Hindawi
Mobile Information Systems
Volume 2018, Article ID 3890341, 14 pages
https://doi.org/10.1155/2018/3890341

Research Article

An Evaluation Model and Benchmark for Parallel Computing Frameworks

Weibei Fan,[1,2] Zhijie Han,[2,3] and Ruchuan Wang[2]

[1]*School of Computer Science and Technology, Soochow University, Suzhou, Jiangsu 215006, China*
[2]*Jiangsu High Technology Research Key Laboratory for Wireless Sensor Networks, Nanjing, Jiangsu 210003, China*
[3]*College of Computer & Information Engineering, Henan University, Kaifeng, Henan 475001, China*

Correspondence should be addressed to Zhijie Han; hanzhijie@126.com

Received 15 December 2017; Revised 8 February 2018; Accepted 10 February 2018; Published 29 March 2018

Academic Editor: Laurence T. Yang

MARS and Spark are two popular parallel computing frameworks and widely used for large-scale data analysis. In this paper, we first propose a performance evaluation model based on support vector machine (SVM), which is used to analyze the performance of parallel computing frameworks. Furthermore, we give representative results of a set of analysis with the proposed analytical performance model and then perform a comparative evaluation of MARS and Spark by using representative workloads and considering factors, such as performance and scalability. The experiments show that our evaluation model has higher accuracy than multifactor line regression (MLR) in predicting execution time, and it also provides a resource consumption requirement. Finally, we study benchmark experiments between MARS and Spark. MARS has better performance than Spark in both throughput and speedup in the executions of logistic regression and Bayesian classification because MARS has a large number of GPU threads that can handle higher parallelism. It also shows that Spark has lower latency than MARS in the execution of the four benchmarks.

1. Introduction

Cloud computing has increased exponentially because of the increasing demands in storing, processing, and retrieving a large amount of data in a cloud cluster. Apache Hadoop [1] is proposed as a framework that allows for the distributed processing of large data sets, which use a simple programming model through a cluster. It is crucial for processor architects to understand what processor microarchitecture parameters affect performance [2]. MapReduce is one of the main components of Hadoop, which is parallelized scalable for computing frameworks. Several applications based on Hadoop are widely used in machine learning, data mining, and graph processing due to its simple interface [3]. Encouraged by the success of the CPU-based MapReduce, a MapReduce framework on graphics processors is proposed in [4]. Spark [5] is another cluster-computing framework that supports the MapReduce paradigm yet does not depend on it. Among them, Spark gains the most popularity because it outperforms Hadoop significantly for interactive and iterative applications. A clear understanding of system performance under different circumstances is key to make decision in resource management and task planning, such as providing decisions for the hardware configuration of nodes in the cluster and the adjustment of system parameters.

Since time critical and high performance is necessary for dealing with these tasks, the throughput of server CPUs and I/O is a serious challenge when dealing with massive data [6]. The performance cannot satisfy the requirements of data processing for the traditional technical architecture, such as data processing, data storage, fault tolerance, and data acquisition. MapReduce is a calculation model based on CPU, which involves two procedures: Map and Reduce. Since CPU supports only a few outstanding memory accesses, fetching massive data from memory can lead to significant latency. Consequently, the high parallelism of query processing is difficult to explore in alleviating the memory access latency. As a result, the CPU cache could not help much in reducing

the memory access latency due to its small capacity. GPU has been recently utilized in various domains, including high-performance computing. GPU can be regarded as massively parallel processors with 10x faster computation and 10x higher memory bandwidth than CPU. GPU has a strong ability in parallel computing and is composed of thousands of computer units [7]. GPU is especially suitable for data-intensive parallel computing mainly because it utilizes a number of threads to process different data at the same time. MARS is a GPU-based MapReduce, which is designed for batch tasks, but it is also widely used for iterative tasks. Spark is a parallel computing engine which is designed mainly for iterative tasks, but it is also used for batch tasks. Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing. Spark provides a variety of data set operations called resilient distributed data sets (RDDs) [5], which implement in-memory data structures by using cache intermediate data across a set of nodes. Since RDDs are kept in memory, they are efficient for algorithms that need multiple iterations. However, the input data type and size can affect the performance of Spark and MARS on a particular job significantly. In design and implementation of algorithms, it will be extremely difficult in predicting the performance metric of a job with limited computing capability, such as execution time, throughput, resource consumption, and latency [8].

Given the increasing use of parallel computing frameworks, the design of methods that allow one to understand and predict the performance of such applications is appealing [9]. Evaluation performance modeling is crucial in both researching and engineering work to gain insights into which part of the complex systems [10]. Evaluation performance models are attractive tools that serve the purpose as they might provide reasonably accurate task performance estimates such as throughput and execution time and ultimately help one to answer the aforementioned questions, at significantly lower cost than simulation and experimental evaluation of real setups [11]. They are usually used as a means to state a parallel computing system which is influenced by a wide range of equations.

Multifactor line regression (MLR) [12] is an important method in statistical analysis and data mining and is widely used in the field of engineering technology. The traditional MLR prediction model is only suitable for small-scale input sample data and can only run on a single node. When the input sample data size increases, it tends to slow down due to the increase of the amount of computation or cannot even get the conclusion within the valid time. This is because matrix multiplication is a basic operation in MLR prediction. When the input data are large, high-order matrices are formed, and the product operations of high-order matrices have higher time complexity. Since the calculation process needs more system resources, the computational efficiency is reduced. Expanding the computational complexity of matrix multiplication and reducing its computation time will meet the requirements to deal with large-scale data of the MLR prediction model.

Support vector machine (SVM) [13] is a kind of binary class classification model, and its basic model is defined as the largest linear classifier on the feature space. In particular, analytical techniques based on SVM have been used to analyze and predict the performance of various distributed and parallel systems. Since SVM does not involve probability measurement and laws of large numbers, it only requires small sample data. SVM can effectively achieve from the training sample to predict the test data inference process. The learning strategy is the interval maximization, which can be transformed into a convex quadratic programming problem. The final decision of the SVM is determined by some support vectors, not by the dimensions of the sample data. The process to classify data sets by SVM involves mapping the input space to the high-dimensional feature space through preselected nonlinear mappings. SVM can identify key sample data and eliminate redundant data, which means it is insensitive to the increase and decrease of unsupported vectors. The process of the SVM is easy to implement and robust. To summarize the description, SVM and MLR are two major performance evaluation methods. We apply both to the parallel computing framework to compare which one has higher accuracy in terms of execution time and resource consumption requirement.

Our work aims to build an analytical evaluation model for parallel computing frameworks and to deploy MARS and Spark on the cloud computing environment. By comparing two evaluation models, we conclude that SVM has higher accuracy in predicting than MLR. The purpose of studying benchmarks is to help big data's clients select the appropriate framework for processing data. Our major contributions are as follows:

(1) We propose an evaluation performance model based on the machine learning method SVM for parallel computing frameworks and have given a comparison with MLR.

(2) We study four benchmarks on MARS and Spark, respectively, which conduct a detailed analysis to understand how MARS and Spark process batch and iterative jobs.

The rest of the paper is organized as follows. Section 2 introduces the background and related works. Section 3 presents the analytical performance modeling techniques. In Section 4, a performance evaluation model is proposed based on SVM and gave a performance comparison with MLR. Section 5 illustrates the implementation and analysis experimental results of four benchmarks for MARS and Spark. The whole paper is concluded in Section 6.

2. Related Works

Big data has wide applications, such as batch processing, stream processing, interactive analysis, and query processing. There have been many proposals for performance analysis techniques specific to parallel computing frameworks both in evaluation performance models and benchmarks. There are several implementations of the MapReduce model, specifically in terms of workload, task scheduling, and heterogeneous environments [14–16]. Zhang et al. [17] proposed a distributed HOPCM method based on

MapReduce for very large amounts of heterogeneous data. Vianna et al. [18] presented an evaluation model that estimates performance for a Hadoop online prototype using the job pipeline parallelism method. Zhang et al. [19] reviewed the emerging researches of deep learning models for big data feature learning and pointed out the remaining challenges of big data deep learning. In [3], machine learning techniques were applied to predict the performance of MapReduce workloads. The modeling approach consists in correlating the preexecution characteristics of the workload with measured postexecution performance metrics. Considering other parallel computing frameworks, Wang and Khan [8] proposed a prediction model for Apache Spark, which simulates the execution of the actual job by using only a fraction of the input data and collects execution traces (e.g., I/O overhead, memory consumption, and execution time) to predict job performance for each execution stage individually. Chawla et al. [20] evaluated the performance of a cloud workstation from the perspective of the mathematical analysis model, using impact benchmarks such as CPU, internal memory, and network bandwidth to conduct an integrated evaluation of the system impact caused by different parameters in an application, with the use of a comprehensive fuzzy evaluation model.

Queuing models are also used to model the performance for a computing framework. In a queuing model, hardware and software resources are represented by a service center that includes a server and an associated queue. Specifically, this approach operates through two steps: (i) jobs are spawned at a fork node in multiple tasks and then (ii) they are submitted to queuing stations that, in turn, model the available servers. Markov models are used to solve queuing network models, which are based on a representation of the system by a state diagram and capture all possible states that the modeled system may find itself, as well as the possible transitions between such states and the rates at which such transitions occur. However, the limitation of Markov models is the complexity, such that the size of the state space grows exponentially with the number of tasks.

