Postprint: Network Big Data Classification Processing Method Based on Spark Framework Combined with Distributed KNN Classifier

Authors: Cao Yu, Wang Nan, Xu Zhichao

Date: 2018-08-13T00:00:00+00:00

Abstract

To address the inability of existing big data classification methods to satisfy the time and storage space constraints in big data applications, we propose a design method for a parallel multi-label K-nearest neighbor classifier for big data based on the Apache Spark framework. To reduce the cost overhead of existing MapReduce schemes through the use of in-memory operations, first, the training set is divided into multiple partitions leveraging the parallel mechanism of the Apache Spark framework; then, in the Map phase, the K nearest neighbors of the sample to be predicted are found within each partition, and subsequently, in the reduce phase, the final K nearest neighbors are determined based on the results from the Map phase; finally, the label sets of the neighbors are aggregated in parallel, and the target label set of the sample to be predicted is output by maximizing the posterior probability. Experiments conducted on four big data classification datasets, including PokerHand, demonstrate that the proposed method achieves lower Hamming loss, thereby validating its effectiveness.

Full Text

Preamble

Network Big Data Classification Processing Method Based on Spark Framework and Distributed KNN Classifier

Cao Yu[1], Wang Nan[2,3], Xu Zhichao[2] [1](Dept. of Computer, Harbin Finance University, Harbin 150030, China) [2](Institute of Management & Credit, Jilin University of Finance & Economics, Changchun 130117, China) [3](School of Computing, Jilin University, Changchun 130012, China)

Abstract: Existing big data classification methods struggle to meet the time and storage space constraints inherent in big data applications. To address this

limitation, we propose a design method for a big data parallel multi-label K-nearest neighbor classifier based on the Apache Spark framework. To reduce the cost overhead of existing MapReduce schemes by leveraging additional in-memory operations, we first partition the training set into multiple partitions using Spark's parallel mechanism. Then, in the Map phase, we find the K nearest neighbors for each partition of the sample to be predicted, and subsequently determine the final K nearest neighbors in the Reduce phase based on the Map phase results. Finally, we aggregate the label sets of the neighbors in parallel and output the target label set for the sample to be predicted by maximizing the posterior probability. Experiments conducted on four big data classification datasets including PokerHand demonstrate that our proposed method achieves lower Hamming loss, thereby proving its effectiveness.

Keywords: classification processing; Apache Spark; parallel mechanism; data mining; Hamming loss; K-nearest neighbor

0 Introduction

In recent years, big data has become a focal point of research, and large-scale information processing has emerged as a necessary and critical task [1~3]. Continuous technological innovation and changing consumption patterns have driven the informatization of commerce, which in turn has led to a dramatic increase in available data volume. Meanwhile, social media, biomedicine, and entertainment activities also generate massive amounts of data daily. Without proper knowledge extraction processes, such data remains useless. Consequently, there is an urgent need for convenient tools capable of leveraging these data resources.

The K-nearest neighbor algorithm (kNN) [4] is an intuitive and effective non-parametric model commonly used for classification and regression tasks. Some studies have introduced kNN classifiers into the MapReduce process, but their objective is not to perform precise KNN classification; instead, they employ partial KNN (applying KNN to subsets of training data) for clustering large datasets. For instance, literature [5] proposes a novel approach combining KNN with principal component analysis for large-scale dataset clustering. Literature [6] presents a method with two distinct stages: first using K-means to partition the entire dataset, then computing KNN within each group to provide approximate results. Other research focuses not on classification or regression tasks, but on proposing distributed KNN computations for queries in MapReduce. For example, literature [7] applies KNN-join (exact or approximate) queries in a two-stage MapReduce process; literature [8] proposes an iterative Hadoop MapReduce process for KNN-based image classification. However, this approach iteratively executes MapReduce for each individual test instance, resulting in significant time consumption. Literature [9] proposes a Hadoop MapReduce process that can classify large numbers of test samples simultaneously, reducing Hadoop' s startup costs, but this method suffers from low classification accuracy

and requires further improvement.

In this paper, we propose a design method for a big data parallel multi-label K-nearest neighbor classifier based on the Apache Spark framework. In our implementation, we aim to leverage Spark' s flexibility to reduce the cost overhead of existing MapReduce schemes by utilizing additional in-memory operations. To manage massive test sets, we minimize the number of iterations based on memory constraints by using the maximum possible number of test examples per iteration. In each iteration, we apply a KNN MapReduce process that finds the K nearest neighbors for each partition of the sample to be predicted in the map phase, and determines the final K nearest neighbors in the reduce phase. Finally, we aggregate the label sets of the neighbors in parallel.

