Hindawi
Mathematical Problems in Engineering
Volume 2017, Article ID 2198262, 11 pages
https://doi.org/10.1155/2017/2198262

Research Article

Optimizing Hadoop Performance for Big Data Analytics in Smart Grid

Mukhtaj Khan,[1] Zhengwen Huang,[2] Maozhen Li,[2] Gareth A. Taylor,[2] Phillip M. Ashton,[3] and Mushtaq Khan[4]

[1]*Department of Computer Science, Abdul Wali Khan University Mardan, Khyber Pakhtunkhwa, Pakistan*
[2]*Department of Electronic and Computer Engineering, Brunel University London, Uxbridge UB8 3PH, UK*
[3]*National Grid, System Operation, Wokingham, UK*
[4]*Department of Computer Science, COMSATS Institute of Information Technology, Wah Cantt, Pakistan*

Correspondence should be addressed to Mukhtaj Khan; mukhtaj.khan@awkum.edu.pk

Received 30 June 2017; Accepted 17 October 2017; Published 19 November 2017

Academic Editor: Panos Liatsis

The rapid deployment of Phasor Measurement Units (PMUs) in power systems globally is leading to Big Data challenges. New high performance computing techniques are now required to process an ever increasing volume of data from PMUs. To that extent the Hadoop framework, an open source implementation of the MapReduce computing model, is gaining momentum for Big Data analytics in smart grid applications. However, Hadoop has over 190 configuration parameters, which can have a significant impact on the performance of the Hadoop framework. This paper presents an Enhanced Parallel Detrended Fluctuation Analysis (EPDFA) algorithm for scalable analytics on massive volumes of PMU data. The novel EPDFA algorithm builds on an enhanced Hadoop platform whose configuration parameters are optimized by Gene Expression Programming. Experimental results show that the EPDFA is 29 times faster than the sequential DFA in processing PMU data and 1.87 times faster than a parallel DFA, which utilizes the default Hadoop configuration settings.

1. Introduction

Phasor Measurement Units (PMU) are being rapidly deployed throughout global electricity networks, facilitating the development and deployment of Wide Area Monitoring Systems (WAMS). WAMS provide a far more immediate and accurate view of the power grid than traditional Supervisory Control and Data Acquisition (SCADA) monitoring [1, 2], collecting real-time synchronized measurements at a typical rate of 1 sample per cycle of the system frequency. This brings in new challenges in terms of data management that need to be addressed to fully realize the benefits of the technology.

The devices transmit 4.32 million measurements per parameter per day for a 50 Hz system. This is orders of magnitude larger than traditional monitoring solutions and requires fast-acting, scalable algorithms, combined with novel visualization techniques to turn the growing datasets into actionable information for network operators and planners alike.

The authors' previous research has focused on the detection of transient events in PMU datasets using Detrended Fluctuation Analysis (DFA), for the purpose of triggering steady-state estimators [3] and on determining events suitable for system inertia estimation [4]. However, processing an ever increasing volume of PMU data in a timely manner necessitates a high performance and scalable computing infrastructure. For this purpose we have parallelized the works presented in [3] using the MapReduce computing model [5] and implemented a parallel DFA (PDFA) [6] using the Hadoop MapReduce framework [7].

The MapReduce model has become a de facto standard for Big Data analytics by capitalizing on clusters of inexpensive commodity computers. The Hadoop framework is an open source implementation of the MapReduce model and has been widely adopted due to its remarkable features such as high scalability, fault-tolerance, and computational parallelization [8, 9]. In addition, the Hadoop framework has

also been applied in the power system domain for power grid data analysis [6, 10–13].

Despite its remarkable features, Hadoop is a complex framework, which has a number of components that interacts with each other across a cluster of nodes. The execution times of Hadoop jobs are sensitive to each component of the framework including the underlying hardware, network infrastructure, and configuration parameters. It is worth noting that the Hadoop framework has more than 190 tunable configuration parameters, some of which have a significant impact on the execution of a Hadoop job [14]. Manually tuning these parameters is a time consuming and ineffective process and is highly challenging when attempting to ensure that Hadoop operates at an optimal level of performance. In addition, the Hadoop framework has a black-box-like feature, which makes it extremely difficult to find a mathematical model or an objective function that represents a correlation among the parameters. The large parameter space together with the complex correlation among the configuration parameters further increases the complexity of a manual tuning process. Therefore, an effective and automatic approach to tuning Hadoop's parameters has become a necessity.

In this paper, we present an Enhanced Parallel Detrended Fluctuation Analysis (EPDFA) algorithm for scalable analytics on massive PMU datasets. EPDFA is based on an enhanced Hadoop platform whose configuration parameters are optimized by Gene Expression Programming (GEP) [15]. The EPDFA employs GEP to construct an objective function based on a historical profile of the execution of jobs. The objective function represents a mathematical correlation among the core Hadoop parameters. It then makes use of the constructed objective function to find a set of optimal values of the core Hadoop parameters for performance enhancement. It should be noted that, in the proposed optimization process, the entire parameter search space is considered in order to maintain the interdependencies among the configuration parameters. The performance of the EPDFA is evaluated on an experimental Hadoop cluster configured with 8 Virtual Machines (VMs) and is compared with both the original sequential DFA and the PDFA that only utilizes the default Hadoop configuration settings. The PMU data used in the evaluation was collected from the WAMS of the Great Britain (GB) transmission system.

The reminder of the paper is organized as follows. Section 2 reviews the related work on Hadoop configuration tuning. Section 3 introduces a set of core Hadoop configuration parameters, which are considered in this work. Section 4 presents in detail the design and implementation of the EPDFA for scalable analysis on PMU data. Section 5 compares the performance of the EPDFA with that of the sequential DFA and the PDFA, respectively, using an experimental Hadoop cluster. Section 6 concludes the paper and proposes some further work.

