm is more uniform than traditional method.

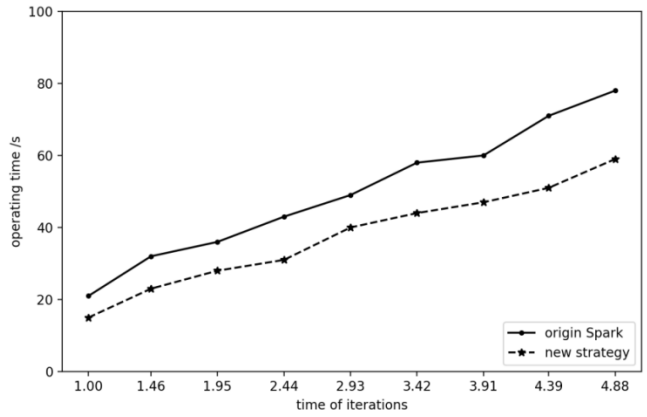

Figure 5: Word Count time consumptions of different job size

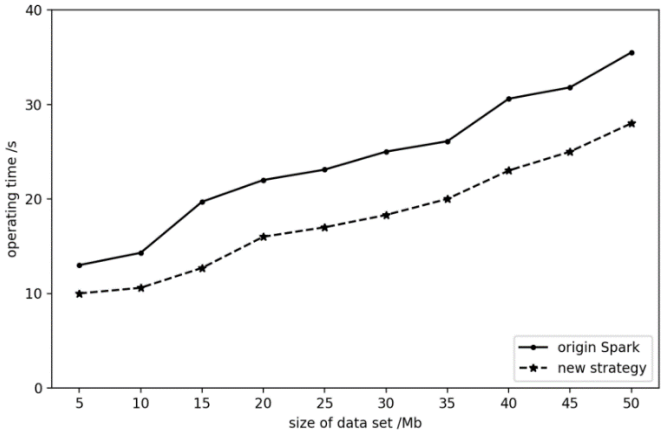

Figure 6: Spark LR time consumptions of different job size

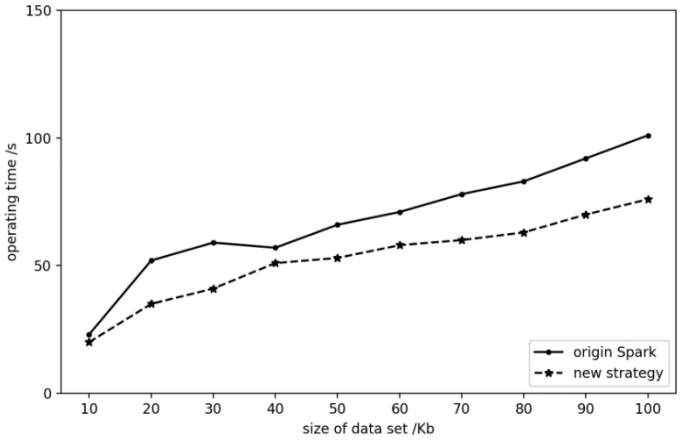

Figure 7: Spark Page Rank time consumptions of different job size

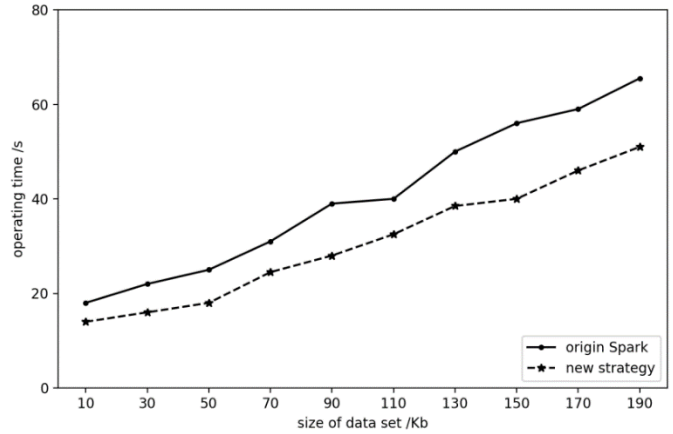

Figure 8: Spark K Means time consumptions of different job size

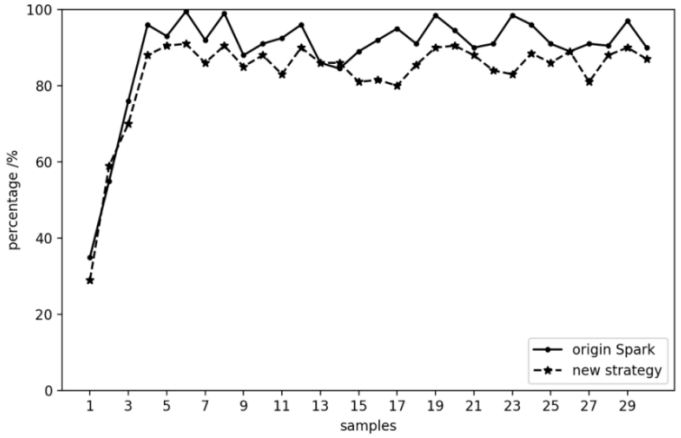

Figure 9: CPU usage monitor

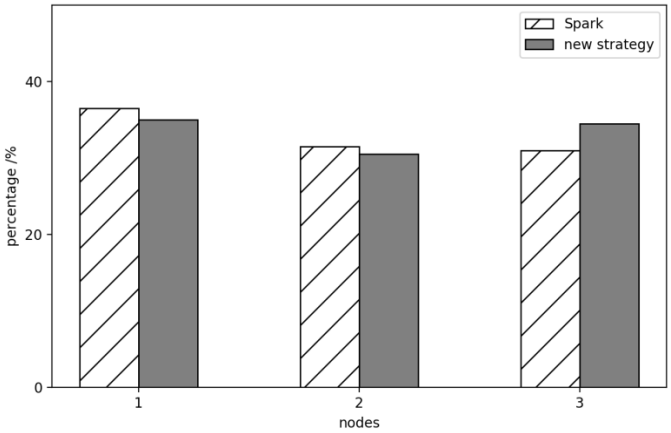

Figure 10: CPU distribution in different system

5 Conclusions

To develop Spark and distributed computing, it is significant to improve the performance of Spark system. We propose several new ideas in the field of task scheduling and integrate a scheduling strategy for heterogeneous cluster, compared with the random method of original Spark system. This strategy stratifies all nodes in cluster according to their computing performance and monitor their usage. Then task with complexity identified will be distributed to proper node to execute. Furthermore, we conducted several simulations and experiments proving that the new method is more uniform and efficient than traditional one of Spark system.

The new scheduling strategy takes advantage of capacity of computing nodes sufficiently, so that it effectively optimizes the operational efficiency of Spark in the actual cluster environment. Besides, it also enlightens a way to quickly increase the performance of cluster. Spark cluster can be recognized with high-performance components such as a small number of GPUs, and then the strategy will adjust structure of the cluster. Tasks with large-computing demand can be identified and assigned to these high-performance components in priority, so that the overall system improves its operational efficiency regardless of increasing complexity of tasks by making full use of the newly introduced high-performance components, which contains some engineering significance.