1.2 Spark Framework

The Spark framework can be regarded as the next-generation distributed computing framework and an extension of MapReduce [13]. The key difference lies in Spark' s introduction of a new abstraction for data under processing called Resilient Distributed Dataset (RDD), which enables practitioners to define additional data operations without strictly adhering to Map and Reduce functions. However, the MapReduce paradigm remains central to the Spark platform, as most computations follow the same pattern of applying several functions individually to each data unit and then combining them through network shuffling.

The primary motivation for Spark becoming the de facto standard for distributed data processing relies on architectural differences: while Hadoop MR relies on hard disk drives to persist intermediate data between operations, Spark focuses on faster main memory to maintain its data structures. This allows for defining more complex execution paths and increased support for iterative processes. Figure 2 [Figure 2: see original paper] illustrates Spark operators and data space.

1.1 MapReduce Framework

The MapReduce (MR) framework provides a highly scalable and flexible framework for parallel data computation, primarily designed to address the growing volume of digital data [10~12]. In recent years, data generation and storage capabilities have grown to the petabyte scale, driven by more mature hardware and novel processing technologies. MR's main potential lies in its computational abstraction, where entire processing is divided into smaller task types—Map and Reduce—that are distributed and processed uniformly across the cluster. Practitioners only need to provide these two functions, avoiding the need to adapt processing to the underlying cluster architecture or data characteristics. This framework offers a highly scalable and fault-tolerant environment for parallel data processing.

Data managed in MR workflows is represented as key-value pairs in the form

key, value . Each Map task' s initial input typically serves as an input partition of raw data associated with arbitrary keys. Each Map task processes its chunk of data by applying a defined function, producing numerous new intermediate key-value output pairs. The system collects, combines, and sorts these intermediate output pairs by their corresponding keys, and sends matching pairs that share the same key as input to the Reduce phase, which consists of a common key and a list containing associated values. The Reduce step then executes, producing final output typically composed of newly formatted key-value pairs. Figure 1 [Figure 1: see original paper] shows the detailed flow of MR programs.

The popularity of this framework was driven by Apache Hadoop' s implementation; the open-source community and many private companies have jointly worked to provide a widely available software stack for big data preprocessing. The Hadoop ecosystem is extensive, but mainly consists of two core components: the general-purpose MapReduce and the Hadoop Distributed File System (HDFS). They enable deployment of inexpensive large computer clusters as powerful execution engines as well as reliable and fault-tolerant distributed data storage.

Recent cloud computing technologies have provided an ideal environment for handling such problems. The MapReduce framework and its open-source implementation in Hadoop are tools for processing data-intensive applications based on the principle of data locality, and have been widely disseminated in data mining. However, researchers have identified several limitations of Hadoop MapReduce in designing scalable machine learning tools. For applications that share data across multiple steps (including iterative algorithms or interactive queries), MapReduce is inefficient. Recently, multiple platforms for large-scale data processing have emerged to overcome Hadoop MapReduce' s issues, among which Spark has become one of the most flexible and powerful engines for faster distributed computing in big data. This platform allows user programs to load data into memory and query it repeatedly, making it more suitable for online, iterative, or data streaming algorithms.

2 KNN Classifier

The kNN algorithm is a non-parametric method that can be used for both classification and regression tasks [15]. Let D_s be the training dataset and T_s be the test set, consisting of n and m samples respectively. Each sample x is a multivariate tuple $(x_1, x_2, ..., x_p)$, where p is the number of features. Here, x_{ij} represents the value of the j-th feature of the i-th sample. Samples belong to category ω, determined by x and a p-dimensional space. For set D_s, the category ω is known, while this is unknown for set T_s.

For each test sample x_{test} contained in set T_s, the kNN algorithm searches for the k closest samples in set D_s. Therefore, KNN calculates distances between x_{test} and all samples in D_s. Euclidean distance is the most widely used metric for this purpose. Based on the calculated distances, training samples are sorted

in ascending order, and the k nearest samples are selected. They are then used to calculate the most dominant class label. The choice of k value may affect the technique' s performance and noise tolerance. Although KNN performs excellently in a wide variety of problems, it lacks scalability for large training datasets.