Kavulya and Gandhi [21] predicted Hadoop processs execution time by using behavior analysis of Hadoop users and logistic regression algorithm to measure the similarities of jobs. However, the result accuracy of this approach is unstable and required a large amount of historical data. Ganapathi [3] analyzed the performance of the loaded historical operation information prediction system by using the machine learning algorithm. They extracted historical operation information on jobs and proposed a resource scheduling model that ensures jobs could finish in a specific time. Popescu et al. [23] worked on the issues of execution time prediction on the network-intensive iterative algorithm on MapReduce. However, their study mainly focused on the iterative algorithm that requires representable tuning data in order to achieve high prediction accuracy. Zhang et al. [24] attempted to base distributed computing jobs on heterogeneous machines and to predict job completion time based on boundary-based performance modeling. Its aim is to evaluate the upper and lower limits of task completion time to predict job performance. In [25], the author proposed a performance evaluation model for parallel computing models deployed in cloud centers to support big data applications, such that a big data application is divided into lots of parallel tasks and the task arrivals follow a general distribution.

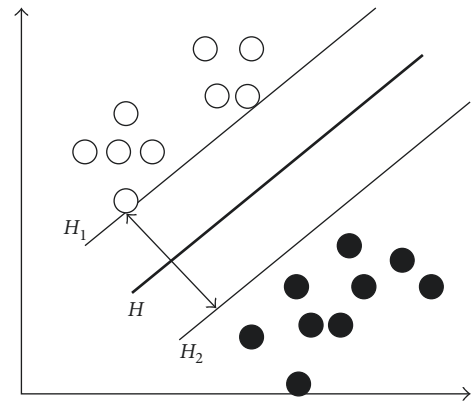

Figure 1: Optimal classification of hyperplane.

3. Analytical Performance Modeling Techniques

In this section, we conduct a detailed analysis to understand analytical performance modeling techniques. Various techniques have been applied to model the performance of computer systems. Superplane segmentation of the training data in the high-dimensional attribute can be achieved to prevent nonlinear surface segmentation calculation in the original input space.

First, the linear binary classification problem is described mathematically as follows:

$$T = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \in (x \times y)^n\}, \quad (1)$$

where $x \in R^n$ is the training sample and $y \in \{-1 + 1\}$ denotes two different categories. For sample T, there is at least one sort line that separates positive and negative samples. This sort line can be described as

$$(w, x) + b = 0. \quad (2)$$

As illustrated in Figure 1, countless dividing lines exist between H_1 and H_2. For the two-dimensional space, we want to find the best divider line. For the n-dimensional space, our ultimate goal is to find the best classification of the superplane, that is, to find the final decision-making boundaries.

For the binary classification problem, the process of finding the largest desired classification line is described as follows: assume that the current direction of law w has been determined. For an arbitrary line parallel to H and between H_1 and H_2, the training data set T can be classified correctly. The classification line is not uniquely determined by W. The sort line is translated to the upper left until it reaches the hollow sample point straight line H_1, right to the bottom of the collision to the solid sample point to obtain a straight line H_2. All the w lines with direction between H_1 and H_2 can be

regarded as classification lines of the training set T. The straight line H in the middle of H_1 and H_2 is chosen as the optimal classification line because it has the largest classification interval for the two types of training samples. The classification of the unknown data tuples is more accurate than that of the other lines, and finally, the problem is transformed into the direction of the solution.

Based on the assumption that the direction of law W has been determined, after normalization, the expressions of H_1 and H_2 are as follows:

$$\begin{aligned}(w \cdot x) + b &= -1,\\ (w \cdot x) + b &= 1.\end{aligned} \tag{3}$$

The expression for H is

$$(w \cdot x) + b = 0. \tag{4}$$

According to the distance between the parallel lines, the distance between H and H_2 is $2/||w||$, where $||w||$ is the norm of the normal vector W. The problem is transformed into a formula under the case of maximization:

$$\min_{\omega,b} \frac{2}{||w||}, \text{ s.t. } y_i(w \cdot x_i + b) \geq 1, \quad i = 1, 2, \ldots, n. \tag{5}$$

To facilitate the Lagrangian operation, the aforementioned solution to the maximum value of the problem is transformed into the following minimum value:

$$\min_{\omega,b} \frac{1}{2}||w||^2, \text{ s.t. } y_i(w \cdot x_i + b) \geq 1, \quad i = 1, 2, \ldots, n. \tag{6}$$

The following Lagrangian function is introduced:

$$L(w, b, a) = \frac{1}{2}||w||^2 - \sum_{i=1}^{n} a_i (y_i (w x_i + b) - 1), \tag{7}$$

where a_i for $i = 1, 2, \ldots, n$ is the Lagrangian multiplier, according to the Lagrangian extreme value of the theorem; that is, for the Lagrangian trapped on the minimum value of W and b, according to the following extreme conditions:

$$\begin{aligned}\nabla_b L(w, b, a) &= 0,\\ \nabla_w L(w, b, a) &= 0,\end{aligned} \tag{8}$$

which can be obtained as

$$\nabla_b L(w, b, a) = \sum_{i=1}^{n} y_i a_i = 0, \tag{9}$$

$$\nabla_w L(w, b, a) = w - \sum_{i=1}^{n} y_i a_i = 0. \tag{10}$$

We can obtain the result after the extreme value of the results in the original simplified form:

$$\begin{aligned}&\max_{\alpha} \sum_{i=1}^{n} a_i - \frac{1}{2} \sum_{i,j=1}^{n} a_i a_j y_i y_j (x_i, x_j),\\ &\text{s.t. } \sum_{i=1}^{n} a_i y_i = 0, \quad a_i \geq 0, \quad i = 1, 2, \ldots, n.\end{aligned} \tag{11}$$

The equivalence of the dual issues is as follows:

$$\begin{aligned}&\min_{\alpha} \frac{1}{2} \sum_{i,j=1}^{n} a_i a_j y_i y_j (x_i, x_j) - \sum_{i=1}^{n} a_i,\\ &\text{s.t. } \sum_{i=1}^{n} a_i y_i = 0, \quad a_i \geq 0, \quad i = 1, 2, \ldots, n.\end{aligned} \tag{12}$$

The optimal solution $a^* = (a_1^*, a_2^*, \ldots, a_n^*)^t$ in the dual problem can be solved by the sequential minimal optimization (SMO) algorithm [26]:

$$w^* = \sum_{i=1}^{n} y_i a_i^* x_i. \tag{13}$$

Finally, the final form of the decision function corresponding to the superplane is

$$f(x) = \text{sign}\left(\left(\sum_{i=1}^{n} y_i a_i^* (x_i \cdot x)\right) + b\right). \tag{14}$$

4. Performance Model

We start this section with a brief review. We propose an evaluation performance model based on SVM in Section 4.1. In Section 4.2, we select and analyze some parameters to evaluate the performance computing framework.

4.1. Evaluation Model Based on SVM. In this section, we apply SVM into the performance evaluation of the MARS and Spark frameworks. We comprehensively evaluate the performance of these two parallel computing frameworks and present representative results of a set of experiments. Experiment evaluation is discussed in the next section.

As the standard of the test indicators are inconsistent and some of the indicators are large, the above indicators are standardized and the indicators are limited to the range [0, 1] to reduce the effect of the numerical size on the statistical results. Suppose there are N items, the kth item of the ith index a_{ki} can be standardized as follows:

$$\overline{a}_{ki} = \frac{a_{ki}}{\max_{1<i<N} a_{ki}}, \quad i = 0, 1, 2, 3, 4. \tag{15}$$

Since execution time, throughput, speedup, resource utilization, and latency are cost functions, then input is $1 - a_{ki}$, for $0 \leq 1 - a_{ki} < 1$. When the five indicators are processed by the kth project, we denote execution time, throughput, speedup, resource utilization, and latency as PI_{k0}, PI_{k1}, PI_{k2}, PI_{k3}, and PI_{k4}.