2. Hadoop Parameter Tuning

In this section the related work on autotuning Hadoop configuration parameter settings is reviewed.

In the relevant literature, there are several Hadoop performance models that focus on tuning Hadoop configuration parameters in order to enhance the execution of Hadoop jobs [14, 16–20]. Wu and Gokhale proposed Profiling and Performance Analysis-Based System (PPABS) [17], which automatically tunes the Hadoop configuration parameter settings based on executed job profiles. The PPABS framework consists of Analyzer and Recognizer components. The Analyzer trains the PPABS to classify the jobs having similar execution times into a set of equivalent classes. The Analyzer uses *K-means++* to classify the jobs and simulated annealing to find optimal settings. The Recognizer classifies a new job into one of these equivalent classes using a pattern recognition technique. The Recognizer first runs the new job on a small dataset using default configuration settings and then applies the pattern recognition technique to classify it. Each class has the best configuration parameter settings. Once the Recognizer determines the class of a new job, it then automatically uploads the best configuration settings for this job. However, PPABS is unable to determine the fine-tuned configuration settings for a new job that does not belong to any of these equivalent classes. Herodotou et al. proposed Starfish [14, 16] that employs a mixture of cost model [21] and simulator to optimize a Hadoop job based on previously executed job profile information. However, the Starfish model is based on simplifying assumptions [20], which indicate that the obtained configuration may be suboptimal. Liao et al. [18] proposed a search based model that automatically tunes the configuration parameters using a Genetic Algorithm (GA). One critical limitation is that it does not have a fitness function implemented in the GA. The fitness of a set of parameter values is evaluated by physically executing a Hadoop job using the tuned parameters, which is an exhaustive and time consuming process. Liu et al. [19] proposed two approaches to optimize Hadoop applications. The first approach optimizes the compiler at run time and a new Application Programming Interface (API) was developed on top of a Java Bytecode Optimization Framework [22] to reduce the overhead of iterative Hadoop applications. The second approach optimizes a Hadoop application by tuning Hadoop configuration parameters. This approach divides the parameters search space into subsearch spaces and then searches for optimum values by trying different values for parameters iteratively within the range. However, both approaches are unable to provide a sophisticated search technique and a mathematical function that represents the correlation of the Hadoop configuration parameters. Li et al. [23] proposed a performance evaluation model for the whole system optimization of Hadoop. The model analyzes the hardware and software levels and explores the performance issues in both levels. The model mainly focuses on the impact of different configuration settings on job execution time instead of tuning the configuration parameters. Yu et al. [20] proposed a performance model, which employs a combination of Random-Forest and GA techniques. The Random-Forest approach is used to build performance models for the *map* phase and the *reduce* phase and a GA is employed to search optimum configuration parameter settings within the parameter space. It should be noted that a Hadoop job

Table 1: Hadoop core parameters as GEP variables.

GEP variables	Hadoop parameters	Default values	Data types
$x0$	*io.sort.factor*	*10*	*Integer*
$x1$	*io.sort.mb*	*100*	*Integer*
$x2$	*io.sort.spill.percent*	*0.80*	*Float*
$x3$	*mapred.reduce.tasks*	*1*	*Integer*
$x4$	*mapreduce.tasktracker.map.tasks.maximum*	*2*	*Integer*
$x5$	*mapreduce.tasktracker.reduce.tasks.maximum*	*2*	*Integer*
$x6$	*mapred.child.java.opts*	*200*	*Integer*
$x7$	*mapreduce.reduce.shuffle.input.buffer.percent*	*0.70*	*Float*
$x8$	*mapred.inmem.merge.threshold*	*1000*	*Integer*
$x9$	*Input data size (number of samples/MB)*	*User*	*Integer*

is executed in overlapping and nonoverlapping stages [24], which are ignored in the proposed performance model. As a result, the performance estimation of the proposed model may be inaccurate. Furthermore, the proposed model uses a dynamic instrumentation tool (BTrace) to collect the timing characteristic of tasks. BTrace utilizes extra CPU cycles that generate extra overheads, especially for CPU-intensive applications. As a result, the proposed model overestimates the execution time of a job.

3. Hadoop Parameters

The Hadoop framework has more than 190 tunable configuration parameters that allow users to manage the flow of a Hadoop job in different phases during the execution process. Some of them are core parameters and have a significant impact on the performance of a Hadoop job [14, 18, 25]. Consider that all of the 190 configuration parameters for optimization purposes would be unrealistic and time consuming. In order to reduce the parameter, search space, and effectively speed up the search process, we consider only core parameters in this research. The selection of the core parameters is based on previous research studies [14, 17, 18, 25, 26]. The core parameters as listed in Table 1 in brief are as follows.

io.sort.factor. This parameter determines the number of files (streams) to be merged during the sorting process of map tasks. The default value is 10, but increasing its value improves the utilization of the physical memory and reduces the overhead in IO operations.

io.sort.mb. During job execution, the output of a map task is not directly written into the hard disk but is written into an in-memory buffer which is assigned to each map task. The size of the in-memory buffer is specified through the *io.sort.mb* parameter. The default value of this parameter is 100 MB. The recommended value for this parameter is between 30% and 40% of the *Java_Opts* value and should be larger than the output size of a map task which minimizes the number of spill records [27].

