IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 21, 2017; accepted February 10, 2017, date of publication February 23, 2017, date of current version March 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2672675

MEST: A Model-Driven Efficient Searching Approach for MapReduce Self-Tuning

ZHENDONG BEI[1,2], ZHIBIN YU[1], (Member, IEEE), QIXIAO LIU[1], CHENGZHONG XU[1], (Fellow, IEEE), SHENGZHONG FENG[1], (Member, IEEE), AND SHUANG SONG[3]

[1]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, Shenzhen 518055, China
[2]Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences, Shenzhen 518055, China
[3]The University of Texas at Austin, Austin, TX 78712 USA

Corresponding author: Z. Yu (zb.yu@siat.ac.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2016YFB1000204, in part by the Major Scientific and Technological Project of Guangdong Province under Grant 2014B010115003, in part by the Shenzhen Technology Research Project under Grant JSGG20160510154636747, in part by the Shenzhen Peacock Project under Grant KQCX20140521115045448, in part by the Outstanding Technical Talent Program of CAS, and in part by NSFC under Grant U1401258.

ABSTRACT Hadoop is the most popular implementation framework of the MapReduce programming model, and it has a number of performance-critical configuration parameters. However, manually setting these parameters to their optimal values not only needs in-depth knowledge on Hadoop as well as the job itself, but also requires a large amount of time and efforts. Automatic approaches have therefore been proposed. Their usage, however, is still quite limited due to the intolerably long searching time. In this paper, we introduce MapreducE Self-Tuning (MEST), a framework that accelerates the searching process for the optimal configuration of a given Hadoop application. We have devised a novel mechanism by integrating the model trees algorithm with the genetic algorithm. As such, MEST significantly reduces the searching time by removing unnecessary profiling, modeling, and searching steps, which are mandatory for existing approaches. Our experiments using five benchmarks, each with two input data sets ($DS1$ and $2 \times DS1$) show that MEST improves the searching efficiency (SE) by factors of $1.37\times$ and $2.18\times$ on average respectively over the state-of-the-art approach.

INDEX TERMS MapReduce, Hadoop, self-tuning, model trees, genetic algorithm.

I. INTRODUCTION

As the data size scales up at an alarming pace in big data era, even large-scale clusters take days to process a single data set. In this scenario, efficiently managing the massive computing nodes and processing data from applications are both essential. Hadoop, the open-source implementation framework of MapReduce, has already become the most pervasively used data analytic system. However, the broad applications of Hadoop, such as the healthcare [1], [2], transportation [3], [4] and etc., present many challenges and difficulties for users to deploy Hadoop with high performance.

Appropriately configuring the Hadoop framework not only holds the key to achieve high performance, but also avoids system failures [5]. However, an optimal configuration comes at the cost of long searching time. Since the effectiveness of parameter set depends on the characteristics of an application, such as the input data size, the selection of the parameter set has to be performed each time before the application runs. However, due to the huge search space which comprised of more than 70 performance critical multi-valued parameters and their complex relationships [6], to iteratively evaluate the parameter sets concretely is expensive and time-consuming, if at all feasible. To this end, a number of off-line automatic configuring approaches have been proposed.

Self-tuning methods, which are based on off-line trained models, have been proposed to search through the complex parameter space [6]–[10]. It has been shown that the accuracy of a model determines the correctness and efficiency of the searching process. Proposals based on analytical models [6]–[8] have relatively low accuracy because of their oversimplified assumptions in the face of complicated large scale computing nodes and tasks. In this scenario, machine learning model based approaches fit this problem better since

they can capture the characteristics of the application and the complex relationship among parameters from enormous data source [9], [10]. One example, RFHOC (Random-Forest approach to auto-tuning HadOop Configuration) [10], shows the accuracy of such models depends on the sufficiency of collected training data, which may obstruct its use in real world applications. In contrary, other approaches, such as Smart Hill-Climbing algorithm(SHC) [11], search the parameter space directly. The SHC repeatedly executes a global search process in which local search process follows Hill-Climbing algorithm [12], [13]. Analogously, Generic Algorithm (GA) based approach which can be used to iteratively explore the parameter space [14]. Such mechanism produces a Parameter Configuration Settings (PCS) at the beginning, then evaluates and evolves the parameters in it by testing the application in actual cluster. As a consequence, several parameters in the PCS take many iterations to evolve to their optimal values, e.g. 24 iterations to optimize 6 parameters.

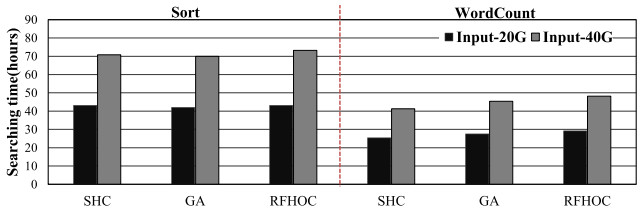


FIGURE 1. The searching time for the best optimized configuration of three typical optimization methods for two benchmarks: *Sort* and *WordCount*. Two input data sizes(20GB, 40GB) are considered per benchmark.

Figure 1 illustrates the searching cost of SHC, GA and RFHOC when benchmark *Sort* and *WordCount* run with 20 GB and 40 GB input data respectively. The detailed experimental setup is described in Section IV. Note that, machine learning model based approaches like RFHOC are the most promising techniques to be used in practice and they explore the parameter space through three serial stages including profiling, modeling, and searching. For convenience, we consider the sum of the time of the three stages as the searching time. We observe that, the searching time is long and varies significantly for different benchmarks and input data sets. For instance, these approaches spend at least 25 hours to search the corresponding optimal configuration for both *Sort* and *WordCount*. When the input data set of *Sort* increases to 40 GB, all these approaches take more than 70 hours, which severely limits the practical usage of them.

In this paper, we dig into the reason why the machine learning based approaches take so long time to optimize the configuration for a Hadoop application. As illustrated in Figure 2 (a), the profiling process usually collects *an overlarge number* of training examples at once to guarantee an accurate model. This is because the number of training examples for an accurate model can not be determined upfront and the only way to guarantee high accuracy is to collect excessive training examples, which may waste a large amount of time.

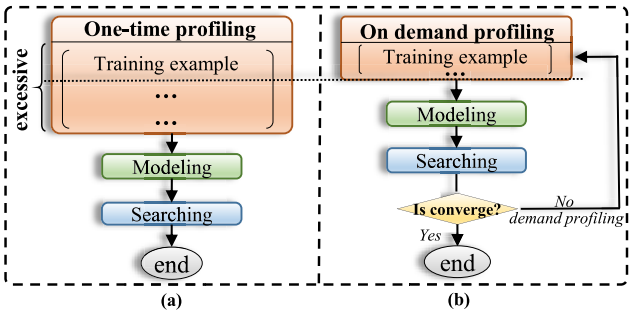


FIGURE 2. The optimization process comparison of MEST with machine learning model based methods. (a) Machine learning model based methods. (b) The main process of MEST.

To address this problem, we introduce MapreducE Self-Tuning (MEST), a framework that efficiently generates an optimal parameter set to configure a Hadoop application. Our target is not only to find the optimal parameter set, but also to find it fast. Moreover, we also aim to improve the scalability of MEST when the input data size increases. In this framework, we have devised a new mechanism based on GA which iteratively searches the parameter space to find the optimal set by enhancing its convergence. Since in each iteration, the searching is recalibrated and accelerated by a model trees (MT) based performance model. As shown in Figure 2 (b), MEST performs profiling according to the actual needs of the searching, thereby removing the unnecessary time used for profiling, modeling, and searching steps.

We evaluate MEST using five typical Hadoop applications, each with two input data sets. The results show that while MEST has halved the searching time, it still manages to derive the best optimal parameter set. Moreover, MEST exhibits the best scalability with respect to the input data size among the experimented approaches. Overall, MEST offers a reliable, accurate and fast solution to find the optimal parameter set for Hadoop applications.

In particular, we make the following contributions:

- We introduce MEST, a framework that aims to accelerate the searching process for the optimal configuration of a given Hadoop application. The key of MEST is that it performs the profiling, modeling, and searching operations on demand, which removes the unnecessary time used for excessive operations compared to existing machine learning based approaches.
- By constructing performance models with MT, we can quickly predict the execution times of Hadoop applications with different configurations, avoiding wasting time to run applications in actual clusters, which usually takes a long time.
- We have enhanced GA to automatically search the parameter space with faster convergence speed, by leveraging the MT based performance model to accelerate the testing runs.