Suppose there are several linear separable samples x and their classes y, which denote $\{(x_i, y_i)\}$, $i = 1, 2, \ldots, n$, $x_i \in R^d$, where n is the number of training samples, d is the dimension of the training sample, and $y \in (-1, +1)$ is the category label. For a nonlinear separable sample, the linear problem is mapped from the original space to a high-dimensional space by the nonlinear mapping Φ, and then, the optimal classification surface is obtained in the high-dimensional space. According to the functional theory, the kernel function $K(x_i, y_i)$ corresponds to the inner product

of a certain transformation space, which just needs to stratify the Mercer condition, such that $K(x_i, y_i) = \Phi(x_i)\Phi(y_i)$ without knowing the specific form of $\Phi(x_i)$. In the original space, the class equation $\Phi(x_i) = 0$ should satisfy the constraint

$$y_i\left[w\Phi\left(x_i\right) + b\right] - 1 + \xi_i, \quad i = 1, 2, \ldots, n, \tag{16}$$

where w is the weight vector for the classification plane, ξ_i represents the slack variable, and b is the threshold. The optimal classification surface is to maximize $2/||w||$; thus, the optimal classification surface is transformed into the minimum value of $(1/2)w^2 + c\sum_{i=1}^{n}\xi_i$ under conditional constraints, which is considered as the minimum misclassified sample and the maximum classification interval when determining the optimal classification surface, where the constant $c > 0$ is the penalty function, which controls the degree of penalty for the misclassified sample. This problem is transformed into a simpler duality problem by the Lagrange function, which is intended to compute the maximum value of function under constraints of $\sum_{i=1}^{n} y_i a_i = 0$ and $0 \le a_i \le c$, $i = 1, 2, \ldots, n$:

$$L_{\mathrm{D}}(a) = \sum_{i=1}^{n} a_i - \frac{1}{2}\sum_{i,j=1}^{n} a_i a_j y_i y_j K\left(x_i, y_j\right), \tag{17}$$

where a_i represents the multiplier or the optimization factor of Lagrange, which must satisfy $a_i y_i [w\Phi(x_i + b)] - 1 + \xi_i = 0$, $i = 1, 2, \ldots, n$. A sample corresponding to $a \neq 0$ is called a support vector sample (SVs). It is the support vector sample that determines the final classification result. For a given test sample x, the optimal classification function of the support vector machine classifier is

$$f(x) = \mathrm{sign}\left[\sum_{\mathrm{SVs}} a_i y_j K\left(x_i \cdot x\right) + b\right], \tag{18}$$

where $\mathrm{sign}(\cdot)$ is the symbolic function. The sign of the function $f(x)$ determines the category to which the subordinate belongs, which is the support vector machine. The establishment of the SVM model based on parallel computing frameworks is described as follows:

(1) Fuzzy scoring the computation frameworks according to the five parameters $\mathrm{PI}_{k0}, \mathrm{PI}_{k1}, \ldots, \mathrm{PI}_{k4}$ put the performance of the calculation framework into five levels, and each level has a different value: 1 is very bad, 2 is bad, 3 is average, 4 is good, and 5 is excellent.

(2) Selecting a number of computing frameworks that have been graded by experts as learning samples (x_i, y_i), where x_i is a 5-dimensional vector. Let $\mathrm{PI}_{k0}, \mathrm{PI}_{k1}, \ldots, \mathrm{PI}_{k4}$, y_i be a 5-dimensional vector which denotes five performance levels of the computing framework.

(3) Selecting the appropriate kernel function and the parameter value and obtaining the classification discriminant function by using the SVM for sample training.

(4) Evaluating the performance of the computing frameworks by using the obtained classification function.

The tasks can be divided into multiple stages, with each stage containing multiple tasks. This can be represented by (8) and (9):

$$\begin{aligned} \mathrm{job} &= \mathrm{stage}_i | 0 \le i \le M, \\ \mathrm{stage}_i &= \mathrm{task}_{i,j} | 0 \le j \le N, \end{aligned} \tag{19}$$

where M is the number of stages in the job, and each job is labelled with i. N is the number of tasks in the ith stage.

This calculation process enables prediction to execution time by the SVM. We assess the prediction ability of the forecasting function by using the following estimated methods: mean absolute error: $\mathrm{MAE} = \left(\sum_{i=1}^{m}|a_i - p_i|\right)/m$, estimated mean square error: $\mathrm{MSE} = \left(\sum_{i=1}^{m}|a_i - p_i|^2\right)/(m-1)$, and goodness-of-fit determination coefficient: $R^2 = \left(\sum_{i=1}^{m}(p_i - \overline{a}_i)^2\right)/\left(\sum_{i=1}^{m}(a_i - \overline{a}_i)^2\right)$, where a_i is the actual value, $\overline{a}_i$ is the mean value, and p_i is the prediction value.

4.2. Results of the Evaluation Model. In this section, we use multifactor line regression (MLR) to provide a comparison. As for MLR, we build a single hidden-layer neural network. At first, the network is initialized with random numbers. We construct the evaluation model by tuning sample data to solve for the benchmark's corresponding parameter β. Execution time is affected by the number of cores used (cores), the amount of data (nums), and internal memory (mem). The standard coefficient of each factor's time effect to logistic regression is listed in Table 1, and Bayesian Classification and PageRank are listed in Tables 2 and 3.

From Table 1, it can be observed that, for logistic regression, the benchmark "cores" account for the biggest value. It means that more number of CPU cores gives the less consumption of execution time.

Similar to logistic regression, the parameters of the Bayesian classification are illustrated in Table 2. The same number of CPU cores is needed for the Bayesian classification.

As shown in Table 3, the iterative computing PageRank, benchmark "memory" accounts for the biggest value. Thus, we can conclude that the more the amount of internal memory allocated, the shorter the execution time of the job. From Table 4, HiveQL query takes up a lot of memory resources to interact with data from multiple data sources (Table 5).

The prediction samples are imported into this model. These prediction samples sizes are 10G, 20G, and 50G. The number of cores used was 6, and the internal memory was 30G. The experiment showed that the predicted execution time for both types of the job is within the prediction range. Results from repeated experiments showed that the predicted job execution time has an error within 10% when compared with the mean value.

We compared the results from the evaluation model based on SVM with those from the MLR model by computing the estimated methods. The remarkable advantage of

Table 1: Related parameters for logistic regression.

Benchmark	t-test	Significance
Constant	0.296	0.002
x_1: cores	3.429	0.001
x_2: nums	0.517	0.001
x_3: mem	−2.153	0.000

Table 2: Related parameters for the Bayesian classification.

Benchmark	t-test	Significance
Constant	0.512	0.002
x_1: cores	3.611	0.003
x_2: nums	0.528	0.001
x_3: mem	−1.689	0.000

Table 3: Related parameters for PageRank

Benchmark	t-test	Significance
Constant	0.401	0.002
x_1: cores	6.151	0.001
x_2: nums	0.597	0.001
x_3: mem	−3.021	0.000

Table 4: Related parameters for HiveQL queries.

Benchmark	t-test	Significance
Constant	0.351	0.001
x_1: cores	4.394	0.001
x_2: nums	0.709	0.000
x_3: mem	−5.591	0.000

Table 5: Spark's speedup over MARS on running logistic regression.

Input	10G	20G	30G	40G	50G
Spark	1240	2041	2482	3620	5218
MARS	1511	2315	2913	3110	4258
Speedup	1.21	1.13	1.17	0.85	0.81

the SVM is that MLR can suffer from multiple local minima, and the solution to an SVM is global and unique. Two more advantages of the SVM is that it has a simple geometric interpretation and gives a sparse solution. Unlike MLR, the computational complexity of the SVM does not depend on the dimensionality of the input space. MLR uses empirical risk minimization, while the SVM uses structural risk minimization. The results are shown in Table 6, in which the evaluation model based on SVM has a higher fitting degree. Experiment results show that the SVM has a higher accuracy than MLR in predicting execution time.

5. Experiment Evaluation

In this section, we present the experiment results, which evaluate and characterize MARS and Spark, in terms of execution time (i.e., job running time), throughput (i.e., the number of tasks completed per minute), speedup (i.e., ratio of the execution time), system resource (e.g., CPU, memory, and I/O) utilizations, and latency.

Table 6: Related parameters for different models.

Model	MSE	MAE	R^2
SVM	2.646	2.015	1.521
MLR	3.121	2.809	1.271

Table 7: Benchmarks used in the experiment and their categories.

Category	Benchmark	MARS	Spark
Machine learning	Logistic regression	√	√
	Bayesian classification	√	√
Web search benchmarks	PageRank	√	√
SQL engine	HiveQL queries	√	√

5.1. Benchmark Model. In this section, benchmarks including logistic regression, Bayesian classification, PageRank, and HiveQL queries are classified into four categories as shown in Table 7. All test data are created by the data generator provided in HiBench suite [27]. HiBench is a new, realistic, and comprehensive benchmark, which is suited for Hadoop and consists of a set of Hadoop programs including both harvest microbenchmarks and real-world applications. We use HiBench in the Hadoop framework as a standard of execution time, throughput, speedup, system resource utilization, and latency.

The Bayesian classification and PageRank are provided as examples in MapReduce library files, and the Spark versions are provided by HiBench [27]. Logistic regression [2] and Bayesian classification are widely used regression, classification, and recommendation algorithms for machine learning applications. Logistic regression (LR) is a popular method to predict a categorical response. It contains three different kinds of data types, including categorical data, continuous data, and binary data. The Bayesian classification workload implements the trainer part of naive Bayesian. Naive Bayes is a multiclass classification algorithm with the assumption of independence between every pair of features. PageRank is a representative and popular graph computation algorithm. HiveQL is compiled into MapReduce jobs executed on Hadoop, which is proposed to support queries expressed in a SQL-like declarative language. It is mainly used to simplify storing and accessing big data.

5.1.1. Execution Time. Execution time is one of the indicators to evaluate the performance of a program. In MARS, every job reads its input data, processes them by using caches of CPU and GPU, and then writes them back to HDFS. The next job is to repeat the read process and write cycle after the previous job done. Spark implements in-memory data structures to cache intermediate data of nodes by using RDDs function. The execution time can reflect the execution efficiency of different frameworks in the same batch of data.