io.sort.spill.percent. The default value of this parameter is 0.8 (80%). When an in-memory buffer is filled up to 80%, the data of the in-memory buffer *(io.sort.mb)* should be spilled into the hard disk. It is recommended that the value of *io.sort.spill.percent* should not be less than 0.50.

mapred.reduce.tasks. This parameter can have a significant impact on the performance of a Hadoop job [24]. The default value is 1. The optimum value of this parameter is mainly dependent on the size of an input dataset and the number of reduce slots configured in a Hadoop cluster. Setting a small number of reduce tasks for a job decreases the overhead in setting up tasks on a small input dataset while setting a large number of reduce tasks improves the hard disk IO utilization on a large input dataset. The recommended number of reduce tasks is 90% of the total number of reduce slots configured in a cluster [28].

mapreduce.tasktracker.map.tasks.maximum, mapreduce.tasktracker.reduce.tasks.maximum. These parameters define the number of the *map* and *reduce* tasks that can be executed simultaneously on each cluster node. Increasing the values of these parameters increases the utilization of CPUs and physical memory of the cluster node which can improve the performance of a Hadoop job.

mapred.child.java.opts. This is a memory related parameter and the main candidate for Java Virtual Machine (JVM) tuning. The default value is *-Xmx200m* which gives at most 200 MB physical memory to each child task. Increasing the value of *Java_Opt* reduces spill operations to output map results into the hard disk which can improve the performance of a job.

mapred.inmem.merge.threshold. The threshold value indicates the number of files for the in-memory merge process. When the number of *map* output files equal to threshold value is accumulated then the system initiates the process of merging the *map* output files and spill to a disk. A value of zero for this parameter means there is no threshold and the spill process is controlled by the *mapred.reduce.shuffle.merge.percent* parameter [27].

4. The Optimization of Hadoop Using GEP

The automated Hadoop performance tuning approach is based on a GEP technique, which automatically searches

Table 2: Mathematic functions used in GEP.

Functions	Function descriptions	Input data types
plus	$f(a,b) = a + b$	*Integer* or *float*
minus	$f(a,b) = a - b$	*Integer* or *float*
multiply	$f(a,b) = a * b$	*Integer* or *float*
divide	$f(a,b) = a/b$	*Integer* or *float*
sin	$f(a) = \sin(a)$	*Integer* or *float*
cos	$f(a) = \cos(a)$	*Integer* or *float*
tan	$f(a) = \tan(a)$	*Integer* or *float*
acos	$f(a) = \text{acos}(a)$	*Integer* or *float*
asin	$f(a) = \text{asin}(a)$	*Integer* or *float*
atan	$f(a) = \text{atan}(a)$	*Integer* or *float*
exp	$f(a)$ returns the exponential e^a	*Integer* or *float*
log	$f(a) = \log(a)$	*Positive integer* or *float*
$\log_{10}$	$f(a)$ returns the (base-10) logarithm of a	*Positive integer* or *float*
pow	$f(a,b)$ returns base a raised to the power exponent b	*Integer* or *float*
sqrt	$f(a) = \text{sqrt}(x)$	*Positive integer* or *float*
fmod	$f(a,b)$ returns the floating-point remainder of a/b (rounded towards zero)	*Integer* or *float*
pow10	$f(a)$ returns base 10 raised to the power exponent a	*Integer* or *float*
inv	$f(a) = 1/a$	*Integer* or *float*
abs	$f(a)$ returns absolute value of parameter a	*integer*
neg	$f(a) = -a$;	*Integer* or *float*

for Hadoop optimum configuration parameter settings by building a mathematical correlation among the configuration parameters. In this section we first describe the GEP technique and then present the implementation of the EPDFA algorithm with the Hadoop performance enhancement.

GEP [15] is a new type of Evolutionary Algorithm (EA) [29]. It is developed based on concepts that are similar to Genetic Algorithms (GA) [30] and Genetic Programming (GP) [31]. Using a special representational format of the solution structure, GEP overcomes some limitations of both GA and GP. GEP uses a combined chromosome and expression tree structure [15] to represent a targeted solution of the problem being investigated. The factors of the targeted solution are encoded into a linear chromosome format together with some potential functions, which can be used to describe the correlation of the factors. Each chromosome generates an expression tree, and the chromosomes containing these factors are evolved during the evolutionary process.

4.1. GEP Design. The execution time of a Hadoop job can be expressed using (1) where $x_0, x_1, \ldots, x_n$ represent the Hadoop configuration parameters.

$$\text{Execution Time} = f\left(x_0, x_1, \ldots, x_n\right). \quad (1)$$

In this research, we consider 9 core Hadoop parameters and based on the data types of these Hadoop configuration parameters, the functions shown in Table 2 can be applied in the GEP method. A correlation of the Hadoop parameters can be represented by a combination of the functions. Figure 1 shows an example of mining a correlation of 2 parameters

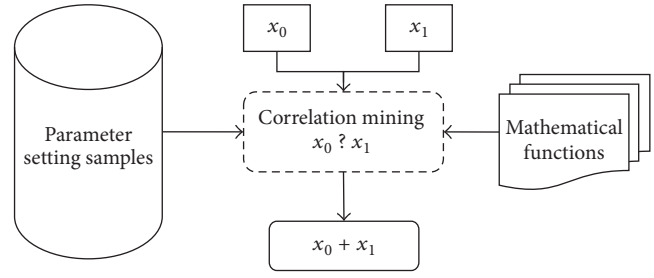

Figure 1: An example of parameter correlation mining.

(x_0 and x_1) which is conducted in the following steps in the proposed GEP method:

(i) Based on the data types of x_0 and x_1, find a function, which has the same input data type as either x_0 or x_1 and has 2 input parameters.