The main issues in processing large-scale data are: a) **Runtime complexity**. Finding nearest neighbor training samples for a single test instance is computationally expensive because it requires sorting computed distances. When finding k neighbors, this becomes even more computationally complex, requiring additional complexity $O(n \log n)$. This process needs to be repeated for each test example. b) **Memory consumption**. For rapid distance computation, the KNN model requires storing training data in memory. When both D_s and T_s are too large, they can easily exceed available RAM storage.

These drawbacks motivate distributing KNN processing across node clusters. Literature contains various methods for executing KNN on MapReduce. While KNN classifiers aim to provide predicted classes, KNN-join itself outputs neighbors for individual tests. Therefore, these methods cannot be used for classification. Figure 3 [Figure 3: see original paper] illustrates the distributed KNN process.

For classification tasks (also applicable to regression), existing methods are simpler than KNN-join approaches because they don' t need to provide neighbors themselves, only their classes. So far, two main methods have been proposed, both focusing on using the Map phase to split training data into p disjoint parts. For example, literature [14] proposes iteratively repeating the MapReduce process for each individual test example (without a clearly defined reduce function), which is extremely time-consuming in both Hadoop and Spark. Literature [15] proposes a classifier called MR-KNN that uses one MapReduce process to manage test set classification. Hadoop reads test data row by row in the map phase. Therefore, this pattern is scalable, but its performance can be further improved through in-memory solutions.

3 Proposed Method

This paper presents an alternative distributed KNN model for big data classification using Spark, referred to as a design method for big data parallel multi-label K-nearest neighbor classifier based on the Spark framework. When both training and test sets are large datasets, this paper focuses on reducing the runtime of the KNN classifier. When computing KNN within the parallel framework, many other factors may affect execution time, such as the number of MapReduce jobs j or the required number of Map and Reduce tasks (m and r). Therefore, writing efficient and accurate KNN in Spark is challenging, and multiple key points must be considered to achieve an efficient and scalable model.

This section introduces the MapReduce process that manages classification of test data subsets fitting into memory. Therefore, this MapReduce process is

based on MR-KNN, but differs in that it allows multiple Reducers, checks runtime requirements to avoid memory swapping, and is implemented under Spark. As a MapReduce model, this divides computation into two main phases: Map and Reduce. The Map phase splits training data and computes distances and corresponding categories of the k nearest neighbors for each test sample within each block. The Reduce phase aggregates the k nearest neighbor distances from each map and produces a definitive list of k nearest neighbors. Finally, the majority voting procedure in the kNN algorithm is executed to predict the resulting category.

3.1 Map Phase

First, we assume that the training set D_s and corresponding test sample subset T_s have been previously read from HDFS as RDD objects. Therefore, when the training dataset D_s is read, it is already partitioned into p user-defined disjoint subsets. Each map processes subset D_{si} using samples from each partition of the training set, where $i = 1, 2, ..., p$. Thus, each map processes approximately the same number of training instances.

To achieve an exact KNN implementation, the input test set T_s is not split together with the training set, meaning both D_s and T_s should fit completely in memory. In the Spark implementation described in this paper, a KNN searcher is initialized with k in each partition. Meanwhile, each partition obtains local K nearest neighbors for the entire test set samples.

Each map sends multiple outputs and allows using multiple reducers. The result data from each partition exists as key-value pairs $(key, value)$. The key represents the number of each test sample, while the value represents the neighbor array with elements as key-value pairs $(id, distance)$. When the used training and test datasets are very large, using more Reducers may be beneficial. Algorithm 1 describes the main process of the Map phase, where Ds_{file} represents the training set file, Ts_{file} represents the test set file, and $List[id, distance]$ represents the sample neighbor set.

Algorithm 1 Distributed KNN Map Phase Process

```
Ds = textFile(Ds_file).map(normalize).map(mapLinearNNSearch).cache
Ts = textFile(Ts_file).map(normalize).map(keyvalue).collect
arr_sc = broadcast(Ts)
ans = computeKNN(arr, Ds)
```

3.2 Reduce Phase

The Reduce phase includes aggregating the k nearest neighbors from the p partitions provided by the map phase. After the map phase completes, for each test sample x_{test} there are $p \times k$ neighbors, and all elements with the same key are grouped. This function processes each element in this list one by one and updates the result list with k neighbors.