The rest of this paper is organized as follows: Section II depicts background and related work. Section III presents the architecture design of MEST. Section IV depicts our

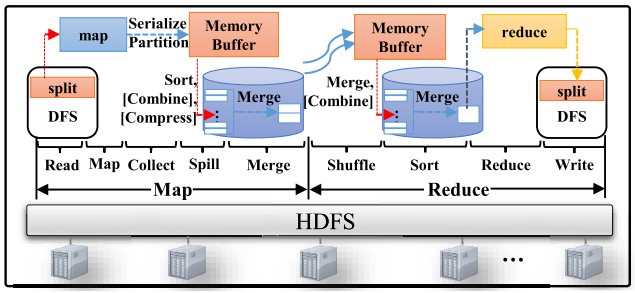


FIGURE 3. The detailed execution process of a Hadoop/MapReduce job.

experimental methodology. Section V provides results and analysis, and finally, Section VI concludes the paper.

II. BACKGROUND AND RELATED WORK

In this section, we present the background and related work of this paper.

A. Hadoop OPTIMIZATION APPROACHES

Hadoop is a well-known and pervasively used framework for the MapReduce programming model which divides an application into *Map* and *Reduce* tasks. To alleviate programming complexity, Hadoop takes over the administration of job tasks including task scheduling, resource allocating, and parameter configuring. In Figure 3, we illustrate a typical execution process of a Hadoop/MapReduce job and its interactions with the framework.

In this case, the performance of the MapReduce job depends on task scheduling and resource allocating approach at runtime. For example, a scheduler can queue tasks with specific requirements to satisfy a Service Level Objective (SLO) [15]. Researchers have also shown that, by carefully optimizing the task scheduling policy, the performance of Hadoop/MapReduce jobs can be guaranteed on complex heterogeneous systems [16] with severe interferences and different data locality [17]. Analogously, due to the large amount of resource demand, optimizing resource allocation techniques such as Support Vector Machine (SVM) [9] based approaches can guarantee the performance for a task. Moreover, task provisioning strategies can also benefit the performance of Hadoop/MapReduce jobs [5].

In addition, Hadoop provides a set of performance-critical configuration parameters for flexibility. However, how to optimize the configuration parameter values for a given application is extremely difficult. Currently, off-line configuration optimizing techniques have been commonly used, which is complementary to those run-time approaches. All these approaches however suffer from long optimal configuration searching time. We hereby focus on how to reduce the searching time in this paper.

B. CONFIGURATION OPTIMIZATION IN Hadoop

Some configuration parameters significantly impact the overall performance of a Hadoop application. For example, the parameter *MR.task.io.sort.mb* controls the memory buffer size for disk I/O operations when sorting files. On the one hand, using an over-small value of this parameter to configure a task would incur more disk I/O operations, and in turn degrade the overall performance. On the other hand, if we use an over-large value, it would reserve more unnecessary memory for a single task and hereby cause memory shortage for other tasks, still leading to overall performance degradation.

Moreover, the optimal values for these configuration parameters are usually application-specific. For example, *WordCount* requires smaller memory buffer than *TeraSort*, as *WordCount* usually produces much fewer data than *TeraSort* during the *spill* phase of its *map* stage. In addition, the optimal configuration might also affected by the input data size of a given application. For example, the larger input data size processed by a task of *TeraSort* would produced more intermediate data, which thus needs to improve the value of *MR.task.io.sort.mb* accordingly for better performance, and vice versa.

In summary, the optimal configuration parameter values of Hadoop applications are affected by a number of factors such as application characteristics and input data size. This makes manually tune them extremely difficult, if at all feasible. A number of automatic approaches have therefore been proposed.

These approaches can be generally classified into three categories: analytical, statistic reasoning, and machine learning based approaches. Herodotou et al. [6]–[8] firstly develop analytical models to predict the performance of the various phases of Hadoop/MapReduce jobs. They then recursively search for the optimal configuration randomly based on those models. However, the derived configuration is usually suboptimal because the models suffer from oversimplified assumptions and are therefore not accurate enough. Gencer et al. [18] employ a statistic reasoning technique, Response Surface (RS), to build performance models for Hadoop systems and then interactively tune their configurations to achieve optimal performance. More over, machine learning based approaches such as RFHOC [10] have also been used to build accurate performance models. However, statistic reasoning as well as machine learning based approaches usually need an over-large number of real execution experiments to construct a training set at the beginning at once, which is time-consuming. MEST tries to remove the time used for collecting the excessive training examples.

To identify optimal configuration parameter values, a number of approaches such as recursive random search (RRS) [19], smart hill climbing (SHC) [11], and genetic algorithm (GA) [14], [20] have been proposed. RRS consists of global search and local search, which operate in an iterative way. During the global search, RRS uses random sampling to identify promising areas and then invokes local search by starting recursive random sampling processes in these areas which shrink to local optima gradually. SHC improves RRS, by using weighted Latin Hypercube Sampling(wLHS) to replace the inefficient random sampling [11]. However, SHC needs many time-consuming iterations to

converge and gets stuck into local optimum easily when searching a high dimension parameter space. GA is a global optimization approach which mainly consists of selection, crossover and mutation operations. Gunther [14], a GA based method, is capable to search the best Hadoop configuration for a given application. However, GA accounts on actual experiments, which is time-consuming and impractical. Note that all the recent proposed approaches as well as our MEST are heuristic best-effort optimization techniques because the Hadoop configuration parameter space is too huge to perform exhaustive parameter search to identify optimal configurations. However, existing approaches do not consider how to achieve the heuristic best-effort optimization efficiently. In contrast, MEST is an efficient heuristic best-effort optimization. It employs the model trees algorithm to filter out the non-optimal parameter set in each iteration of the genetic algorithm, leading to accelerated optimal configuration searching process.

C. CONFIGURATION OPTIMIZATION IN OTHER SYSTEMS

Another kind of related work is about using auto-tuning techniques to optimize compilers and Java virtual machines. Dubach et al. [21] employ machine learning to automatically learn the best optimisations to apply for any new program on a new microarchitectural configuration. This work builds a model off-line which maps a microarchitecture description plus the hardware counters to the best compiler optimisation passes. Lengauer et al. [22] first propose a work on automatically tuning Java garbage collectors in a black box manner considering all available parameters. Compared to these works, our work mainly focus on tuning parameters provided by Hadoop Framework. Our target is not only to search the best configuration, but also to optimize it fast. Moreover, we also aim to improve the scalability of MEST with the input data size increases.

III. MaprEduce SELF-TUNING APPROACH

In this section, we first introduce the architecture of MEST. Subsequently, we describe the performance modelling approach. Next, we depict the optimal configuration identification technique. Finally, we described how to put them together.

A. MEST ARCHITECTURE

MEST aims to efficiently identify the optimal configuration for a given Hadoop application. It consists of three components: *data collecting*, *performance modeling*, and *optimal configuration searching*. MEST is launched when a job is invoked in Hadoop. When MEST completes execution, it outputs the optimal parameter set which is used to configure that job. Figure 4 shows the block diagram of our MEST framework. We now elaborate each component in detail:

1) *Data Collecting (DC)*. DC is used to run a MapReduce job, and profile the dynamic execution states including the execution time of that job. During which, a number of randomly generated configuration parameter values must be set in the first iteration whereas in the following ones, the values of them are updated towards the optimal configuration.
2) *Performance Modeling (PM)*. PM employs the model trees (MT) algorithm to build a performance model as a function of configuration parameters for an application based on the profiling information obtained from the DC component. In MEST, the accuracy of the model is gradually improved by using accumulated profiling data in iterations.
3) *Optimal Configuration Searching (OCS)*. OCS leverages a GA-based approach to search for the optimal configuration from a huge configuration space. Note that the performance model built by PM is used to predict the performance of a set of configuration parameter values, and in turn filter out those non-optimal ones.

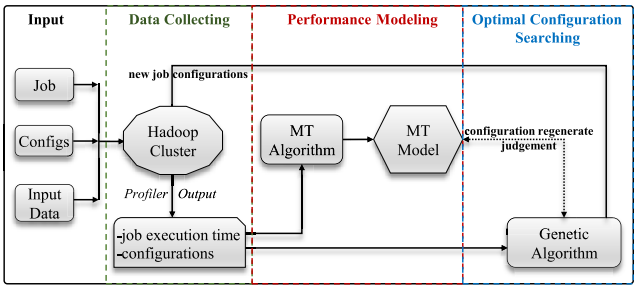


FIGURE 4. Framework diagram of MEST.