(ii) Calculate the estimated execution time of the selected function using the parameter setting samples.

(iii) Find the best function between x_0 and x_1, which produces the closest estimate to the actual execution time. In this case, the Plus function is selected.

Similarly, a correlation of $x_0, x_1, \ldots, x_n$ can be mined using the GEP method. The chromosome and expression tree structure of GEP is used to hold the parameters and functions. A combination of functions, which takes $x_0, x_1, \ldots, x_n$, as inputs is encoded into a linear chromosome that is maintained and developed during the evolution process. Meanwhile, the expression tree generated from the linear

```
Input: A set of Hadoop job running samples;
Output: A correlation of the Hadoop parameters;
(1)  FOR x = 1 TO size of population DO
(2)    create chromosome (x) with the combination of mathematic function and parameter;
(3)    fitness value (x) = 0;
(4)    x++;
(5)  ENDFOR
(6)  best chromosome = chromosome (1);
(7)  best fitness value = 0;
(8)  WHILE i < termination generation number DO
(9)    FOR x = 1 TO size of population DO
(10)     Translate chromosome (x) into expression tree (x);
(11)     FOR y = 1 TO the number of training samples DO
(12)          evaluate the estimated execution time for case (y)
(13)          IF ABS (timeDiff) < bias window THEN
(14)             fitness value (x)++;
(15)          ENDIF
(16)          y++;
(17)     ENDFOR
(18)     IF fitness value (x) = the number of training samples THEN
(19)          best chromosome = Chromosome (x) GO TO(29);
(20)     ELSE IF fitness value (x) > best fitness value THEN
(21)          best chromosome = Chromosome (x);
(22)          best fitness value = fitness value (x);
(23)     ENDIF
(24)     Apply replication, selection and genetic modification on chromosome (x) proportionally;
(25)     Use the modified chromosome (x) to overwrite the original one;
(26)     x++;
(27)   ENDFOR
(28)   i++;
(29) ENDWHILE
(30) Return best chromosome
```

Algorithm 1: GEP implementation.

chromosome produces a form of $f(x_0, x_1, \ldots, x_n)$ based on which an estimated execution time is computed and compared with the actual execution time. A final form of $f(x_0, x_1, \ldots, x_n)$ will be produced at the end of the evolution process whose estimated execution time is the closest to the actual execution time.

In the GEP method, a chromosome can consist of one or more genes. For computational simplicity, each chromosome has only one gene in the proposed method. A gene is composed of a head and a tail. The elements of the head are selected randomly from the set of Hadoop parameters (listed in Table 1) and the set of functions (listed in Table 2). However, the elements of the tail are selected only from the Hadoop parameter set. The length of a gene head is set to 20, which covers all the possible combinations of the functions. The length of a gene tail can be computed using

$$\text{Length}\left(\text{Gene}_{\text{Tail}}\right) = \text{Length}\left(\text{Gene}_{\text{Head}}\right) \times (n-1) + 1, \quad (2)$$

where n is the largest number of input arguments of a function. In the following section, we present how the GEP method evolves when mining a correlation from the Hadoop configuration parameters in order to construct an objective function.

4.2. Mining Hadoop Parameter Correlation with GEP. Algorithm 1 shows the implementation of the GEP method in order to construct an objective function that represents the correlation between the Hadoop configuration parameters and estimates the execution time of a job. The input of Algorithm 1 is a set of Hadoop job execution samples, which are used as a training dataset.

In Algorithm 1, Lines (1) to (5) initialize the first generation of 500 chromosomes, which represent 500 possible correlations between the Hadoop parameters. Lines (8) to (29) implement an evolution process in which a single loop represents a generation of the evolution process. Each chromosome is translated into an expression tree. Lines (11) to (17) calculate the fitness value of a chromosome. For each training sample, GEP produces an estimated execution time of a Hadoop job and compares with the actual execution time of the job. If the difference is less than a predefined bias window, the fitness value of the current chromosome will be increased by 1.

The size of the bias window is set to 50 seconds, which allows a maximum of 10% of the error space taking into account the actual execution time of a Hadoop job sample. Line (18) states that the evolution process terminates in an ideal case when the fitness value is equal to the number of

training samples. Otherwise, the evolution process continues and the chromosome with the best fitness value will be retained as shown in Lines (20) to (23). At the end of each generation as shown in Lines (24) to (25), a genetic modification is applied to the current generation to create variations of the chromosomes for the next generation.

We varied the number of generations from 20000 to 80000 in the GEP evolution process and found that the quality of a chromosome (the ratio of the fitness value to the number of training samples) was finally higher than 90%. As a result, we set 80000 as the number of generations. The genetic modification parameters were set using the classic values [15] as shown in Table 3.

After 80000 generations, GEP generates an objective function as described in (3) representing a correlation of the Hadoop parameters listed in Table 1.

$$
\begin{aligned}
f(x_0, x_1, \ldots, x_9) = {} & \left((x_7 * x_9) - \left(\frac{((((x_2 - x_8) + (x_1 - x_4)) / ((x_4 * x_4 * x_4) / x_7)) - x_6)}{x_4} \right) \right) \\
& + \left(\frac{x_9}{(x_4 - \operatorname{atan}(\exp(x_1)))} \right) + x_9 \\
& - \tan\left(((\operatorname{pow}((x_9 - x_7), (x_3 - x_0))) - ((x_4 * x_4 * x_4) - (x_4 - x_9))) * \left(\cos\left(\frac{(x_7 / x_0)}{(x_9 - x_7)} \right) \right) \right) \\
& + (x_9 - (x_4 * x_4)) \\
& + \left(x_9 - \tan\left((\operatorname{pow}((x_1 - x_5), (x_3 - x_0)) - (\log_{10}(x_4) - (x_4 + x_9))) * \cos\left(\frac{\operatorname{fmod}(x_7, x_3)}{x_6} \right) \right) \right).
\end{aligned}
\tag{3}
$$

4.3. Optimizing Hadoop Settings with GEP. The correlation mined in the previous section describes each Hadoop parameter's contribution to the execution time. In GEP optimization, each chromosome represents a Hadoop configuration setting. Based on the objective function represented by (3), GEP finds the best chromosome that leads to the shortest execution time of a Hadoop job in each generation. GEP uses a range for each parameter that is involved in the evolution process as shown in Table 4. The range of each involved parameter is selected based on the values used in the training dataset for the corresponding parameters.