Since vectors from the map are sorted by distance, the update process becomes faster. This involves merging two sorted lists to obtain the k values; therefore, the worst-case complexity is $O(k)$. The function compares each distance value of each neighbor one by one, starting from the nearest neighbor. If the distance is less than the current value, the corresponding category and distance at that position are updated; otherwise, it continues to the next value. Algorithm 2 provides details of the Reduce operation.

Algorithm 2 Distributed Reduce Phase Process

```
res = new Array[type](id, k+1)
for i = 0 to p do
    for j = 0 to k do
        if ans[i][j].distance < res[j].distance then
            res[j] = ans[i][j]
            id = j + 1
```

In summary, for each instance in the test set, the Reduce function aggregates these values according to the previously described function. To simplify this process, we use Spark' s ReduceByKey(func) transformation.

4.1 Experimental Configuration

In this experimental study, we use four big data classification datasets: PokerHand [16], NUS-WIDE-bow [17], Susy [18], and Higgs [19]. We randomly sample these datasets to obtain balanced categories. These datasets contain not only large numbers of instances but also relatively numerous features, allowing us to observe how this characteristic affects the proposed model.

Table 1 summarizes the properties of these datasets, including sample count, feature count, and label count information.

Table 1 Dataset Description | Dataset | Samples | Features | Labels | |——|——|——-|—-| | Higgs | 5,000,000 | 28 | 2 | | PokerHand | 1,025,010 | 11 | 10 | | NUS-WIDE-bow | 269,648 | 500 | 81 | | Susy | 5,000,000 | 18 | 2 |

For the experimental study, all datasets are partitioned using a 5-fold cross-validation (5-fcv) scheme. This means the dataset is divided into 5 subsets, with 80% of each subset used as training samples and the remainder as test instances. For each subset, the kNN algorithm computes the nearest neighbors. In the presented MapReduce scheme, the number of dataset instances is directly related to the number of Maps used; therefore, more Maps result in fewer instances per Map.

All test experiments are executed on a cluster consisting of 16 nodes: one master node and 16 compute nodes. The compute nodes use 2x Intel Xeon CPU E5-2620 processors. Specific details of the software and configuration are as follows:

a) MapReduce implementations: Hadoop 2.6.0-cdh5.4.2 and Spark 1.5.1

b) Maximum number of map tasks: 256

c) Maximum number of reduce tasks: 128

d) Maximum memory per task: 2GB

e) Operating System: Cent OS 6.5

Note that the total number of available cores is 192, which becomes 384 with hyper-threading enabled. Therefore, when the algorithm explores more than 384 maps, linear speedup cannot be expected because tasks will be queued. For these cases, we focus our analysis on Map and Reduce operations.

4.2 Performance Metrics

We evaluate the performance and scalability of the proposed method using the following three metrics:

a) **Hamming Loss**. This represents the average number of misclassified labels. Better algorithm performance corresponds to smaller Hamming loss.

$$hloss = \frac{1}{p}\sum_{i=1}^{p}\frac{|Y_i \Delta h(x_i)|}{|q|}$$

where Δ is the symmetric difference between two sets, $|q|$ is the length of the label vector, and h represents the multi-label classifier.

b) **Runtime**. This records the total time spent by the kNN classifier to classify a given test set according to the training dataset. The total runtime of the parallel method includes reading and distributing all data, computing the k nearest neighbors, and performing majority voting.

c) **Speedup**. To demonstrate the efficiency of the parallel algorithm compared to its sequential execution version, we measure the runtime of both sequential and parallel versions. According to Amdahl' s law [20], in a fully parallel environment, the theoretical maximum speedup equals the number of cores used.

$$Speedup = \frac{time_{baseline}}{time_{parallel}}$$

where $time_{baseline}$ is the total runtime of the sequential version, and $time_{parallel}$ is the total runtime after using the parallel version.

microF (label-based micro F-score) represents label-based precision:

$$microF_\beta = \frac{\sum_{j=1}^{|q|}(1+\beta^2)TP_j}{\sum_{j=1}^{|q|}((1+\beta^2)TP_j + \beta^2 FN_j + FP_j)}$$

where TP_j, FP_j, TN_j, FN_j are true positives, false positives, true negatives, and false negatives in the confusion matrix for label j.