B. PERFORMANCE MODELING

The performance model stands as the core component of MEST, since its prediction determines the value of the parameters sets. However, establishing the performance model for large-scale clusters is non-trivial. Therefore, machine-learning based models are preferred to analyze these data sets. In this work, we have leveraged MT algorithm to build our performance model. The training data of the model is derived from DC, presented in the form of a matrix M, where each row contains a vector of information at each run:

$$T_k = \{ET_k, P_k^1, \ldots, P_k^i, \ldots, P_k^q\}, \quad k = 1, \ldots, n \qquad (1)$$

in which, T_k represents the information vector corresponds to the k^{th} parameter set of a job, with ET_k stands for the execution time results from this configuration. P_k^i represents the value of the i^{th} parameter in the set (these parameters are shown in the first column in Table 2); and q stands for the total number of configuration parameters, e.g. 38 in this study which we selected based on their significance on the performance; n stands for the number of vectors in matrix M which refers to the number of parameter set that are accumulated.

Next, we describe how the performance model is established from the matrix M using the MT algorithm and how the prediction of the model is improved, as shown in Figure 5. At first, we calculate the *STD* which stands for the standard deviation of data set T. And then we build a tree

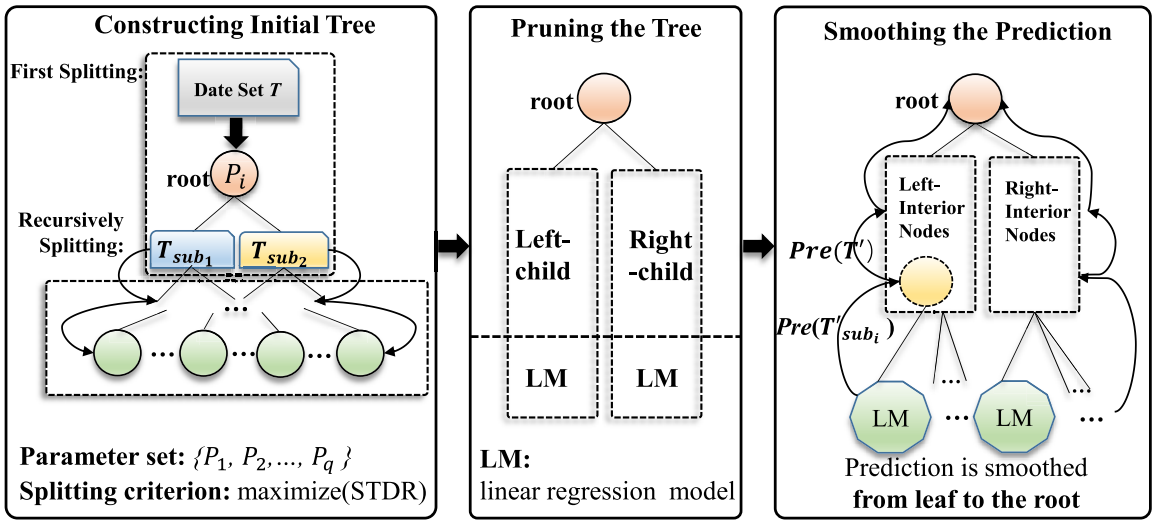


FIGURE 5. The procedure of performance modeling.

from M using a top-down splitting procedure. This procedure is similar to the formation of a binary tree, that we split the data into two subsets recursively. Subsequently, using the performance in each vector as the indicator, we need to ensure the homogeneity in both subsets. As in [23], in which authors propose to choose the splitting factor that gives the greatest expected reduction in either variance or absolute deviation. Thus, we choose from a Parameter set $Ps : \{P_1, P_2, \ldots, P_q\}$ to maximize the standard deviation reduction($STDR$), which we calculate using the following formula:

$$STDR = sd(T') - \sum_i \frac{|T'_{sub_i}|}{|T'|} \times sd(T'_{sub_i}) \tag{2}$$

where T' is the set of profiling vectors that reach the splitting tree nodes, $T'sub_i$ is the result set from the splitting process based on selected parameters, and $sd(T')$ is the standard deviation of T'. So that $STDR$ is used to guarantee the homogeneity of subsets. The splitting recursively processes until the number of profiling vectors reach to a node is smaller than 4 or the standard deviation of these vectors is smaller than $0.05 \times STD$.

Secondly, we invoke the pruning tree procedure after the tree has been built. The main purpose of pruning is to avoid over-fitting, by using standard linear regression which considers only the parameters that are incorporated in the subset to remove the redundancy. To be specific, at first we build a standard regression model for each interior node which only uses the parameters reach to the node and exclude the irrelevant parameters for each linear model one by one to ensure the minimal prediction error. Then, in each node we compute the absolute deviation dev between the actual value and the predicted one which is calculated through the linear model at the node. To make the training data for minimum error, the mean value of the dev is multiplied by the factor $(l+u)/(l-u)$ which takes into account the number (l) of profiling vectors that reach the node and the number (u) of parameters used in the model at that node. Thus, the error of the node $nodeErr$ can be calculated using the equation below:

$$nodeErr = \frac{l+u}{l-u} \times \frac{\sum_{vectors} |dev|}{l} \tag{3}$$

Next, the task is to minimize the $nodeErr$ by pruning the leaves that cause mis-predictions. To this end, we compare $nodeErr$ with $subErr$ which is the error from the subtree below that node. The $subErr$ can be calculated as follows:

$$subErr = \frac{(n_{left} \times subErr_{left} + n_{right} \times subErr_{right})}{l} \tag{4}$$

with $subErr_{left}$ and $subErr_{right}$ the $subErr$ of the left and right child of that node, respectively, n_{left} and n_{right} the numbers of profiling vectors reach to the left child and right child of that node, respectively. Note that, such algorithm is recursively performed from bottom to top as long as the $nodeErr$ decreases. At the end of this pruning process, we can get a robust MT based model. The detailed model building steps are summarized in Algorithm 1.

At last, when we used the MT based model for prediction, we invoke the smoothing prediction procedure to improve the accuracy of prediction. Partly due to the fact that vectors in M are accumulated, that inadequate data sets are used to construct the model in the first a few of iterations. Besides, the pruning tree procedure can lead to sharp discontinuities between the adjacent leaves of the pruned tree in their linear models. The smoothing procedure computes a value using a linear model on each leaf, and then passes it along the path to the root of the tree. Along the path, this value is combined with the value predicted by the linear model in that node. In this way, the discontinuities between nodes can be reduced. The combination of the predicted values can be calculated

Algorithm 1 The Procedure of Model-Trees Algorithm

Input: Data Set T
Output: model trees

```
STD = standard deviation(std) of data set T
node = rootnode(create a root node)
node.vectors = T.
SplitProcedure(node)
if (the number of node.vectors is smaller than 4 or the std
  of node.vectors is smaller than 0.05 × STD) then
    node.type = leaf
else
    node.type = Interior
    for (each P_i in Ps) do
        for (all possible split positions of the P_i) do
            calculate the STDR of P_i.
        end for
    end for
    node.parameter = P_i with maximum STDR
    SplitProcedure(node.leftchild)
    SplitProcedure(node.rightchild)
end if
end SplitProcedure
PruneProcedure(node)
if (node.type = Interior) then
    PruneProcedure(node.leftchild)
    PruneProcedure(node.rightchild)
    node.model = LinearRegressionModel(node)
    if (the subErr is more than the nodeErr) then
        node.type = leaf
    end if
end if
end PruneProcedure
```

using the equation below:

$$Pre(T') = \frac{n_i \times Pre(T'_{sub_i}) + cst \times MT(T')}{n_i + cst} \tag{5}$$

where $Pre(T')$ stands for the result on the node from smoothing which refers to one vector T'. $Pre(T'_{sub_i})$ is passed from the subset T'_{sub_i} which is the subset of T', $MT(T')$ is the value predicted by the regression model at T'. And, n_i represents the number of training sets that reach the branch node of T', cst is a smoothing factor (we set the default value as 15).

In our work, we leverage the MT algorithm to accelerate the search process since MEST operates in an iterative manner. In the sense that, unlike other machine learning techniques that need to collect a full set of training data at once, MT algorithm can be trained with accumulative data sets on demand. In the first a few of iterations, although MT builds a tree with limited data, it can point out the directions towards the optimal values. MEST therefore makes use of these hints to carefully select the parameter sets to run in subsequent iterations. The benefit is two-fold. On the one hand, MEST can reduce the time for running the MapReduce job with all parameter sets; on the other hand, MEST only runs with the potentially optimal configurations because MT has filtered out potentially non-optimal ones which would eventually take much more time. In particular, as the data sets are accumulated, the accuracy of the model improves with the increasing number of iterations. Although the complexity of the MT algorithm is also linearly increased, the cost is still negligible comparing with running the MapReduce job. As long as the prediction accuracy is saturated, MEST can use the tree produced by MT to directly predict the performance of leftover configurations.