Initially default values were set for the involved parameters and the values were then updated to obtain optimal solution. Updating the configuration values for the involved parameters is dependent on a number of factors such as input data block size, available physical memory, the number of CPUs, and the type of applications. For example, we set a range of 10~100 for $X1$. The value of $X1$ is based on the data block size and its value must be greater than the input data block size in order to reduce the number of spill records. In this work the size of the data block is 5 MB. In the PDFA, the computation is mainly conducted in the *map* phase and very little work is performed in the *reduce* phase. Therefore, we set a large range of values for $X4$ (i.e., 1~8) as compared to $X5$ (i.e., 1~2).

4.4. The Implementation of EPDFA. The EPDFA algorithm proposed in this paper is optimizing the authors' previous work [3, 4, 6] where a dataset of PMU frequency measurements is detrended on a sample-by-sample sliding window. The window was configured to be 50 samples long, this is to detect for changes or fluctuations in the power systems state over a 1-second period (at 50 Hz), looking for a specific loss shape in frequency, following an instantaneous loss in generation. A root mean square (RMS) value is then taken of the fluctuation, F for every window, as shown in (4); this value is then compared with a threshold value, predetermined through a number of previous baseline studies, $F = 0.2 \times 10^{-3}$, to detect for the presence of an event.

$$
F(n) = \sqrt{\frac{1}{n} \sum_{k=1}^{n} [e(k)]^2}, \tag{4}
$$

where n is the size of the window (50 samples), k is the sample number, and $e(k)$ is the detrended signal.

Figure 2 shows the software architecture of EPDFA for an off-line analysis of PMU data in an enhanced Hadoop cluster. OpenPDC [32] was installed which collects measurements from the installed PMUs, which are then stored in the OpenPDC data historian. OpenPDC was configured in such a way that when the data historian size reaches 100 MB, a new data file is created in .d format with a corresponding timestamp.

A data agent application has been developed in the Java programming language which automatically detects the new data file and moves it to the Hadoop cluster. A portion of the PMU measurements was processed by PDFA with different configuration settings in order to create a historical jobs profile (training datasets). Once the historical job profile was created EPDFA invoked the GEP optimizer. The GEP optimizer has been implemented as a two-stage process. In the first stage, it utilizes the jobs profile and constructs the objective function as presented in Section 4.2. In the second stage, the GEP optimizer searches for an optimal configuration setting within the parameter search space,

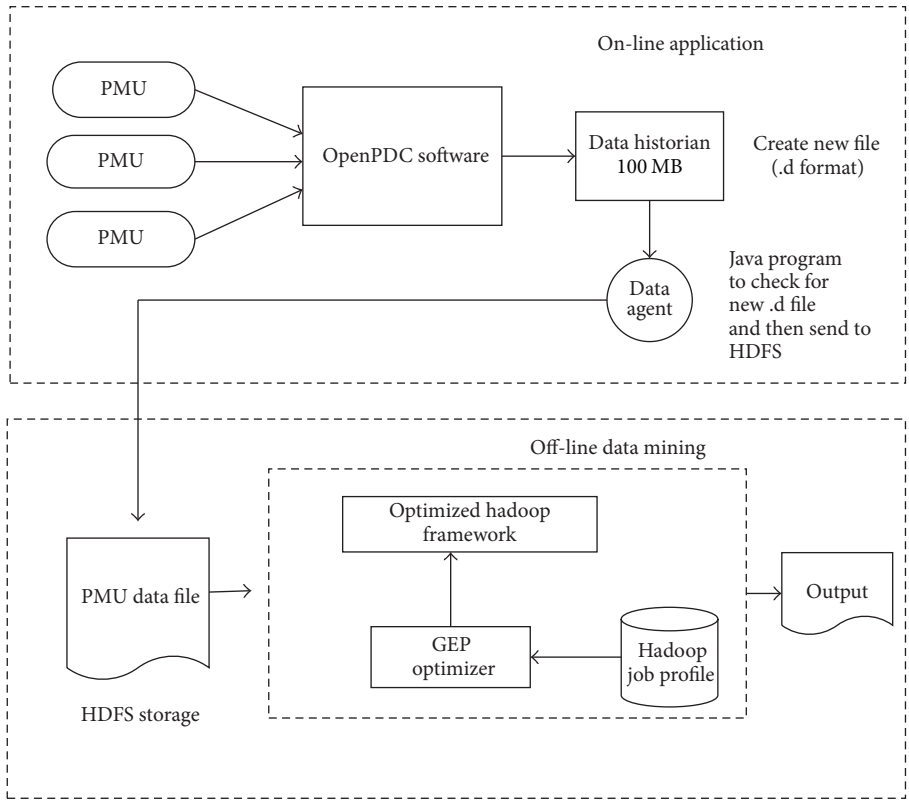

Figure 2: The software architecture of EPDFA.

Table 3: GEP parameter settings.