5.1.2. Throughput. The system throughput or aggregate throughput is the sum of the data rates that are delivered to all terminals in the network [28]. Throughput is essentially synonymous to digital bandwidth consumption, and it can

be analyzed mathematically by applying the queuing theory, in which the load in packets per time unit is denoted as the arrival rate (λ) and throughput in packets per time unit is denoted as the departure rate (μ). MARS puts data in the caches of GPU and CPU, while Spark receives and processes data in the main memory which keeps data in memory for fast accesses to achieve high performance and high throughput. Without the bottleneck of performance, there is a certain link between throughput and the number of virtual users, which can be calculated by the following formula:

$$S = \frac{\sum_{i-1}^{N} T}{k_i}, \tag{20}$$

where S denotes the throughput of the system, k_i is the execution times of the ith operation, T represents the whole running time of the system, and N represents the number of virtual users.

As a critical component in many Internet service systems, such as Facebook, YouTube, and Twitter, data processing is critical to provide quality services to end users with high throughput.

5.1.3. Speedup. We define the speedup as the ratio of the execution time on MARS to that on Spark. In computer architecture, speedup is a process for increasing the performance between two systems processing the same problem. The notion of speedup was established by Amdahl's law, which particularly focused on parallel processing [29]. However, speedup can be used more generally to show the effect on performance after any resource enhancement. After obtaining the parallel execution time and the percentage of serial time, the speedup can be achieved by replacing these values into the formula as

$$\text{Speedup} \leq \frac{1}{(1 = \text{pctPar}) + (\text{pctPar}/p)}, \tag{21}$$

where pctPar is the percentage of execution time that can be parallel code and p is the number of processor cores in parallel program runtime. When calculating the speedup ratio, the serial execution time is normalized as 1 in this formula. The parallel execution time is the denominator, equal to the percentage of serial time $(1 - \text{pctPar})$ plus the percentage of execution time of the parallel code divided by the number of processor cores (pctPar/p).

5.1.4. Resource Utilization. Resource utilization refers to the proportion of the different resources of the system (such as CPU, memory, network bandwidth, and others) in the occupied state of the average time. The utilization indicator can reflect the use of the system resources throughout the operation and accurately describe the use of the system resources. Let k_i denotes the execution times of the ith operation, and then the ratio of resource utilization E_j of j is

$$E_j = \frac{\sum_{i=1}^{N} k_i O_{ij}}{T}, \tag{22}$$

where O_{ij} is the time of possession of resources of j by the ith operation and T is the whole running time of the system.

5.1.5. Latency. Spark only supports a small number of outstanding memory accesses, and an increasingly long delay is spent on waiting for data to be fetched from memory. Consequently, the high parallelism of query processing is hard to be explored to hide the memory access latency. MARS provides simple and many computing units for massive data parallel operations supported by high memory bandwidth. Extremely fast context switch among threads in a warp can help tolerate memory access latency.

We use I/O cost, memory cost, and CPU cost to measure running tasks, in which I/O cost depends on the performance of disk, such that the higher the performance of disk, the less the time in reading and writing the same data. The most complex costs in MARS are CPU costs appearing in data parsing, executing Map function or Reduce function, data serializing, and data sorting. Generally, there is more memory cost in the Spark task; thus, how to accurately measure and estimate this cost is one of the most challenges for modeling the MARS and Spark.

5.2. Experiment Descriptions. We perform MARS and Spark on a cluster of three servers with the same physical configurations, with one master and two slave nodes. The configuration of each server is as follows: Nvidia GTX1060 GPU, Intel Xeon E5-2670 16 processors running at 3.3 GH, 1 disk with 6 TB each, and 64 GB of physical memory. Thus, our cluster has 48 cores CPU, 18 TB locally attached storage, and 192 GB RAM. The source code of implementations can be downloaded from https://github.com/hanzhijie/XinyuLv.

In this paper, we run these four applications on MARS and Spark, respectively. Our experimental data set was downloaded from https://github.com/intel-hadoop/HiBench. The five parameters execution time, throughput, speedup, resource consumption, and latency draw column charts by using the data based on the results consistent with the premise.

5.3. Results on Benchmarks. For the shuffle stage in the execution of tasks, we first analyze the shuffle process in MARS and Spark. Shuffle stage is an operator that expresses many-to-many dependencies, which is the link between the Map phase and the Reduce phase in MARS, such that the Reduce task reads one data from each Map task. Shuffle is typically divided into two parts: the Map phase of the data preparation and the Reduce phase of the data copy. MARS is mainly used for large data Map/Reduce operation in parallel processing. Massive threads of GPU are assigned for the Map and Reduce tasks during the parallel processing, where each thread is processed as key/value pairs. In MARS, data are input into the main memory in the form of key/value pairs, and the key/value pairs are copied into graphics memory while starting GPU computing and then performing Map/Reduce operation. Driver program and Worker node play key roles in Spark, in which the Driver program is the

starting point for the implementation of Spark application and used for operations and allocations of tasks. Multiple workers control the computation node and create executor parallel processing tasks. At the beginning of processing, the Driver program distributes the task and the file, and the compression package depended on the task on the corresponding Worker nodes. Meanwhile, executor handles the tasks of the corresponding data partition.

5.3.1. Logistic Regression. Logistic regression is the most commonly used machine learning algorithm for categorizing data and building recommendation systems. As a machine learning classifier, it can be used to predict continuous or categorical data. The algorithm uses the stochastic gradient descent to train the classification model.

The implementation of MARS is similar to that of MapReduce based on CPU, which is composed of two stages of Map and Reduce. For Map stage, the input data are divided into multiple blocks by split operation so that the number of blocks is equal to the number of threads. In such way, a GPU thread just only responds to a single piece of the same number of data so that each thread of GPU can be load balanced. After each Map task is completed, the results are passed to the merge operation and added. For Spark, the input data set is kept in the memory through RDD abstractions, and the parameter vector is calculated, updated, and broadcast in each iteration. The input data points are assigned to clusters with a closest centroid, and new centroids are created by these points assigned in the clusters. These steps are repeated until it converges. Each time, MARS needs to store the intermediate results back to the disk. However, Spark keeps data in memory. When the data size is smaller than 30G, Spark has improvements on throughput than MARS, which is up to 1.5 times. Figure 2 represents an execution time performance comparison for logistic regression between Spark and MARS. When the file size is greater than 40G, Spark has an obvious decrement as represented by Figure 3.

Spark introduces RDD (resilient distributed data set) model in which intermediate data are stored in the form of RDD, and RDD stored is distributed in the memory of the slave node, which reduces the number of disk reading and writing in the calculation process. RDD also provides a cache mechanism, which reduces MR2 and MR3 repeat reading the same data with respect to MapReduce. Spark has better performance than MARS in that the speedup is up to 1.05 times as shown in Table 8. However, the advantage is bounded by the memory. The speedup decreases when the data are more than 40G and has a minimum value with 0.81 times when the input is 50G.

As shown in Figure 4, Spark saves more CPU resource than MARS especially with small data size. The maximum CPU consumption for it is only half of that of MARS with the input equal to 10G and 20G. The memory usage for Spark is smaller than that of MARS when the data size is 10G and 20G. As shown in Figure 5, the memory usage for Spark is almost 80% with 40G and 50G input that Spark cannot create more RDDs at that point.

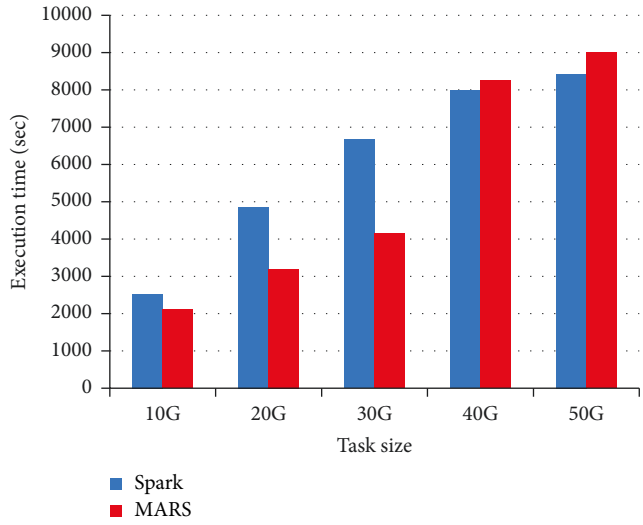

Figure 2: Execution time of logistic regression.

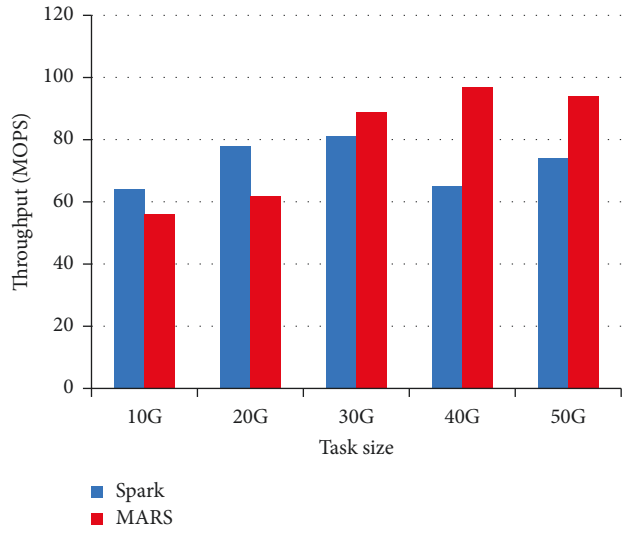

Figure 3: Throughtput of logistic regression.