4.3 Comparison with Sequential KNN Methods

This section compares our method with the sequential version of KNN based on the Hadoop MapReduce framework. For this purpose, we use the NUS-WIDE-bow and Higgs datasets, which are commonly used in computer vision. For these datasets, our method requires only one iteration because the test dataset fits into each map' s memory, and the number of reducers is also fixed at 1.

First, we run the sequential version of kNN on these datasets as a baseline. This sequential version reads the test set row by row as a direct solution to avoid memory issues. Table 2 shows the runtime (in seconds) and average precision results obtained by the standard sequential KNN algorithm with different numbers of neighbors (where the number of Maps is set to 32).

Table 2 Performance of Sequential kNN | Dataset | k=1 | k=3 | k=5 | |——|–|–|–| | NUS-WIDE-bow | 1256s / 0.72 | 1876s / 0.75 | 2432s / 0.76 | | Higgs | 3421s / 0.68 | 4567s / 0.71 | 5890s / 0.73 |

Table 3 summarizes the results obtained by both methods when $k = 1$ (with neighbor count set to 1). It shows the average total time and speedup achieved relative to the sequential version for different numbers of Maps. As mentioned previously, both methods correspond to exact implementations of KNN.

Table 3 Results from Sequential KNN and Our Method | Maps | Sequential KNN Avg Runtime (s) | Our Method Avg Runtime (s) | Speedup | |—|————-|————-|——| | 32 | 1256 | 187 | 6.72 | | 64 | 1256 | 165 | 7.61 | | 128 | 1256 | 152 | 8.27 |

Based on the experimental results, we make the following analysis:

As can be seen in Table 2, the sequential KNN method requires considerably high runtime for both datasets. However, Table 3 shows that as the number of Maps increases, the runtime of both methods decreases. Compared to the sequential version KNN, our parallel KNN achieves faster linear speedup. This is due to the use of in-memory data structures, which allows us to avoid reading test data row by row from HDFS.

Comparing the sequential version KNN with our parallel KNN, the results show that our Spark-based parallel KNN reduces runtime by 6-8 times compared to the Hadoop framework-based sequential version KNN.

4.4 Performance Comparison with Other Methods

Among existing MapReduce-based distributed kNN models, we compare our method with the MR-KNN method, which is based on the Hadoop MapReduce framework. The MR-KNN method was originally designed for purposes other than classification and requires increasing data sizes and even a quadratic number of Map tasks. Consequently, this scheme has theoretically much higher complexity than our proposed technique.

Table 4 presents the performance evaluation results of our method, while Table 5 shows the performance evaluation results of the MR-KNN method, with Map count and neighbor count set to 128 and 1 respectively. From these tables, we can see that our method achieves lower Hamming loss, less runtime, and also outperforms the comparison method in both speedup and microF metrics, demonstrating the effectiveness of our approach.

Table 4 Performance Evaluation Results of Our Method | Dataset | Hamming Loss | Runtime (s) | Speedup | microF | |——|———-|———-|——|——| | Higgs | 0.12 | 152 | 8.27 | 0.89 | | PokerHand | 0.08 | 98 | 7.45 | 0.91 | | NUS-WIDE-bow | 0.15 | 187 | 6.72 | 0.76 |

Table 5 Performance Evaluation Results of MR-KNN Method | Dataset | Hamming Loss | Runtime (s) | Speedup | microF | |——|———-|———-|——|——| | Higgs | 0.18 | 245 | 5.12 | 0.82 | | PokerHand | 0.14 | 156 | 4.67 | 0.85 | | NUS-WIDE-bow | 0.22 | 298 | 4.21 | 0.68 |

5 Conclusion

The K-nearest neighbor classifier is a simple yet effective well-known method in data mining. Due to time and memory constraints, the practical application of this model in the big data domain is not feasible. In this work, we provide a design method for a big data parallel multi-label K-nearest neighbor classifier based on the Apache Spark framework. The Map phase computes the k nearest neighbors in different training data partitions, and the Reduce phase determines the final K nearest neighbors based on the Map phase results. Finally, the label sets of the neighbors are aggregated in parallel, and the target label set for the sample to be predicted is output by maximizing the posterior probability. Test results on four big data classification datasets validate the effectiveness of the proposed approach.

References

[1] Dong Yangyi, Li Weihua, Yu Hui. Hierarchical relation mining of Chinese text based on mixed cosine similarity [J]. Application Research of Computers, 2017, 34 (5): 1406-1409.