C. OPTIMAL CONFIGURATION SEARCHING

With the MT based performance model, we need to automatically search the huge non-linear space to find the optimal configuration for a Hadoop application. Several approaches, such as random recursive search [19], pattern search [24], and genetic algorithms (GA) [20], [25] have been used to solve this issue. However, the random recursive search is sensitive in getting stuck in local optima and pattern search typically suffers from slow local (asymptotic) convergence rates. In contrast, GA is a particular class of evolutionary algorithms, that well-known for being robust against local optima [20]. It uses techniques inspired by evolutionary biology, such as inheritance, mutation, selection, and crossover [25]. GA matches our goal to search the complicated global space of configurations comprises of the vectors accumulated through many iterations with potentially many local optima. Therefore, we have adopted GA in MEST to search the optimal configuration automatically. To accelerate the searching process, we incorporate the MT algorithm between iterations of GA. We thus name our novel approach MT-driven Genetic Algorithm (MTGA).

In general, MTGA has presented an iterative processing method based on GA. At each iteration, MTGA takes the newly generated m rows as its population matrix (PM) from the matrix M. Note that, for the first iteration, we initializes the matrix M by collecting the data from m testing runs with randomly generated configuration parameter sets in real cluster. The new values of each parameter are randomly generated within its value range according to its parameter type, as shown in the last column of Table2. For a numeric parameter, a random value is generated directly. For a categorical parameter, we randomly selected one category from its listed types in an indirect way. For example, we randomly generate an integer between 1 to 4 for the parameter *MR.map.output.compress.codec* which has four compress codec types. If the integer is equal to two, we select C2. In the following iterations, the matrix M accumulates m rows each time which is produced through the MTGA procedure, as shown in Figure 6. Then, MTGA calculates the value of a threshold as follows:

$$Threshold = (Mean(\sum_{k=1}^{m} ET_k) + Min(\sum_{k=1}^{m} ET_k))/2 \tag{6}$$

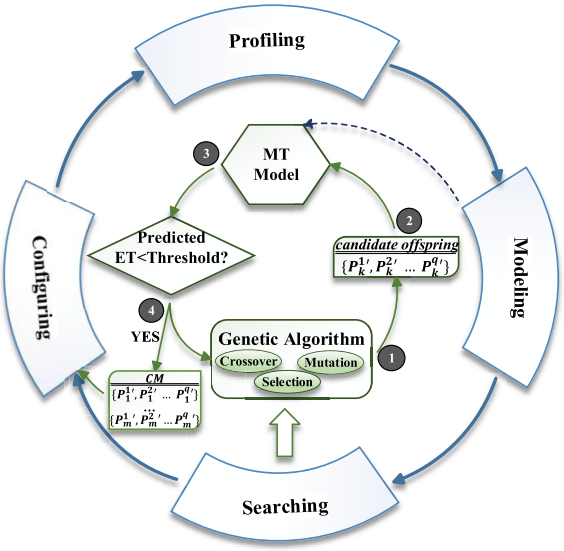


FIGURE 6. The diagram of MTGA prodedure.

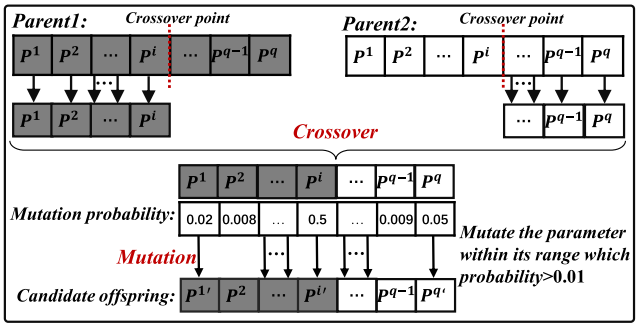


FIGURE 7. The detail of crossover and mutation operation.

with $Mean(\sum_{k=1}^{m} ET_k)$ and $Min(\sum_{k=1}^{m} ET_k)$ the mean and minimum execution time of the matrix PM, respectively. The *Threshold* is used as the criterion by MT model to select a new qualified offspring whose predicted execution time is required to be lower than this threshold.

In Figure 6, we have shown the diagram of MTGA procedure for one iteration. At first, MTGA invokes GA to perform the *selection*, *crossover*, and *mutation* operations in step ①. The *selection* operation chooses two rows from *PM* as the parents for the *crossover* operation. In which, as shown in Figure 7, the paired parameters in the parents are crossed over at a random point. Subsequently, we use a Monte Carlo method to generate a mutation probability for each parameter to decide whether the mutation is needed by comparing with a threshold(0.01). We then randomly generate a new value for the parameter within its range which probability is more than the threshold and the type of each changed parameter is also considered as before. Consequently, a candidate offspring vector with only parameter value is produced as shown in step ②. Next, this parameter vector is input to the MT-based performance model to predict its performance in step ③. The offspring can succeed if it is qualified through Equation (6). Otherwise, this procedure will be repeated to generate a new offspring. Thus, during this procedure, the new values of each parameter in an offspring are produced through selection, crossover and mutations operations of the genetic algorithm. The value of each parameter is finally selected only if the predicted execution time of the offspring is smaller than the *Threshold*. After m succeeded offsprings are obtained, a new matrix CM is formed and used as the input configurations for next iteration in MEST.

Algorithm 2 MTGA Algorithm

Input: population matrix (PM)
Output: configuration matrix (CM)

```
Take m new newly generated rows in M as PM
while (CM does not reach to m rows and CM do not evolve to optimal.) do
    Select two parent configuration sets from PM randomly based on their ET (the better ET, the higher probability to be selected)
    Crossover with a random crossover point cross over the parents to produce a candidate offspring.
    Mutation with a mutation probability(0.01) mutate candidate offspring at each parameter.
    Input the candidate offspring to the MT model.
    MT model predicts the ET of the candidate offspring.
    if (the predicted ET is smaller than threshold) then
        Put the candidate offspring into CM.
    else
        Continue the while loop.
    end if
end while
```

Note that the m offsprings in CM represent explorations in different directions of the parameter space. Since the GA method ensures the hyperplane partitions of the parameter space, we can adjust these parameter sets simultaneously and evolve them to global optimum with high probability, using heuristics that similar to approaches in [26]–[28]. With regards to the slow converge speed of GA, we have leveraged the use of MT-based performance model to accelerate the convergence and filter out those time-consuming configurations. Especially, the performance model is enhanced with the newly profiled data in every iteration of MEST. As a result, it can deliver more accurate estimates to ensure the correctness of the exploration directions. The MTGA is summarized in algorithm 2.

D. PUTTING IT ALL TOGETHER

At last, we introduce the overall procedure of MEST, as shown in Figure 6. At the beginning, given the MapReduce job and its input data set, we initialize MEST by randomly generating m configurations. These parameter sets are used by MEST to process through configuration procedure in the outer loop in the figure. With each set, MEST runs the MapReduce job and record its performance. After the execution finishes, the performance is combined with the

TABLE 1. Hadoop benchmarks and input data size considered in this study.

Benchmarks(Abbr.)	Data-1	Data-2
KMeans(KM)	40(million records)	80(million records)
PageRank(PR)	0.5(million records)	1(million records)
Sort(ST)	20(GB)	40(GB)
TeraSort(TS)	20(GB)	40(GB)
WordCount(WC)	20(GB)	40(GB)

parameter set to form the vector T using Equation (1). Next, these vectors are used to form the matrix M, which is in turn input to MT algorithm to establish the performance model. Next, the parameter space is searched by our MTGA, which will generate other m configurations. However, unlike the traditional GA method, the generated configurations need to be evaluated by the MT tree to ensure their optimality. The parameter set will be input to the MT-based performance model, to predict their performance. Then, the predicted values are compared with the *Threshold* in the current tree. If the parameter set has been validated to be more superior, it will be chosen to configure an application running on the real cluster to obtain its actual performance in the next iteration. Otherwise, the parameter set will be abandoned to avoid wasting time.

Note that, in each iteration, the matrix M contains the parameter sets that have been selected to run with their actual performance. And, the combination of the MT tree and GA algorithm will result in m configurations at the end of each iteration, which are in turn used to run the MapReduce job in next iteration. This procedure will be repeated until the obtained performance with the new configuration does not beat the existed best performance in the tree for a rule of thumb number of iterations (3 times in our experiment). In the end, the configuration that exhibits the best performance through MEST is output as the optimal parameter set.