Table 8: Spark's speedup over MARS on running HiveQL queries.

Input	10G	20G	30G	40G	50G
Spark	210	231	252	273	72.9
MARS	84.1	85.6	84.9	178	210
Speedup	0.54	0.49	0.41	0.39	3.09

Figure 6 illustrates the latency comparison between Spark and MARS. Spark has lower latency than MARS with the increase of data sizes. Spark is suitable for iterative calculation, while MARS is suitable for batch calculation. Spark keeps a gentle low delay because of keeping data in memory. For the shuffle stage, MARS uses the CPU and GPU caches instead of memory, and the data transfer between I/O caused a significant amount of delay.

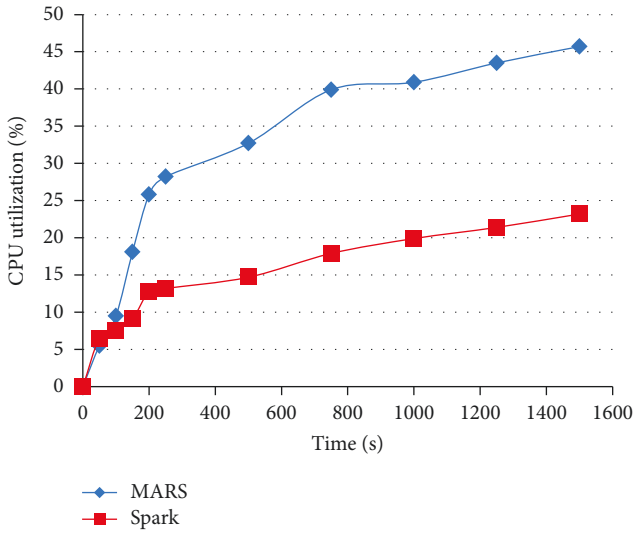

Figure 4: CPU usage of logistic regression.

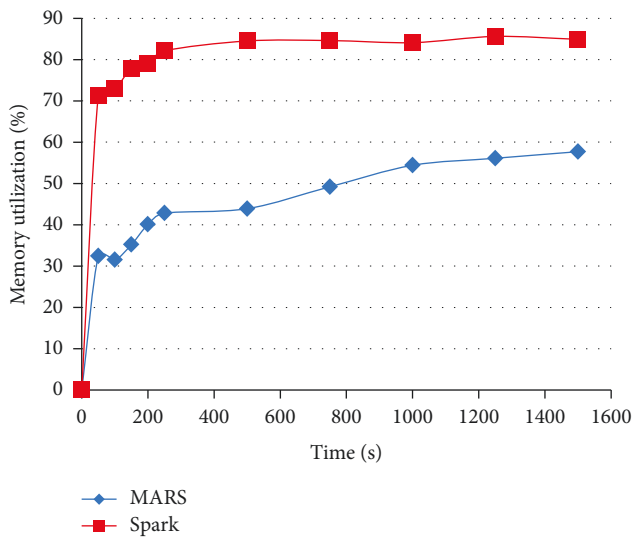

Figure 5: Memory usage of logistic regression.

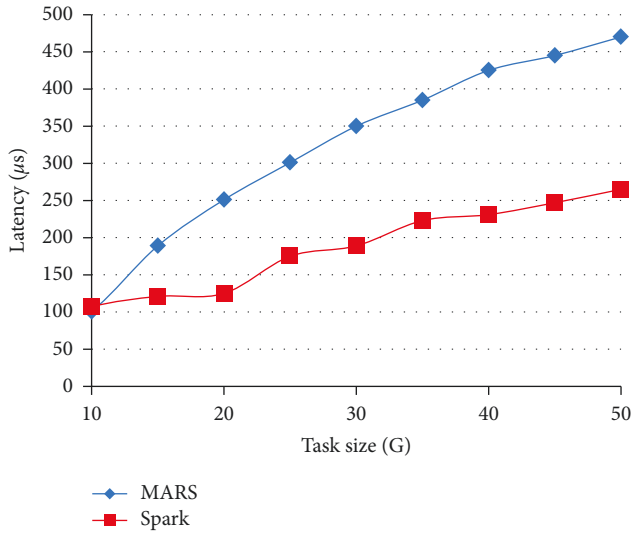

Figure 6: Latency of logistic regression.

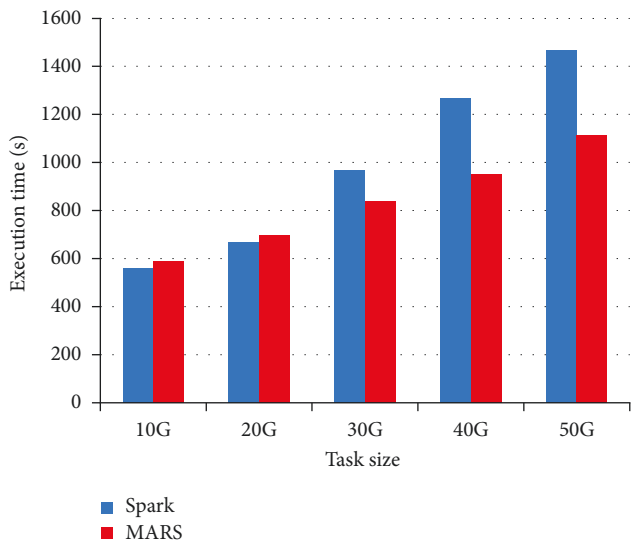

Figure 7: Execution time of Bayesian classification.

5.3.2. Bayesian Classification. Bayesian classification is an effective classification algorithm which has been widely used. The Bayesian classification workload contains four chained Hadoop jobs, and the first job is the most time-consuming (taking about half of the total running time in the experiment). The performance of execution time comparison for running the Bayesian classification on Spark and MARS is shown in Figure 7.

The task size ranges from 10G to 50G. It trains a classification model for the input data set using the stochastic gradient descent. As the input size increases, the execution time of Spark and MARS increases. The Bayesian classification benchmark runs without iterations. Each task in the input task would be assigned a key, and all the data with the identical keys would be counted together. MARS is about 20% higher than Spark in CPU usage, and 30G higher in memory usage. The throughput change for MARS is not obvious with the increase of inputting data, while the throughput for MARS increases as the input size becomes larger at up to 30 MOPS than Spark as shown in Figure 8. Massive threads of GPU are assigned for the mapping and reduction tasks Map/Reduce operation, and small and equal number of keys/values are processed by each thread so that each thread of the GPU can load balance. Finally, data processing performance is optimal, and the performance of data processing is improved efficiently. Furthermore, MARS provides a better performance in processing logistic regression with speedup of up to 0.91 times as shown in Table 9.

The speedup is affected by the input task size. The speedup decreases when the input data size becomes larger. In the processing of the Bayesian classification, both MARS and Spark need high CPU consumptions as shown in Figure 9.

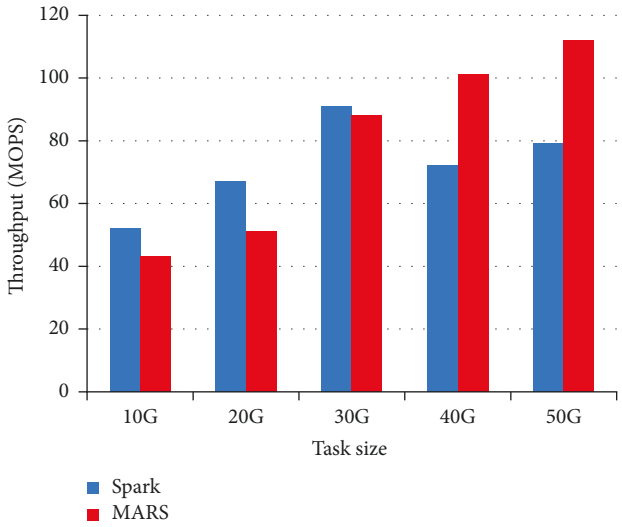

Figure 8: Throughput of Bayesian classification.

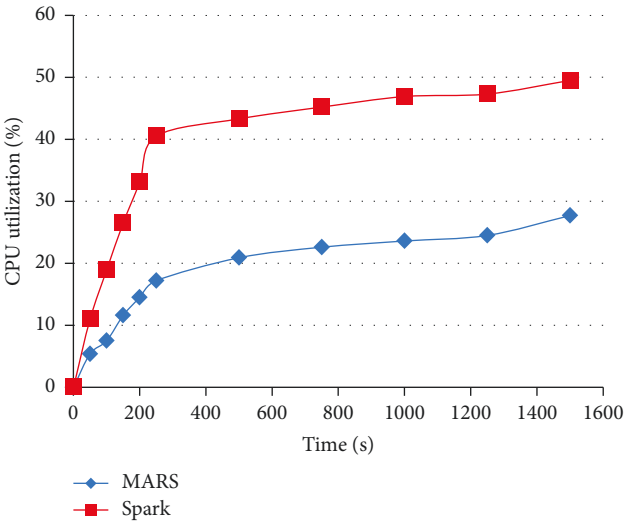

Figure 9: CPU usage of Bayesian classification.