Genetic modification parameters of GEP	Values
One-point recombination rate	30%
Insertion sequence transposition rate	10%
Inversion rate	10%
Mutation rate	0.44%

Table 4: Range of parameters.

Variables	Ranges
$X0$	10~230
$X1$	10~100
$X2$	0.60~0.85
$X3$	1~16
$X4$	1~8
$X5$	1~2
$X6$	110~1000
$X7$	0.70~0.85
$X8$	200~1000
$X9$	Variable

which is then configured in a physical Hadoop cluster for performance enhancement.

5. Experimental Results

The performance of the EPDFA was extensively evaluated from the aspects of both computational speedup and scalability. For this purpose, an experimental Hadoop cluster was set up using an Intel Xeon server machine configured with 8 VMs. In this section, we first give a brief introduction to the experimental environment that was used in the evaluation process and then present the experimental results.

5.1. Experimental Setup. The experiments were performed on a Hadoop cluster using a high performance Intel Xeon server machine comprising 4 Intel Nehalem-EX processors running at 2.27 GHz each with 128 GB of physical memory. Each processor has 10 CPU cores with hyperthread technology enabled in each core. The specification details of the server and the software packages are listed in Table 5. Oracle Virtual Box [33] was installed and configured 8 VMs on the server machine. Each VM was assigned with 4 CPU cores, 8 GB RAM, and 150 GB hard disk storage. Hadoop-1.2.1 was installed and configured on one VM as the Name Node and the remaining 7 VMs as the Data Nodes. The Name Node was also used as a Data Node. The data block size of the Hadoop Distributed File System (HDFS) was set to 5 MB and the replication level of data blocks was set to 2. We varied different numbers of PMU data samples in the experiments.

5.2. Experimental Results. In this section a comparison of the computational efficiency of the EPDFA is presented with

Table 5: Hadoop cluster setup.

Intel Xeon Server	CPU	40 cores with hyperthread enabled
	Processor	2.27 GHz
	Hard disk	2 TB and 320 GB
	Connectivity	100 Mbps Ethernet LAN
	Memory	128 GB
Software	Operating system	Ubuntu 12.04 TLS
	JDK	Version 1.6
	Hadoop	Version 1.2.1
	Oracle Virtual Box	Version 4.2.8
	OpenPDC	Version 1.5
	Python	Version 2.7.3

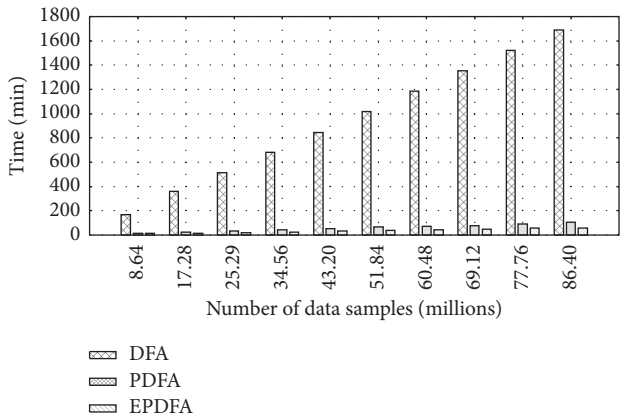

Figure 3: Execution times of the DFA, PDFA, and EPDFA.

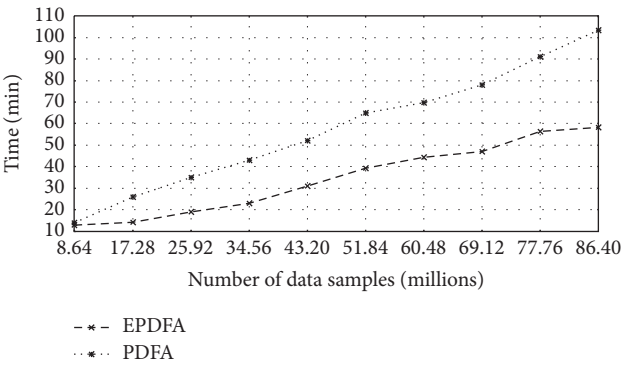

Figure 4: Computational gap between the EPDFA and PDFA.

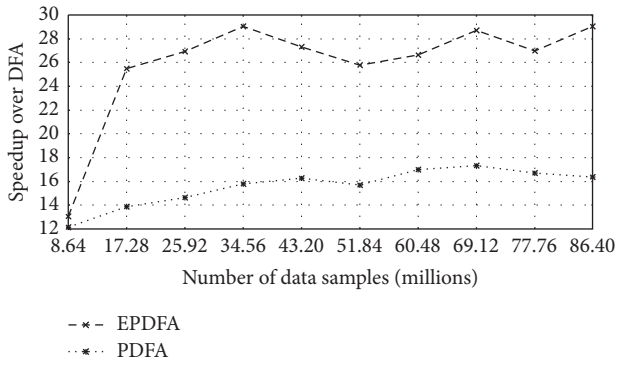

Figure 5: Speedup of both the PDFA and EPDFA over DFA.

that of authors' previous work on the DFA [3] and PDFA [6] algorithms, respectively.