IV. EXPERIMENTAL METHODOLOGY

A. EXPERIMENTAL SETUP

We have deployed ten Sugon servers as our experimental platform. Each node uses SUSE Linux Enterprise Server 11 and Hadoop 2.6.0. These servers are connected through Gigabit Ethernet. We designate one server to host the master node and use the other servers to host the nine slave nodes. The data block size is set to 128 MB. The other components of MEST can run on a separate node or a standalone machine as it processes the gathered information on-line. Each server is equipped with an Intel Xeon CPU E5-2407 2.20 GHz quad-core processor and 32 GB PC3 memory. Although this platform is homogeneous, MEST does not mandatorily require the cluster is homogenous. Instead, it can be directly applied on heterogeneous clusters because MEST profiles the real runtime information of a cluster and builds performance models based on the profiling. To fairly compare against Gunther, SHC, GA, and RFHOC, we re-implement them in our experimental environment.

We run the five Hadoop applications listed in Table 1, each with two corresponding data sets to evaluate MEST. These applications are taken from the HiBench [29] which represent a set of typical Hadoop workload behaviors. To be specific: *WordCount* is CPU-intensive in the map stage; *TeraSort* is CPU- and memory-intensive in the map stage and disk-intensive in the reduce stage; *Sort* is memory- and disk-intensive; Both *KMeans* and *PageRank* are iterative machine learning programs. In each iteration, *KMeans* is a CPU-intensive and *PageRank* is CPU- and disk-intensive. We have chosen 38 parameters that we have evaluated to have a significant impact on Hadoop performance, list in Table 2. These parameters specify many aspects such as data compression, shuffle behavior, CPU and memory allocation, execution behavior, and JVM. For example, we select the two important parameters including *MR.map.java.opts* and *MR.reduce.java.opts* for JVM heap size optimization of map and reduce task, respectively. We will consider other JVM parameters in our future work.

B. METRICS

To evaluate the accuracy of our performance models, we first define the prediction error of our models as follows:

$$err = \left(\sum_{i=1}^{t} \frac{|pre_i - mea_i|}{mea_i} \right) \Big/ t \tag{7}$$

with pre_i stands for the execution time predicted by our model for a given phase, mea_i represents the measured execution time of that phase, and t represents the total number of experiments.

In order to quantitatively compare how MEST performs better than Gunther, SHC, GA, and RFHOC, we consider the searching time and the performance of the optimized configuration are equally important. We therefore define a new metric, search efficiency (SE), as follows:

$$SE = 1/(OCT_{nor} \times SRT_{nor}) \tag{8}$$

where OCT_{nor} is calculated as follows:

$$OCT_{nor} = \frac{ET_{oth}}{ET_{MEST}} \tag{9}$$

with ET_{MEST} the execution time of an application optimized by MEST, and ET_{oth} the execution time of the application optimized by a certain other approach such as Gunther, SHC, GA, and RFHOC; SRT_{nor} is computed by:

$$SRT_{nor} = \frac{SRT_{oth}}{SRT_{MEST}} \tag{10}$$

with SRT_{MEST} the searching time used by MEST for an application, and SRT_{oth} the searching time used by a certain other approach such as Gunther, SHC, GA, and RFHOC. If $OCT_{nor} > 1$, it indicates the application optimized by other approaches runs more slowly than it optimized by MEST, and vice versa. Similarly, If $SRT_{nor} > 1$, it implies that the searching time used by other approaches is longer than that used by MEST, and vice versa. If $OCT_{nor} < 1$ and $SRT_{nor} > 1$ at the same time, it indicates the application optimized by a certain other approach runs faster than it optimized by MEST

TABLE 2. Description of the 38 Hadoop configuration parameters. MR-mapreduce. C1-org.apache.hadoop.io.compress. GzipCodec, C2-com.hadoop.compression.lzo.LzoCodec, C3-org.apache.hadoop.io.compress.DefaultCodec, C4-org.apache.hadoop.io.compress.SnappyCodec, BL-BLOCK, RE-RECORD.

Configuration Parameters—Description	Default	Range
MR.map.memory.mb—The upper memory limit that allows to be allocated to a map task, in MB.	1024	1024-2048
MR.map.cpu.vcores—The number of virtual CPU cores allocated for each map task.	1	1-8
MR.reduce.memory.mb—The upper memory limit that allows to be allocated to a reduce task, in MB.	1024	1024-2048
MR.reduce.cpu.vcores—The number of virtual CPU cores allocated for each reduce task of a job.	1	1-8
MR.reduce.java.opts—The heap-size for child jvms of reduce task.	200	200-700
MR.map.java.opts—Heap-size for child jvms of map task.	200	200-700
MR.task.io.sort.mb—The size of memory buffer to use while sorting intermediate data, in MB.	100	100-400
MR.map.sort.spill.percent—The soft limit in either the buffer or record collection buffers.	0.8	0.5-0.9
MR.reduce.shuffle.merge.percent—The usage threshold at which an in-memory merge will be initiated in reduce.	0.66	0.5-0.8
MR.reduce.shuffle.input.buffer.percent—The percentage of the maximum heap size for storing map outputs.	0.7	0.5-0.8
MR.reduce.merge.inmem.threshold—The threshold(the number of files)for the in-memory merge process in reduce.	1000	10-1000
MR.reduce.input.buffer.percent—The percentage of the maximum heap size to keep map outputs in reduce.	0.7	0.1-0.8
MR.task.io.sort.factor—The number of streams to merge at once while sorting files.	10	10-100
MR.reduce.shuffle.parallelcopies—The default number of parallel transfers run by reduce during the shuffle phase.	5	5-100
MR.reduce.shuffle.memory.limit.percent—The maximum percentage of the in-memory limit that shuffle can consume.	0.25	0.25-0.5
MR.map.output.compress—Whether the outputs of map task should be compressed.	false	true or false
yarn.app.MR.am.resource.mb—The amount of memory the MR AppMaster needs.	1536	1024-1536
yarn.app.MR.am.command-opts—Java opts for the MR App Master processes.	1024	819-1024
MR.tasktracker.indexcache.mb—The maximum memory that map task allows for the index cache.	10	10-50
MR.tasktracker.http.threads—The number of worker threads that for the http server.	40	5-30
MR.output.fileoutputformat.compress.codec—The codec for compression of reduce outputs.	C3	C1-C4
MR.map.output.compress.codec—The codec for compression of map outputs.	C3	C1-C4
MR.output.fileoutputformat.compress—Whether the outputs of reduce task should be compressed.	false	true or false
MR.output.fileoutputformat.compress.type—The type of Sequence Files of reduce outputs	RE	BL or RE
MR.jobtracker.handler.count—The number of server threads for the JobTracker.	10	10-50
MR.tasktracker.reduce.tasks.maximum—The maximum number of reduce tasks run simultaneously by a task tracker.	2	2-6
MR.jobtracker.heartbeats.in.second—Approximate number of heart-beats that could arrive at JobTracker.	100	100-200
MR.task.merge.progress.records—The number of records to process during merge.	10000	10000-20000
MR.job.reduce.slowstart.completedmaps—Ratio of number of maps should be finished before starting reduce.	0.05	0.01-0.2
yarn.app.MR.am.job.task.listener.thread.count—The number of threads used to handle RPC calls in the MR AppMaster.	30	30-100
MR.job.jvm.numtasks—How many tasks to run per jvm.	1	-1 or 1
MR.tasktracker.map.tasks.maximum—The maximum number of map tasks run simultaneously by a task tracker	2	1-10
MR.tasktracker.reduce.tasks.maximum—The maximum number of reduce tasks run simultaneously by a task tracker	2	1-10
MR.job.max.split.locations—The max number of block locations to store for each split.	10	10-30
MR.shuffle.connection-keep-alive.enable—Set to true to support keep-alive connections.	false	true or false
MR.map.speculative—Multiple instances of some map tasks may run in parallel if true.	true	true or false
MR.reduce.speculative—Multiple instances of some reduce tasks may run in parallel if true.	true	true or false
MR.job.reduces—The default number of reduce tasks per job.	1	9-100

whereas the searching time used by the other approach is longer than that used by MEST. In such a case, it is difficult to know which approach performs better overall. We therefore define searching efficiency *SE* as Equation (8) shows and lower is better for MEST. When the *SE* of an approach is less than 1, it indicates that MEST performs better than that approach overall, which is our goal in this study. On the contrary, if the *SE* of an approach is larger than 1, it implies that MEST performs worse than this approach and we hereby need to further improve MEST to make *SE* less than 1.