Table 9: Spark's speedup over MARS on running the Bayesian classification.

Input	10G	20G	30G	40G	50G
Spark	1021	1564	1971	2677	4923
MARS	851	932	992	2128	4491
Speedup	0.83	0.60	0.50	0.79	0.91

MARS has outstanding performance on saving memory resource. In Figure 10, Spark consumes larger memory resource with the time increase. For MARS, the key/value pairs are written back to the disk, and massive GPU threads are assigned to each task. Since GPU does not support dynamic memory allocation, the input data and results are allocated in the graphics memory. This operation saves a lot of memory. Figure 11 represents the overall latency for the Bayesian classification for various input sizes for both Spark and MARS. It has become a bottleneck of data processing with the bandwidth of memory and CPU cache, while GPU has many computing cores that could provide a device memory with high bandwidth. With Spark's DAG programming model, seven MapReduce tasks can be reduced to one Spark task. Spark divides the task into eight stages automatically, and each stage contains multiple tasks that could be executed in parallel. The data among the stages are passed through shuffle and eventually read and write from HDFS only once. Spark saves 65% time to read and write from HDFS than MARS.

5.3.3. PageRank. PageRank is a graphical algorithm that sorts the elements by calculating the number and quality of the links. In MARS, a lot of time is spent on the operation of Map because it cannot be used directly. In order to be able to iterate, it can only format the output. As the amount of data is increasing, there will be a lot of resources to be delivered in the Map phase. Since Spark uses RDDs to represent data structures, it just only needs one stage per iteration.

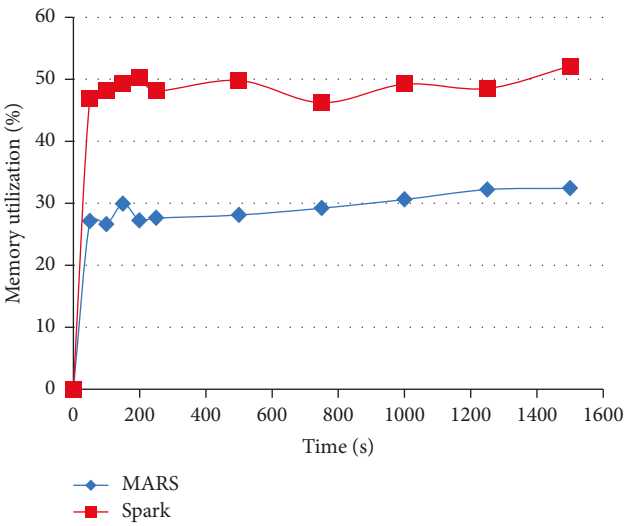

Figure 10: Memory usage of Bayesian classification.

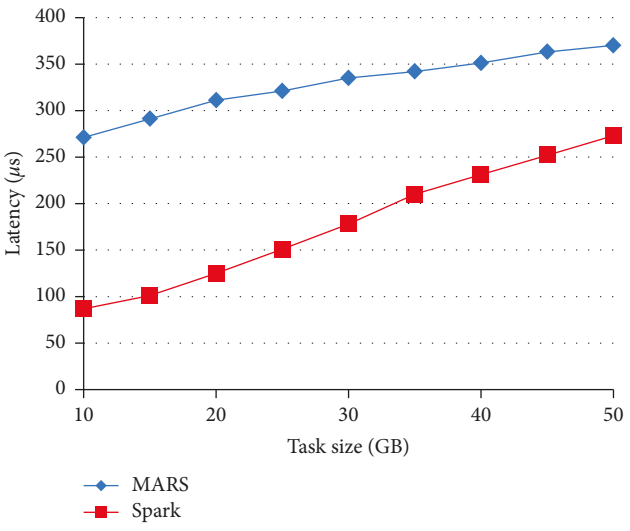

Figure 11: Latency of Bayesian classification.

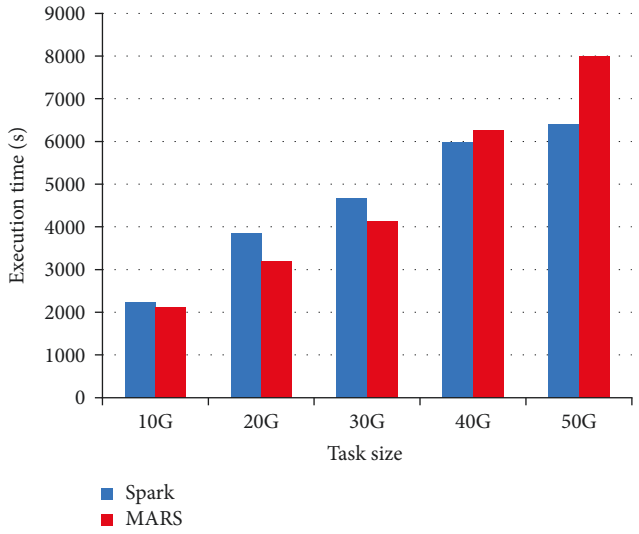

Figure 12: Execution time of PageRank.

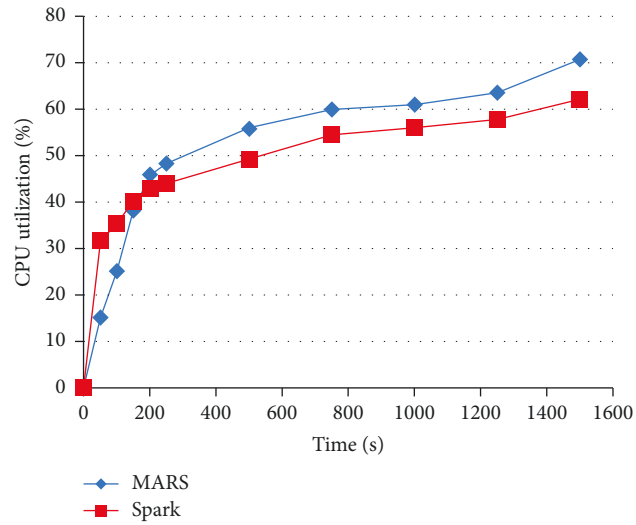

Figure 14: CPU usage of PageRank.

Table 10: Spark's speedup over MARS on running PageRank

Input	10G	20G	30G	40G	50G
Spark	16.2	24.1	39.6	71.1	111.9
MARS	13.2	32.6	41.5	91.5	287.1
Speedup	0.81	1.35	1.05	1.28	2.56

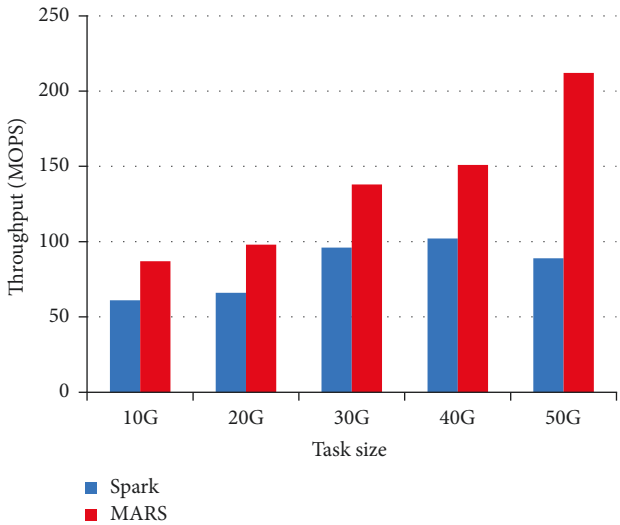

Figure 13: Throughput of PageRank.

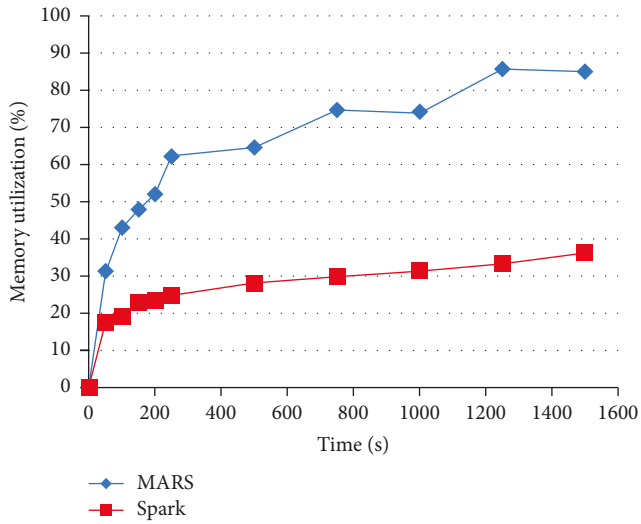

Figure 15: Memory usage of PageRank.