The following work will concern optimization of specific algorithms in this paper, such as auto-identification of tasks' complexity. Furthermore, optimization of parameters is also meaningful, some measures such as machine learning and probability methods are worth to be considered.

Acknowledgement: This work is supported by the National Natural Science Foundation of China (Grant No. 61472248, 61772337) and the SJTU-Shanghai Songheng Content Analysis Joint Lab.

References

Mo, K.; Yang, Y.; Cui, Y. (2014): A homochiral metal-organic framework as an effective asymmetric catalyst for cyanohydrin synthesis. *Journal of the American Chemical Society*, vol. 136, pp. 1746-1746.

Nightingale, E. B.; Chen, P. M.; Flinn, J. (2005): Speculative execution in a distributed file system. *ACM Special Interest Group on Operating Systems: Operating Systems Review*, vol. 39, no. 5, pp. 191-205.

Tang, Z.; Zhou, J.; Li, K.; Li, R. (2013): A mapreduce task scheduling algorithm for deadline constraints. *Cluster computing*, vol. 16, no. 4, pp. 651-662.

Xu, X.; Cao, L.; Wang, X. (2016): Adaptive task scheduling strategy based on dynamic workload adjustment for heterogeneous Hadoop clusters. *IEEE Systems Journal*, vol. 10, no. 2, pp. 471-482.

Yang, Z.; Zheng, Q.; Wang, S. (2016): Adaptive task scheduling strategy for heterogeneous Spark cluster. *Computer Engineering*, vol. 42, no. 1, pp. 31-35, 40.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; Stoica, I. (2010): Spark: Cluster computing with working sets. *Hot Cloud*, vol. 10, pp. 10.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J. et al. (2012b): Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX Association*.

Zaharia, M.; Konwinski, A.; Joseph, A. D.; Katz, R.; Stoica, I. (2008): Improving mapreduce performance in heterogeneous environments. *Operating Systems Design and Implementation*, vol. 8, no. 4, pp. 7.

Zaharia, M.; Borthakur, D.; Sen, S. J.; Elmeleegy, K.; Shenker, S. et al. (2010): Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. *Proceedings of the 5th European conference on Computer systems*.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J. et al. (2011a): Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Technical Report UCB/EECS-2011-82*, EECS Department, University of California, Berkeley.