[2] Dang Hongen, Zhao Erping, Liu Wei, et al. Closed frequent itemset mining method using data transformation and parallel operations [J]. Natural Science

Journal of Xiangtan University, 2018, 40 (1): 119~122.

[3] Seydell-Greenwald A, Raven E P, Leaver A M, et al. Diffusion imaging of auditory and auditory-limbic connectivity in tinnitus: preliminary evidence and methodological challenges [J]. Neural Plasticity, 2014, 2014 (2): 1-10.

[4] Yuan Jidong, Wang Zhihai, Sun Yangei, et al. K-Nearest Neighbor Classifier for Complex Time Series [J]. Journal of Software, 2017, 28 (11): 3002-3017.

[5] Feng Huan, Eyers D, Mills S, et al. PCAF: scalable, high precision KNN search using principal component analysis based filtering [C]// Proc of International Conference on Parallel Processing. 2016: 638-647.

[6] Manjusha M, Harikumar R. Performance analysis of KNN classifier and K-means clustering for robust classification of epilepsy from EEG signals [C]// Proc of International Conference on Wireless Communications, Signal Processing and Networking. 2016: 2412-2416.

[7] Tiwari S, Kaushik S. Boundary points detection using adjacent grid block selection (AGBS) KNN-join method [C]// Proc of International Conference on Machine Learning and Data Mining. 2012.

[8] Demott P J, Prenni A J, Mcmeeking G R, et al. Integrating laboratory and field data to quantify the immersion freezing ice nucleation activity of mineral dust particles [J]. Atmospheric Chemistry & Physics, 2015, 14 (11): 112-128.

[9] Kumar A, Kiran M, Prathap B R. Verification and validation of MapReduce program model for parallel K-means algorithm on Hadoop cluster [C]// Proc of the 4th International Conference on Computing, Communications and Networking Technologies. IEEE, 2014: 1-8.

[10] Wang Shuyan, Yang Xin, Li Keqiu. Skyline Computing on MapReduce with Hyperplane-Projections-Based Partition [J]. Journal of Computer Research and Development, 2014, 51 (12): 2702-2710.

[11] Chen Zhen, Xia Jingbo, Yang Juan, et al. MapReduce-based Situation Assessment Algorithm for Support Vector Machines [J]. Journal of Computer Applications, 2016, 36 (1): 133-137.

[12] Backman N, Pattabiraman K, Fonseca R, et al. C-MR: continuously executing MapReduce workflows on multi-core processors [C]// Proc of International Workshop on Mapreduce and Its Applications. 2012: 1-8.

[13] 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.

[14] Rogala M, Hidders J, Sroka J. DatalogRA: datalog with recursive aggregation in the spark RDD model [C]// Proc of International Workshop on Graph Data Management Experiences and Systems. New York: ACM Press, 2016: 3.

[15] Lu Xuanmin, Yuan Wenle, Qiu Yang, et al. An Improved dynamic prediction fingerprint location algorithm based on KNN [J]. Application Research of

Computers, 2017, 34 (7): 2016-2018.

[16] Jabin S. Poker hand classification [C]// Proc of International Conference on Computing, Communication and Automation. 2017: 269-273.

[17] Gu Yun, Xue Haoyang, Yang Jie, et al. Automatic Image Annotation Exploiting Textual and Visual Saliency [C]// Proc of International Conference on Neural Information Processing. Springer International Publishing, 2016: 95-102.

[18] Polpaya I C, Rao C L, Varughese S. Electromechanical behavior and microstructure of highly sensitive polyaniline//ethylene vinyl acetate composite Piezo-Resistive materials [C]// Proc of Conference on Smart Materials, Adaptive Structures and Intelligent Systems. 2016.

[19] Aad G, Al E, Bentvelsen S, et al. Observation of a new particle in the search for the standard model Higgs boson with the ATLAS detector at the LHC [J]. Physics Letters B, 2012, 716 (1): 1-29.

[20] Das A K, Jaeki H, Goswami S, et al. Augmenting Amdahl' s Second Law: A Theoretical Model to Build Cost-Effective Balanced HPC Infrastructure for Data-Driven Science [C]// Proc of IEEE International Conference on Cloud Computing. 2017: 147-154.

Note: Figure translations are in progress. See original paper for figures.