A set of PMU data samples provided by National Grid, the National Electricity Transmission System Operator (NETSO) for GB, was used in the evaluation. The data comprised 6000 samples of frequency measurements at 50 Hz from a PMU, equating to 2 minutes' worth of system data. The data contained a known system event, in the loss of a generator exporting approximately 1000 MW. In order to create a massive PMU data scenario, this dataset was replicated a number of times to provide a relatively large number of PMU data samples up to 86.40 million. In order to evaluate the computational performance of the EPDFA, a number of experiments were carried out that varied the number of PMU data samples from 8.6 million to 86.4 million samples. The sequential DFA was run on a single VM whereas both the PDFA and EPDFA were executed on 8 VMs. Furthermore, the PDFA was run using the default Hadoop configuration settings as shown in Table 1. The EPDFA was run on the GEP optimized Hadoop configuration settings. Table 6 lists a portion of the optimized Hadoop settings. Both the PDFA and the EPDFA were run 3 times each and average execution time was obtained. Figure 3 shows the execution times of the sequential DFA, the PDFA, and the EPDFA on different numbers of PMU data samples.

From Figure 3 it can be observed that the EPDFA performs better than both the DFA and PDFA. For example, the sequential DFA took 1690 minutes when processing 86.40 million samples whereas the PDFA and EPDFA took 103 minutes and 58 minutes, respectively, when processing the same number of samples. It is worth noting that, due to long execution times of the sequential DFA, it is hard to differentiate computationally between the PDFA and EPDFA. For this purpose, Figure 4 is plotted over the number of data samples in order to clearly show that the EPDFA is computationally faster than PDFA.

Based on results presented in the Figure 4, the computational speedup of the PDFA and the EPDFA when compared to DFA can be calculated using, respectively,

$$\text{Speedup}^{i}_{\text{EPDFA}} = \frac{T^{i}_{\text{EPDFA}}}{T^{i}_{\text{DFA}}}, \tag{5}$$

$$\text{Speedup}^{i}_{\text{PDFA}} = \frac{T^{i}_{\text{PDFA}}}{T^{i}_{\text{DFA}}}, \tag{6}$$

where i represents the number of PMU data samples in millions, $i \in [8.64, 86.40]$, T^{i}_{EPDFA} is the execution time of the EPDFA on i number of data samples, T^{i}_{DFA} is the execution time of the DFA on i number of data samples, and T^{i}_{PDFA} is the execution time of the PDFA on i number of data samples. The speedup results are shown in Figure 5.

Table 6: GEP recommended configuration parameter settings.

Configuration parameters	Optimized values				
Number of data samples in million	17.28	34.56	51.84	69.12	86.40
io.sort.factor	10	10	13	19	12
io.sort.mb	38	13	58	62	55
io.sort.spill.percent	0.90	0.90	0.90	0.89	0.90
mapred.reduce.tasks	14	14	13	2	16
mapreduce.tasktracker.map.tasks.maximum	8	6	8	8	8
mapreduce.tasktracker.reduce.tasks.maximum	1	1	1	2	1
mapred.child.java.opts	169	135	121	124	121
mapreduce.reduce.shuffle.input.buffer.percent	0.73	0.65	0.65	0.65	0.75
mapred.inmem.merge.threshold	200	200	201	202	201

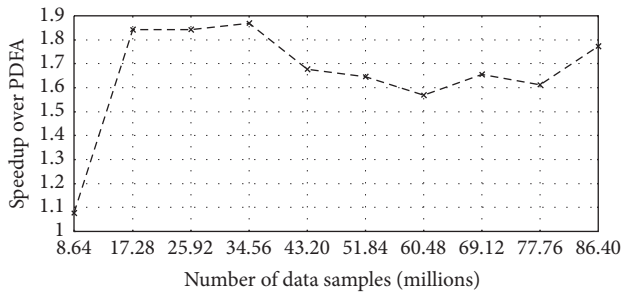

Figure 6: Speedup of the EPDFA over PDFA.

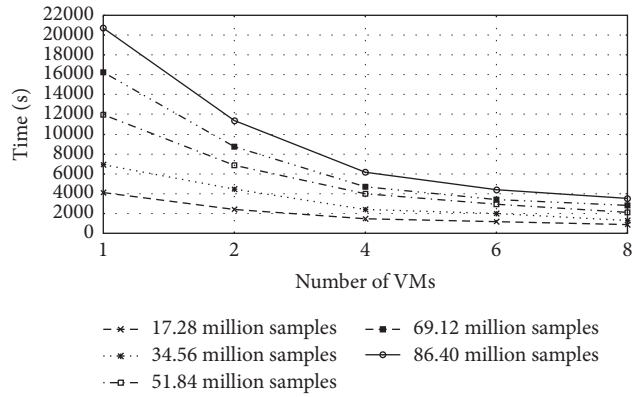

Figure 7: Computation scalability of EPDFA.

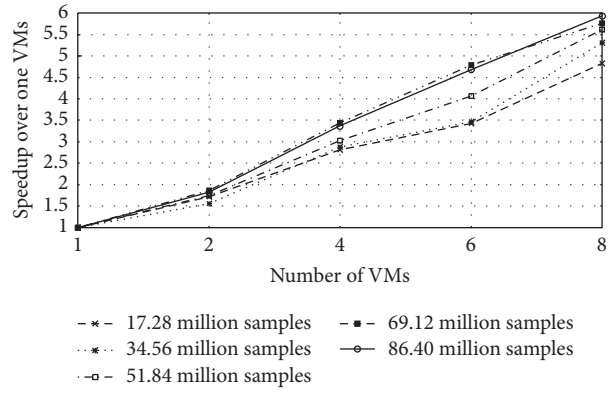

Figure 8: Speedup of the EPDFA over a single VM.

From Figure 5 it can be observed that, compared with the sequential DFA, the EPDFA has achieved a maximum speedup of 29.03 when processing 86.40 million samples. Alternatively, the PDFA has achieved a maximum speedup 17.33 when processing 69.12 million samples. The average speedup of the EPDFA when compared to the DFA is 26 times faster whereas the PDFA is 16 times faster. Furthermore, the maximum speedup of the EPDFA is 1.87 times faster than the PDFA as shown in Figure 6, whereas the minimum speedup is 1.08 times faster than the PDFA when processing 8.64 million samples.