Figure 12 represents the performance comparison of execution time for implementing PageRank between Spark and MARS. Table 10 sums up the MARS's speedup over Spark on the same file sets. The graph data size ranges from 10G to 50G. Spark performs better than MARS with the file size lower than 30G. However, MARS beats Spark when the file size is greater than 40G. The maximum speedup is 2.56 times. The throughput decreases as the file size increases for Spark as shown in Figure 13, while the throughput remains constant for MARS when the data size is increased.

For Spark, PageRank demands fewer CPU resource, and it saturates the memory resource during the execution. The stable use of memory is due to the vertex and edge of the PageRank-cached memory, as an input to each iteration of the algorithm. Figure 14 illustrates the comparison of CPU usage between Spark and MARS. To implement PageRank workload, Spark consumes more memory resources and costs less CPU resources than MARS. When the batch size is small, the total GPU execution and data transfer overhead is high. For MARS, PageRank spends much time on the iterations of several jobs, and these jobs are generally CPU bound, with low memory utilizations as shown in Figure 15. MARS has a higher latency than Spark for 30G input tasks, which has approximately 1.6 times latency than Spark for

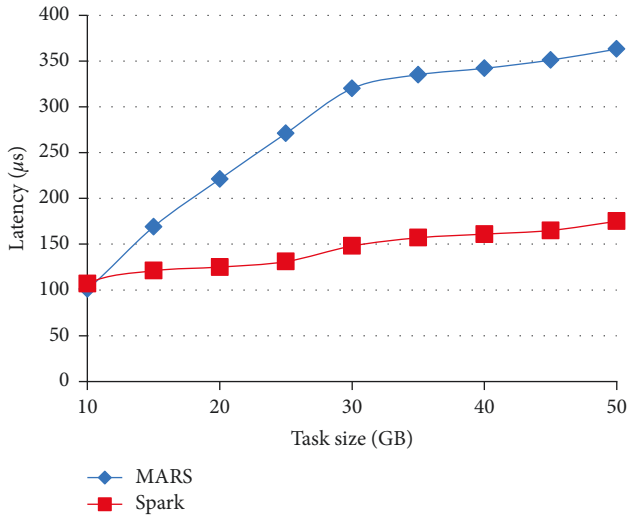

Figure 16: Latency of PageRank.

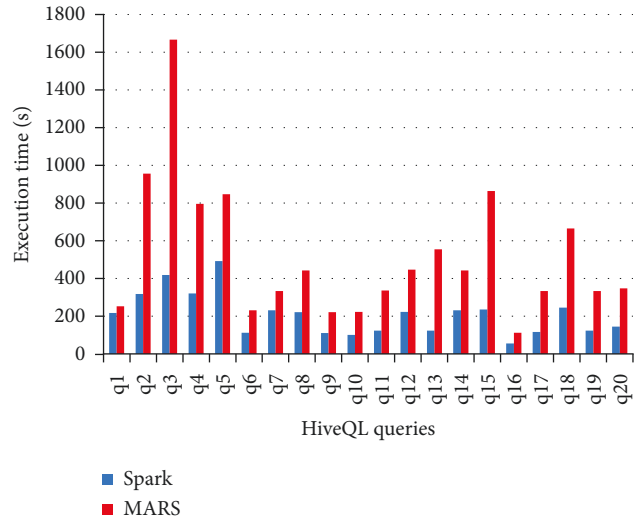

Figure 17: Execution time of HiveSQL.

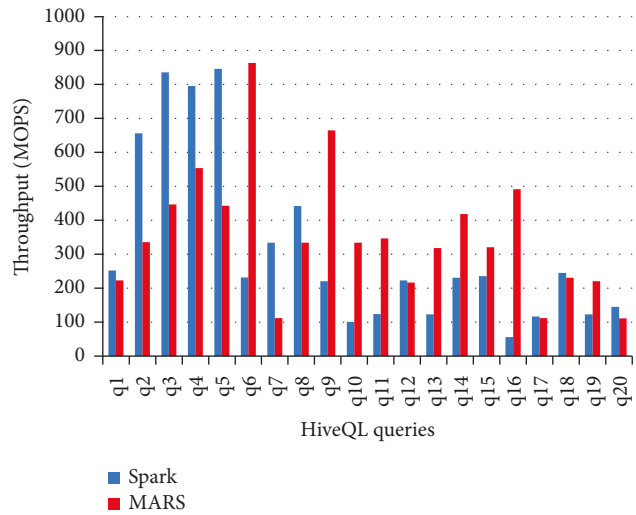

Figure 18: Throughput of HiveSQL.

50G tasks. In the processing of MARS, GPUs need data batching to improve resource utilization. A small-sized task for GPU threads leads to a low throughput, and a large batch size leads to a high latency.

Spark job will apply for the required executor resources, all tasks stages running in the way of threads and sharing executors. Relative to MapReduce, the number of Spark application resources is reduced by nearly 90%. The initialization thread configuration of MARS is included the number of threads on the GPU group and the number of threads per thread group before starting each stage. MARS use massive threads in GPU and assign tasks to threads uniformly at the runtime phase. Each thread is responsible for one Map or Reduce task and manages the MapReduce framework of concurrent write through a lock-free solution with a small number of key/value pairs as input operation. Figure 16 represents the comparison of latency between Spark and MARS. Spark has lower latency than MARS when the data size increases. It is because that MARS performs segmentation and processing data in the GPU cache, in which GPU has a higher parallelism and a large number of threads than CPU while processing data. All the data are placed and processed in the memory in Spark, and the rest of the data had to be stored in the disk when the memory is full. The rest of the data are placed in memory from the disk by CPU instructions, and this will lead to a large time delay of I/O; thereby, the processing speed is reduced as the amount of data increases.

5.4. SQL Engine. Apache Hive supports a SQL-like query language known as the Hive query language over one or multiple data files located either in a local file system or in HDFS. We evaluate Hive on MARS and Spark by selecting 20 different types of queries from BigBench [30]. As the size of data increases from 10G to 50G, all of 20 queries on Spark run faster than those on MARS. As illustrated in Figure 17, in some queries (q2, q3, q4, q5, and q15) which contain "aggregation" operation among big tables or relate to iterative computing, Spark performed faster, for instance, q4 being 4 times faster than on MARS. As seen for Figure 18, Spark has higher throughput than MARS in queries q1, q2, q3, q4, and q5. While MARS has a higher throught than Spark in some queries (from q9 to q16), which related to recursive calculation.

Table 8 summarizes the MARS's speedup over Spark on the same file sets, and the maximum speedup is 3.09 times. As shown in Figure 19, MARS consumes more CPU resource than Spark running on most queries, but the utilization of memory of Spark is much greater than that of MARS, for instance, the results of CPU utilization of query 6. Both Hive on Spark and MARS utilized on average 64% CPU. However, Spark utilized much more of memory, which is 72%, while Hive on MARS maximum used only 40%. Especially, Spark used 50% of the memory for caching data according to Figure 20.

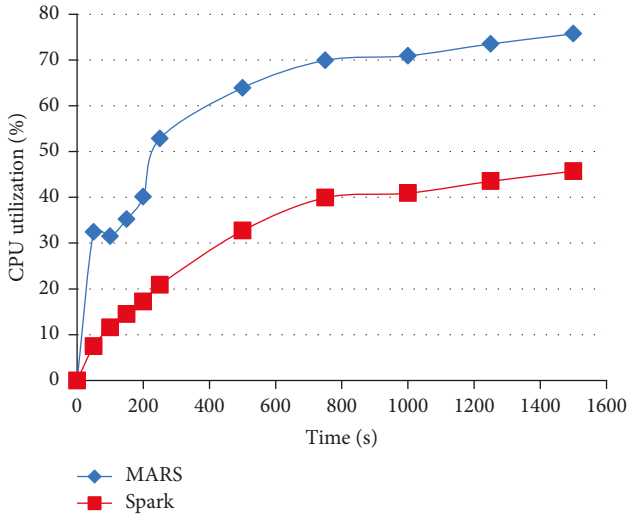

Figure 19: CPU usage of HiveSQL.

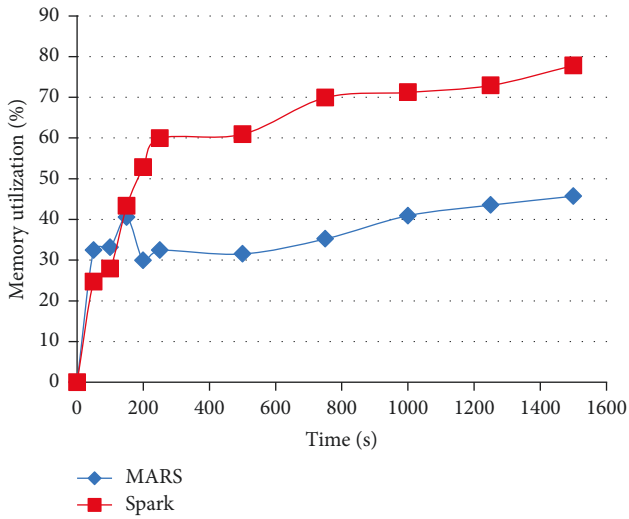

Figure 20: Memory usage of HiveSQL.