V. EVALUATION AND ANALYSIS

In this section, we evaluate the MEST framework. We start with the prediction accuracy evaluation of the MT-based performance model. Subsequently, we evaluate the *SE* of MEST and compare it with those of other approaches. Next,

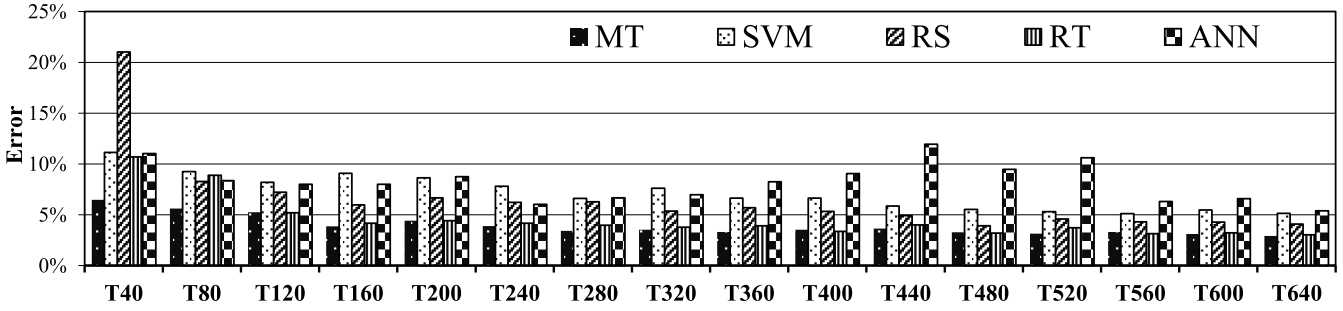


FIGURE 8. The accuracy comparison between performance models built by MT(Model trees),SVM(supported vector machine), RT(regression tree),ANN(artificial neural network),and RS(response surface).

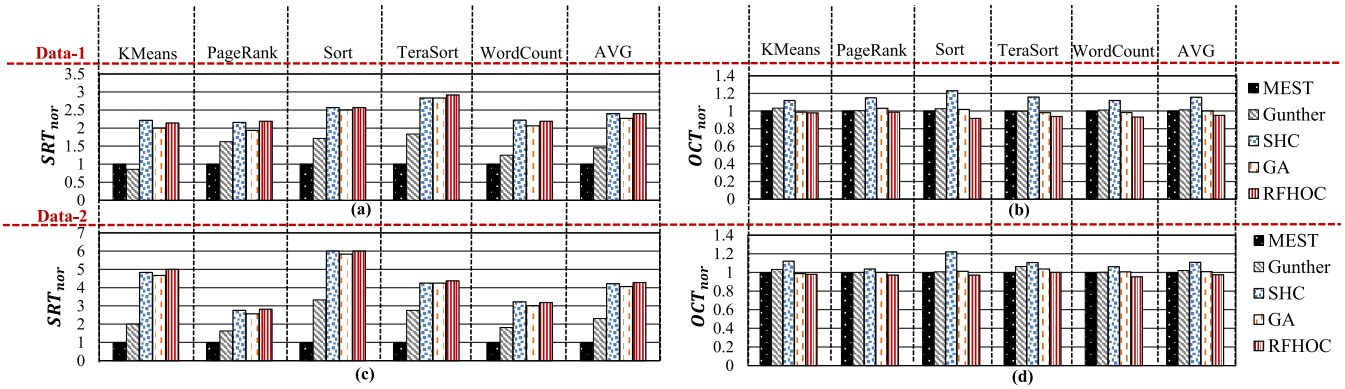


FIGURE 9. (a) and (c) show the SRT_{nor} of MEST, Gunther, SHC, GA and RFHOC(vertical axis) for the five benchmarks as illustrate in Table 1. While (b) and (d) show the OCT_{nor} of MEST, Gunther, SHC, GA and RFHOC(vertical axis) for the five benchmarks. (a) and (b) both use the DATA-1 input data. (c) and (d) both use the DATA-2.

we report the speedup achieved by MEST. At last, we use *sort* as a use case study to show and analyze the derived optimal parameter set by MEST.

A. MODEL ACCURACY

For the evaluation purpose, we split the collected profiling information into training data set and testing data set. To be specific, in the matrix M which contains the vectors of profiling information defined in Equation 1, we take the first 80% vectors as the training data set T, and the rest 20% as the testing data set E. Therefore, t in Equation (7) refers to the number of vectors in E.

The state-of-the-art machine-learning based performance modeling approaches includes Support Vector Machine (*SVM*), Response Surface (*RS*), Regression Tree (*RT*) and Artificial Neural Network (*ANN*). We have re-implemented those models in our experimental platform, and compare the prediction accuracy of our *MT* based model against them. To ensure the fairness, we have used the same M, T, and E to train and test all the models.

We have shown the evaluation of models in Figure 8 and their prediction accuracy is computed by Equation (7). Note that we label the X-axis using $(T+N)$ pattern, where N stands for the number of vectors in T. For example, $T40$ refers to 40 vectors, and we can infer that the number of the total data set is 50, where the number of vectors in E (testing set) is 10.

As can be seen, our *MT* model outperforms all other models in all cases. Secondly, we observe that all models become more accurate as they are trained by more data. Nevertheless, *MT* model needs only 120 data sets to train the model with prediction error below 5%. In contrast, *SVM*, *RS* and *RT* models need 560, 440, and 200 training examples (vectors) respectively, and *ANN* model has never reached below 5% of prediction error. Thirdly, with the least number of training examples, *MT* exhibits significantly superior prediction accuracy than others. More concretely, the error of the models built by *MT* is only 6.5% even when the number of training examples is only 40 while those of other models are all over 10%. We therefore choose 40 training examples to build the MT model in the first iteration with the minimal training cost. And, we increase the number of training examples by 40 in each iteration to improve model accuracy.

B. SEARCHING EFFICIENCY

Next, we evaluate how the searching efficiency (*SE*) of MEST advances over other state-of-the-art approaches, including Gunther [14], Smart Hill Climbing(SHC) [11], the original GA method [20] and RFHOC [10]. Figure 9 (a) and (c) show the SRT_{nor}s of these approaches for benchmark *KMeans*, *PageRank*, *Sort*, *TeraSort*, and *WordCount* listed in Table 1. Figure 9 (b) and (d) illustrate the OCT_{nor}s of these approaches. Note that (a) and (b) of Figure 9 use the same

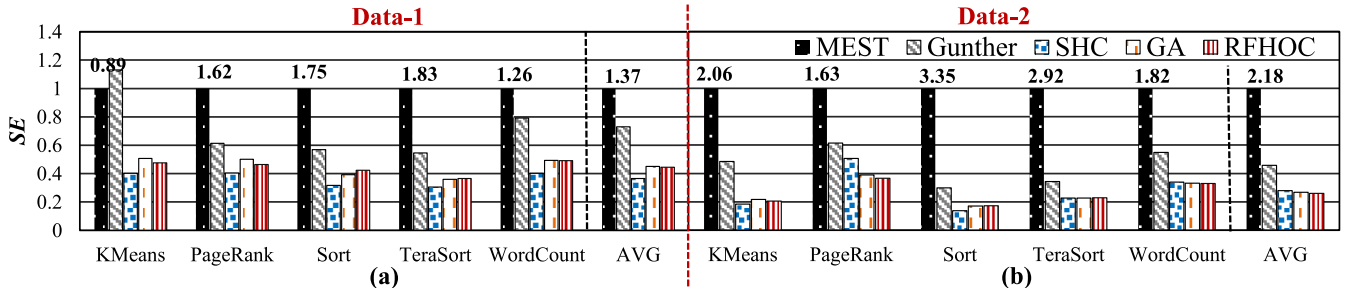


FIGURE 10. (a) and (b) show the *SE* of MEST, Gunther, SHC, GA and RFHOC(vertical axis) for the five benchmarks as illustrate in Table 1. (a) use DATA-1 input data and (b) use DATA-2. The numeric labels show the *SE* speedup of MEST over Gunther.

input data set ($Data-1$), the same applies to (c) and (d) but with another the same input data set ($Data-2$).

From Figure 9 (b) and (d), we observe that the five approaches are all capable of finding optimal configurations for the experimented applications and in turn obtaining the best performance. This indicates all these approaches are good enough for find the optimal configurations. However, it does not tell which one finds the optimal configuration most efficiently. We therefore need to observe other metrics including SRT_{nor} and SE. As shown in Figure 9 (a) and (c), MEST has the much lower value of SRT_{nor} than other approaches, indicating the searching time of MEST is much shorter than those of other approaches for the five applications, each with two corresponding input data sets. For example, the searching time of SHC is 2.4× longer than that of MEST.