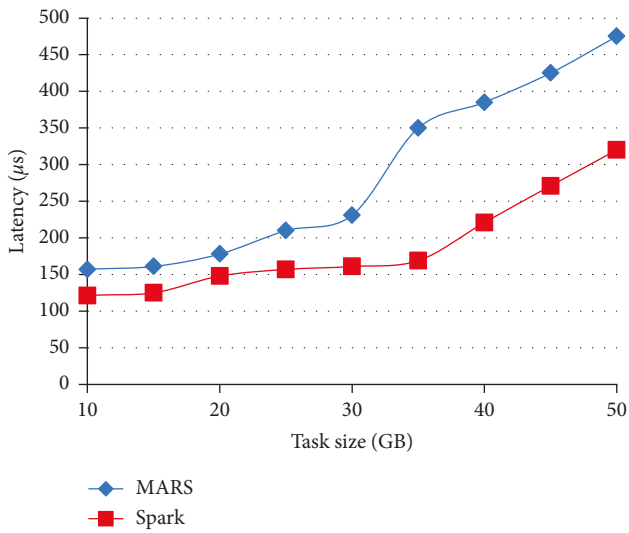

Figure 21: Latency of HiveSQL.

Figure 21 represents a latency performance comparison for the HiveQL query benchmark between MARS and Spark. For MARS, it takes a long time from query submission to the result return, in which a large number of GPU threads interact with the disk, and the query takes a lot of time. When Spark executes HiveQL queries, both filter and join queries are stored in memory, which can effectively reduce the latency of interaction with I/O.

6. Conclusion

In this paper, we propose an evaluation performance model based on SVM for the distributed computing framework and apply it to MARS and Spark. There are two major steps to build the performance model: the first step is selecting a number of systems that have been scored by experts as the sample data set. After each time of mapping tasks, the resource utilization and execution time are stored in a vector and later used as a sample data set. The second step is performance modeling. The models are built by inputting the matrix of resource utilization and execution time. After comparison of the performance evaluation model based on SVM with MLR, the result shows that it has higher accuracy and ensures the objectivity and scientificity of the evaluation. The proposed evaluation model based on SVM has good classification ability and learning generalization ability for small sample data, and the evaluation process will become more complicated with the increase of performance indicators. There is no general solution to the nonlinear problem for the proposed evaluation model due to the sensitivity of SVM to missing data, and thus the selection of kernel function must be cautious. Our future work is to improve the expansibility of the traditional SVM and implement the multiclassification SVM.

There are four benchmarks for comparing the performance of these two frameworks, which conclude that MARS has better performance than Spark in batch processing, while Spark has better performance in running PageRank because it is good at iteration of the same data set. The current parallel processing frameworks such as Phoenix and Disco were both developed by MapReduce based on CPU. Our future work will consider the coprocessing with MARS and Spark to achieve a higher parallel computing capability.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

This study was sponsored by the National Natural Science Foundation of China (nos. 61672296, 61672209, and 61602261), the Natural Science Foundation of Jiangsu Province (nos. BK20160089 and BK20140888), Scientific and Technological Support Project of Jiangsu Province (nos. BE2016777 and BE2016185), and Jiangsu High Technology Research Key

Laboratory for Wireless Sensor Networks Foundation (no. WSNLBKF201701).

References

[1] Apache Hadoop, http://hadoop.apache.org/.

[2] J. A. Issa, "Performance evaluation and estimation model using regression method for Hadoop WordCount," *IEEE Access*, vol. 3, pp. 2784–2793, 2015.

[3] A. Ganapathi, H. Kuno, U. Dayal et al., "Predicting multiple metrics for queries: better decisions enabled by machine learning," in *Proceedings of the International Conference on IEEE Data Engineering*, pp. 592–603, Shanghai, China, April 2009.

[4] B. He and W. Fang, "Mars: a MapReduce framework on graphics processors," in *Proceedings of the 17th International Conference on ACM Parallel Architectures and Compilation Techniques (PACT 2008)*, pp. 260–269, Toronto, ON, Canada, 2008.

[5] M. Zaharia, M. Chowdhury, and T. Das, "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, San Jose, CA, USA, April 2012.

[6] R. E. Bryant, R. H. Katz, and E. D. Lazowaska, *Big-Data Computing: Creating Revolutionary Breakthroughs in Commerce, Science and Society*, Academic Press, Cambridge, MA, USA, 2008.

[7] K. Zhang, K. Wang, and Y. Yuan, "Mega-KV: a case for GPUs to maximize the throughput of in-memory key-value stores," *Proceedings of the VLDB Endowment*, vol. 8, no. 11, pp. 1226–1237, 2015.

[8] K. Wang and M. M. H. Khan, "Performance prediction for Apache Spark platform," in *Proceedings of the International Conference on IEEE High PERFORMANCE Computing and Communications*, pp. 166–173, Bangalore, India, December 2015.

[9] G. Wang, A. R. Butt, P. Pandey, and K. Gupta, "Using realistic simulation for performance analysis of MapReduce setups," in *Proceedings of the 1st ACM Workshop on Large-Scale System and Application Performance*, pp. 19–26, Garching, Germany, 2009.

[10] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proceedings of the VLDB Endowment*, vol. 4, pp. 1111–1122, 2011.

[11] A. Kahol, S. Khurana, S. K. S. Gupta, and P. K. Srimani, "A strategy to manage cache consistency in a disconnected distributed environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 7, pp. 686–700, 2001.

[12] A. K. Seghouane, "New AIC corrected variants for multivariate linear regression model selection," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 47, no. 2, pp. 1154–1165, 2011.

[13] D. Singh, D. Roy, and C. K. Mohan, "DiP-SVM: distribution preserving kernel support vector machine for big data," *IEEE Transactions on Big Data*, vol. 3, no. 1, pp. 79–90, 2017.

[14] J. Berliska and M. Drozdowski, "Scheduling divisible MapReduce computations," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 450–459, 2011.

[15] E. Vianna, G. Comarela, and T. Pontes, "Analytical performance models for MapReduce workloads," *International Journal of Parallel Programming*, vol. 41, no. 4, p. 495, 2013.

[16] N. Yigitbasi, T. L. Willke, G. Liao, and D. Epema, "Towards machine learning-based auto-tuning of MapReduce," in *Proceedings of 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 11–20, San Francisco, CA, USA, August 2013.

[17] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "PPHOPCM: privacy-preserving high-order possibilistic c-Means algorithm for big data clustering with cloud computing," *IEEE Transactions on Big Data*, vol. 99, p. 1, 2017.

[18] E. Vianna, G. Comarela, and T. Pontes, "Modeling the performance of the Hadoop online prototype," in *Proceedings of International Symposium on IEEE Computer Architecture and High Performance Computing*, pp. 152–159, 2011.

[19] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Information Fusion*, vol. 42, pp. 146–157, 2018.

[20] P. Chawla, I. Chana, and A. Rana, "Cloud-based automatic test data generation framework," *Journal of Computer and System Sciences*, vol. 82, no. 5, pp. 712–738, 2016.

[21] T. J. Kavulya and R. Gandhi, "An analysis of traces from a production MapReduce cluster," in *Proceedings of the International Conference on IEEE/ACM Cluster, Cloud and Grid Computing*, pp. 94–103, Melbourne, VIC, Australia, May 2010.

[22] B. Mozafari, C. Curino, and A. Jindal, "Performance and resource modeling in highly-concurrent OLTP workloads," in *Proceedings of International Conference on the ACM SIGMOD Management of Data*, pp. 301–312, New York, NY, USA, June 2013.

[23] A. D. Popescu, A. Balmin, V. Ercegovac, and A. Ailamaki, "Predict: towards predicting the runtime of large scale iterative analytics," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1678–1689, 2013.

[24] Z. Zhang, L. Cherkasova, and B. T. Loo, "Performance modeling of MapReduce jobs in heterogeneous cloud environments," in *Proceedings of the IEEE 6th International Conference on Cloud Computing*, pp. 839–846, Singapore, December 2014.

[25] C. Shen, W. Tong, K. K. R. Choo, and S. Kausar, "Performance prediction of parallel computing models to analyze cloud-based big data applications," *Cluster Computing*, vol. 1, pp. 1–16, 2017.

[26] J. C. Platt, "Sequential minimal optimization: a fast algorithm for training support vector machines," in *Advances in Kernel Methods-Support Vector Learning*, pp. 212–223, MIT Press, Cambridge, MA, USA, 1999.

[27] S. Huang, J. Huang, and Y. Liu, "The HiBench benchmark suite: characterization of the MapReduce-based data analysis," in *Proceedings of the International Conference on 26th IEEE Data Engineering Workshops (ICDE 2010)*, pp. 41–51, Long Beach, CA, USA, March 2010.

[28] G. Miao, J. Zander, and K. W. Sung, *Fundamentals of Mobile Data Networks*, Cambridge University Press, Cambridge, UK, 2016.

[29] S. S. Lavenberg and M. Reiser, "Stationary state probabilities at arrival instants for closed queueing networks with multiple types of customers," *Journal of Applied Probability*, vol. 17, no. 4, pp. 1048–1061, 1980.

[30] A. Ghazal, F. Raab, F. Raab, M. Poess, A. Crolotte, and H. Jacobsen, "BigBench: towards an industry standard benchmark for big data analytics," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 1197–1208, New York, NY, USA, June 2013.