We now turn to evaluate the searching efficiency (SE) defined by Equation (8), as shown in Figure 10. The SEs of other approaches are much lower than 1 except the Gunther for *KMeans*. This indicates the overall performance of the applications optimized by MEST is much higher than they optimized by other approaches. The average SEs for Gunther, SHC, GA, and RFHOC are 0.73, 0.37, 0.45, and 0.45 respectively for the five benchmarks with input data set $Data-1$. This implies that SHC performs the worst, then RFHOC and GA, and Gunther performs the most closely to MEST. When $Data-2$ is used, the average SEs change to 0.45, 0.27, 0.26, and 0.26 respectively, indicating MEST obtains more overall performance benefits as input data size increases. In other words, MEST reveals better scalability in terms of input data size, which is a nice property for big data analytics.

We now evaluate the SE improvement over Gunther made by MEST. We define the improvement as follows:

$$SE_{imp} = SE_{MEST}/SE_{Gunther} \tag{11}$$

with SE_{MEST} and $SE_{Gunther}$ the SEs of MEST and Gunther, respectively. As shown in Figure 10, MEST achieves significantly different SE_{imp}s for different benchmarks. For example, The SE_{imp} for *Sort* achieves 3.35 whereas that for *KMeans* is only 0.89. This indicates that the characteristics of benchmarks significantly affect SE_{imp}. However, the SE_{imp}s for all the experimented benchmarks except *KMeans* with $Data-1$ are significantly larger than 1, demonstrating that MEST indeed performs much better than Gunther. In summary, MEST improves SE by factors of 1.37× and 2.18× on average for input data $Data-1$ and $Data-2$ respectively over Gunther. Correspondingly, MEST reduces the searching time by factors of 1.45× and 2.3× for $Data-1$ and $Data-2$ respectively compared to Gunther.

C. CONVERGENCY SPEED ANALYSIS

As introduced in Section III-C, we use MTGA to search the huge and non-linear configuration space to find the optimal configuration for a Hadoop application. During which, we stop the searching iterations when the performance of the application does not improve anymore as we try different configurations. Subsequently, we count the number of training examples been used, which indicates the length of the time used for searching the optimal configuration for that application.

Since we have recognized Gunther as the second efficient algorithm, we compare the convergence speed of MTGA and Gunther in Figure 11. Using benchmarks listed in Table 1 with 20 GB input data, we observe that the search process of MTGA converges faster than Gunther for all benchmarks. And MTGA ensures the convergence within 320 training examples for each workload while Gunther needs up to 520 training examples to converge. Moreover, Gunther even converges prematurely to a suboptimal configuration for benchmark *KMeans*. In addition, we have observed similar results when the input data size increases to 40 GB, that no more than 320 training examples are needed by MTGA to find the optimal configuration.

D. CASE STUDY: SORT

Next, we exhibit one case study of MEST, on one particular benchmark *Sort*. Since MEST achieves the highest speedup for it, we intend to analyze the advantage of MEST based on this example. Figure 12 shows some essential parameter value variation in the optimal configurations when 40, 120, 200, 280 training examples are used to train MEST, during the search process for the benchmark *Sort*. The significantly different parameter values draw our attention. To be specific, the value of parameter *MR.map.java.opts* is 330 MB in the first iteration of MEST search process. However, the values

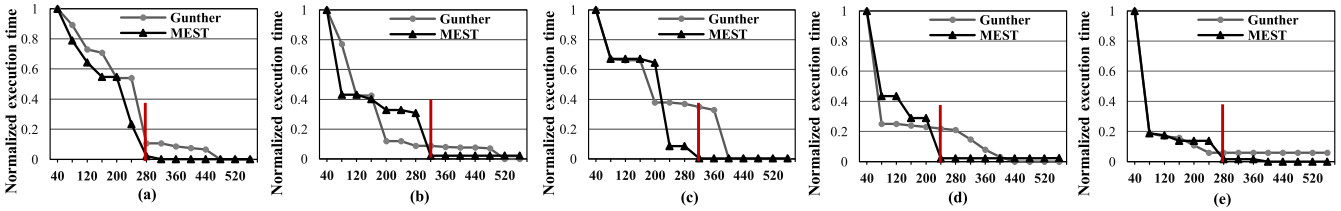


FIGURE 11. The convergency speed comparison of MEST with Gunther. The X-axis represents the number of tested configurations for each experimented benchmark. The Y-axis represents the shortest execution time among the corresponding number of tested configurations showed at X-axis. The time is normalized to a range of 0 to 1 using the Min-Max normalization technique(lower is better). (a) Sort. (b) PageRank. (c) WordCount. (d) TeraSort. (e) KMeans.

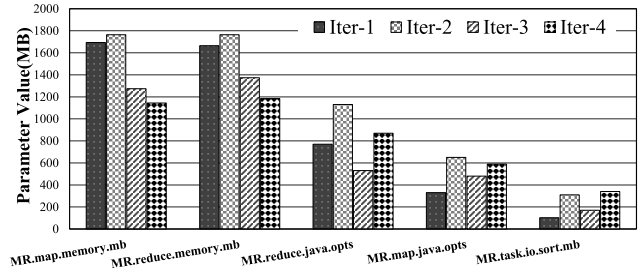


FIGURE 12. The variation of four configurations found in four search iterations respectively of MEST for *Sort*; here only show five parameters(horizontal axis) from the 38 parameters as illustrated in Table 2.

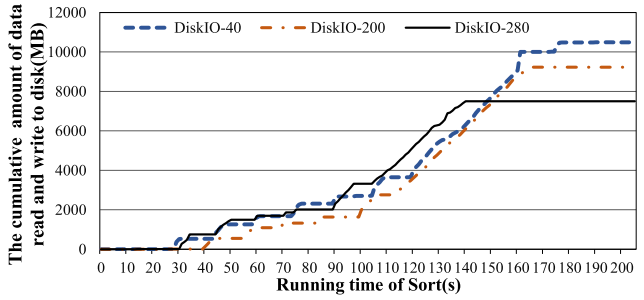


FIGURE 13. The detail of disk/IO along with execution process of *Sort* which run with three best configurations found by MEST. The DiskIO-40 means the cumulative disk/IO of *Sort* runs with the best configuration found within 40 tested configurations generated by MEST. And so on for DiskIO-200 and DiskIO-280.

of this parameter in the following three iterations are adjusted to 650 MB, 480 MB, and 590 MB respectively. We notice that the value 330 MB in the first iteration is the lowest value, while the value 650 MB in the second round is the highest. And in the iterations thereafter, the value of this parameter exhibits the behavior similar to the method of bisection, which is non-linear. Furthermore, based on domain knowledge, we know that the value of this parameter is also correlated with *MR.task.io.sort.mb* and *MR.map.memory.mb*. Nevertheless, with such complicated parameter space, MEST manages to find the best parameter set in only 280 training examples.

To investigate these parameters in detail, we have deployed the *dstat* tool [30] at each slave node to measure disk usage(amount of data in and out) every second. This tracing process runs along with the execution process of *Sort*. We first select three best configurations found among 40, 200, 280 training examples respectively during MEST searching process. We then run and trace *Sort* three times and each time with one of the three configurations. As can be seen in Figure 13, the cumulative amount of disk IO operations generated during the execution process of *Sort* for the DiskIO-280 is significantly smaller than DiskIO-40 and DiskIO-200. As we know, *Sort* is a typical data-intensive program and the case of DiskIO-280 reduces a large amount of disk IO operations. Therefore, this result further demonstrates that MEST is able to automatically to optimize memory-related configuration parameter values, and in turn to significantly alleviate the bottlenecks of data-intensive applications.

VI. CONCLUSION

In this paper, we propose MEST, an MT-driven GA approach to automatically search the optimal configuration for Hadoop applications. The integration of MT and GA can significantly reduce the searching time. Since, compared to other machine learning and statistic reasoning techniques, building performance models by MT is more flexible for data collection and model training. And MT can help GA to avoid running a program in an actual cluster with those time-consuming configurations. In particular, this approach also has good scalability in terms of input data size.

We run several typical Hadoop benchmarks with two input data sets in an actual cluster to evaluate MEST. The results show that MEST is indeed more efficient than current best approaches that have been recognized, which they have obtained immediate optimal configuration. Moreover, MEST shows a searching efficiency improvement over the recent most efficient method by 1.37× and 2.18× on average for the two experimented input data sets, respectively. Experiment results further demonstrate that MEST is more scalable to the incremental input data size than other approaches.

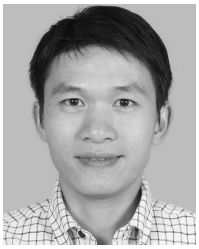
REFERENCES

[1] V. Chandola, S. R. Sukumar, and J. C. Schryver, "Knowledge discovery from massive healthcare claims data," in *Proc. 19th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2013, pp. 1312–1320.

[2] B. Lee and E. Jeong, "A design of a patient-customized healthcare system based on the hadoop with text mining (PHSHT) for an efficient disease management and prediction," *Int. J. Softw. Eng. Appl*, vol. 8, no. 8, pp. 131–150, 2014.

[3] W. Xiong, Z. Yu, L. Eeckhout, Z. Bei, F. Zhang, and C. Xu, "SZTS: A novel big data transportation system benchmark suite," in *Proc. 44th Int. Conf. Parallel Process. (ICPP)*, Sep. 2015, pp. 819–828.

[4] M. M. Rathore, A. Ahmad, A. Paul, and U. K. Thikshaja, "Exploiting real-time big data to empower smart transportation using big graphs," in *Proc. Region Symp. (TENSYMP)*, May 2016, pp. 135–139.
[5] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proc. Conf. Hot Topics Cloud Comput. (HotCloud)*, Jun. 2009. Art. no. 22.
[6] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.
[7] H. Herodotou, "Hadoop performance models," Duke Univ., Durham, NC, USA, Tech. Rep. CS-2011-05, 2011.
[8] H. Herodotou *et al.*, "Starfish: A self-tuning system for big data analytics," in *Proc. Biennial Int. Conf. Innov. Data Syst. Res. (CIDR)*, Jan. 2011, pp. 261–272.
[9] P. Lama and X. Zhou, "AROMA: Automated resource allocation and configuration of MapReduce environment in the cloud," in *Proc. 9th ACM Int. Conf. Auto. Comput. (ICAC)*, San Jose, CA, USA, Sep. 2012, pp. 63–72.
[10] Z. Bei *et al.*, "RFHOC: A random-forest approach to auto-tuning hadoop's configuration," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1470–1483, May 2016.
[11] B. Xi, Z. Liu, M. Raghavachari, C. H. Xia, and L. Zhang, "A smart hill-climbing algorithm for application server configuration," in *Proc. 13th Int. Conf. World Wide Web*, 2004, pp. 287–296.
[12] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial Intelligence: A Modern Approach*, vol. 2. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.
[13] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *J. Mach. Learn. Res.*, vol. 1, pp. 77–112, Nov. 2000.
[14] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based auto-tuning of MapReduce," in *Proc. Euro-Par Parallel Process.*, Aug. 2013, pp. 406–419.
[15] A. Verma, L. Cherkasova, and R. H. Campbell, "ARIA: Automatic resource inference and allocation for mapreduce environments," in *Proc. 8th ACM Int. Conf. Auto. Comput. (ICAC)*, Jun. 2011, pp. 235–244.
[16] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proc. USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, Dec. 2008, pp. 29–42.
[17] X. Bu, J. Rao, and C.-Z. Xu, "Interference and locality-aware task scheduling for MapReduce applications in virtual clusters," in *Proc. 22nd Int. Symp. High-Perform. Parallel Distrib. Comput.*, Jun. 2013, pp. 227–238.
[18] A. E. Gencer, D. Bindel, E. G. Sirer, and R. van Renesse, "Configuring distributed computations using response surfaces," in *Proc. 16th Annu. Middlew. Conf.*, 2015, pp. 235–246.
[19] T. Ye and S. Kalyanaraman, "A recursive random search algorithm for large-scale network parameter configuration," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 196–205, 2003.
[20] M. Kumar, M. Husian, N. Upreti, and D. Gupta, "Genetic algorithm: Review and application," *Int. J. Inf. Technol. Knowl. Manage.*, vol. 2, no. 2, pp. 451–454, 2010.
[21] C. Dubach, T. M. Jones, E. V. Bonilla, G. Fursin, and M. F. O'Boyle, "Portable compiler optimisation across embedded programs and microarchitectures using machine learning," in *Proc. 42nd Annu. IEEE/ACM Int. Symp. Microarchitecture MICRO*, Dec. 2009, pp. 78–88.
[22] P. Lengauer and H. Mössenböck, "The taming of the shrew: Increasing performance by automatic parameter tuning for java garbage collectors," in *Proc. 5th ACM/SPEC Int. Conf. Perform. Eng.*, 2014, pp. 111–122.
[23] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Boca Raton, FL, USA: CRC Press, 1984.
[24] V. Torczon and M. W. Trosset, "From evolutionary operation to parallel direct search: Pattern search algorithms for numerical optimization," *Computing Sci. Statist.*, vol. 29, pp. 396–401, May. 1998.
[25] L. Lie, "Heuristic artificial intelligent algorithm for genetic algorithm," *Key Eng. Mater.*, vol. 439, pp. 516–521, Jun. 2010.
[26] T. Weise, *Global Optimization Algorithms-Theory and Application*. Self-Published, 2009, pp. 25–26. [Online]. Available: http://www.it-weise.de/projects/book.pdf
[27] D. Whitley, "A genetic algorithm tutorial," *Statist. Comput.*, vol. 4, no. 2, pp. 65–85, Jun. 1994.
[28] P. Bajpai and M. Kumar, "Genetic algorithm—An approach to solve global optimization problems," *Indian J. Comput. Sci. Eng.*, vol. 1, no. 3, pp. 199–206, 2010.
[29] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," in *Proc. IEEE 26th Int. Conf. Data Eng. Workshops*, Mar. 2010, pp. 41–51.
[30] D. Wieers. *DSTAT: Versatile Resource Statistics Tool*, accessed on Dec. 2013. [Online]. Available: http://dag.wiee.rs/home-made/dstat/



ZHENDONG BEI received the B.S. degree from the National University of Defense Technology in 2006, and the M.S. degree from Central South University in 2009. He is currently pursuing the Ph.D. degree with the Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences. His research interests include performance optimization of big data system, data mining, machine learning, and image processing.

ZHIBIN YU (M'11) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST) in 2008. He visited the Laboratory of Computer Architecture of ECE, The University of Texas at Austin, for one year, and he was with Ghent University as a Post-Doctoral Researcher for half a year. He is currently a Professor with the Shenzhen Institutes of Advanced Technology. His research interests are micro-architecture simulation, computer architecture, workload characterization and generation, performance evaluation, multi-core architecture, GPGPU architecture, virtualization technologies, big data processing, and so forth. He is a member of ACM. He received the outstanding technical talent program of Chinese Academy of Science in 2014 and the Peacock Talent Program of Shenzhen City in 2013. He also received the first award in teaching contest of HUST young lectures in 2005 and the second award in teaching quality assessment of HUST in 2003. He serves for ISCA 2013, MICRO 2014, ISCA2015, and HPCA 2015.

QIXIAO LIU received the Ph.D. degree in computer architecture from the Universitat PolitÃlcnica de Catalunya, Spain, in 2016. He had a half-year internship with the IBM T. J. Watson Research Center in New York, USA, in 2015. He is currently an Assistant Researcher with the Shenzhen Institute of Advance Technology, Chinese Academy of Science. He is also with the Barcelona Supercomputing Center as a Junior Researcher. His research interests includes the multicore architecture, resource allocation, and cloud computing.

CHENGZHONG XU (S'01–F'16) received the Ph.D. degree from The University of Hong Kong in 1993. He is currently a Tenured Professor of Wayne State University and the Director of the Institute of Advanced Computing and Data Engineering, Shenzhen Institute of Advanced Technology, Chinese Academy of Sciences. He has published over 200 papers in journals and conferences. His research interest is in parallel and distributed systems and cloud computing. He is the Chair of the IEEE Technical Committee on Distributed Processing. He was the Best Paper Nominee of the 2013 IEEE High Performance Computer Architecture, and the Best Paper Nominee of 2013 ACM High Performance Distributed Computing. He was a recipient of the Faculty Research Award, the Career Development Chair Award, and the President's Award for Excellence in Teaching of WSU. He was also a recipient of the Outstanding Oversea Scholar Award of NSFC. He serves on a number of journal editorial boards, including the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, the *Journal of Parallel and Distributed Computing*, and *China Science Information Sciences*.

SHENGZHONG FENG received the B.Sc. degree in computer science and engineering from the University of Science and Technology of China in 1991, and the Ph.D. degree in computer science and engineering from the Beijing Institute of Technology in 1997. He is currently a Professor with the Shenzhen Institutes of Advanced Technology (SIAT) and the Assistant Director of SIAT. His main research interests include big data, cloud computing, and networked computing systems with an emphasis on system architecture design and resource management for system's performance, reliability, availability, power efficiency, and security.

SHUANG SONG received the B.S. degree in electrical and computer engineering from The University of Texas at Austin, Austin, TX, USA, in 2014, where he is currently pursuing the Ph.D. degree. His current research interests include graph processing software/hardware codesign, performance modeling of high–performance computing systems, and benchmark snippet generation for mobile computing.