The computational scalability of the EPDFA was also evaluated from the aspects of both VMs and PMU data samples. Figure 7 shows the execution times of the EPDFA in processing the 5 sets of PMU data samples with a varied number of VMs from 1 to 8. It can be observed that the execution time of the EPDFA is continuously decreased with an increasing number of VMs. Compared with the performance of the EPDFA running on a single VM, the EPDFA achieves the highest speedup on the largest number of data samples which is also clearly indicated by Figure 8. For example, when processing 8.64 million samples, the EPDFA running on 6 VMs is 3.43 times faster than running a single VM whereas it achieves a speedup of 4.83 on 8 VMs. Furthermore, when processing the 86.40 million samples, the EPDFA achieves a speedup of 4.68 on 6 VMs and 5.93 on 8 VMs.

6. Conclusion

Executing a Hadoop job using default parameter settings has led to performance issues. In this paper we have presented EPDFA to improve Hadoop performance by automatically tuning its configuration parameters. The optimized Hadoop framework can be utilized for scalable analytics on massive PMU data. The EPDFA achieved a maximum computational

speedup of 29.03 times faster than the sequential DFA and 1.87 times faster than a parallel DFA.

At present the Hadoop framework is highly applicable to off-line scalable data analytics. However, the high processing overhead associated with input and output files limits the application of Hadoop to on-line analysis of PMU data streams. Further research will apply in-memory processing techniques [34] in order to enable real-time data stream analytics for power system applications.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

References

[1] M. Zima, M. Larsson, P. Korba, C. Rehtanz, and G. Andersson, "Design aspects for wide-area monitoring and control systems," *Proceedings of the IEEE*, vol. 93, no. 5, pp. 980–996, 2005.

[2] M. Rihan, M. Ahmad, and M. S. Beg, "Phasor measurement units in the Indian smart grid," in *Proceedings of the 2011 IEEE PES International Conference on Innovative Smart Grid Technologies-India, ISGT India 2011*, pp. 261–267, India, December 2011.

[3] P. M. Ashton, G. A. Taylor, M. R. Irving, I. Pisica, A. M. Carter, and M. E. Bradley, "Novel application of detrended fluctuation analysis for state estimation using synchrophasor measurements," *IEEE Transactions on Power Systems*, vol. 28, no. 2, pp. 1930–1938, 2013.

[4] P. M. Ashton, G. A. Taylor, A. M. Carter, M. E. Bradley, and W. Hung, "Application of phasor measurement units to estimate power system inertial frequency response," in *Proceedings of the 2013 IEEE Power and Energy Society General Meeting, PES 2013*, Canada, July 2013.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design Implementation*, vol. 6, p. 10, 2004.

[6] M. Khan, P. M. Ashton, M. Li, G. A. Taylor, I. Pisica, and J. Liu, "Parallel detrended fluctuation analysis for fast event detection on massive pmu data," *IEEE Transactions on Smart Grid*, vol. 6, no. 1, pp. 360–368, 2015.

[7] Hadoop., "Apache Hadoop," https://hadoop.apache.org/.

[8] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce," in *Proceedings of the the VLDB Endowment*, vol. 2, pp. 1426–1437, 2009.

[9] A. Pavlo, E. Paulson, A. Rasin et al., "A comparison of approaches to large-scale data analysis," in *Proceedings of the International Conference on Management of Data and 28th Symposium on Principles of Database Systems, SIGMOD-PODS'09*, pp. 165–178, USA, July 2009.

[10] F. Bach, H. K. Çakmak, H. Maass, and U. Kuehnapfel, "Power grid time series data analysis with pig on a Hadoop cluster compared to multi core systems," in *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2013*, pp. 208–212, UK, March 2013.

[11] R. Hafen, T. D. Gibson, K. K. Van Dam, and T. Critchlow, "Large-scale exploratory analysis, cleaning, and modeling for event detection in real-world power systems data," in *Proceedings of the 3rd Int'l Workshop on High Performance Computing, Networking and Analytics for the Power Grid, HiPCNA-PG 2013 - Held in Conjunction with the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2013*, USA, November 2013.

[12] M. Edwards, A. Rambani, Y. Zhu, and M. Musavi, "Design of hadoop-based framework for analytics of large synchrophasor datasets," in *Proceedings of the 2012 Complex Adaptive Systems Conference*, pp. 254–258, USA, November 2012.

[13] P. Trachian, "Machine learning and windowed subsecond event detection on PMU data via hadoop and the openPDC," in *Proceedings of the IEEE PES General Meeting, PES 2010*, USA, July 2010.

[14] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of MapReduce programs," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.

[15] C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001.

[16] H. Herodotou, H. Lim, G. Luo et al., "Starfish: A self-tuning system for big data analytics," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, CIDR 2011*, pp. 261–272, usa, January 2011.

[17] D. Wu and A. Gokhale, "A self-tuning system based on application profiling and performance analysis for optimizing hadoop mapreduce cluster configuration," in *Proceedings of the 20th Annual International Conference on High Performance Computing, HiPC 2013*, pp. 89–98, India, December 2013.

[18] G. Liao, K. Datta, and T. L. Willke, "Gunther: Search-based auto-tuning of MapReduce," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics): Preface*, vol. 8097, pp. 406–419, 2013.

[19] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir, "Panacea: Towards holistic optimization of MapReduce applications," in *Proceedings of the 10th International Symposium on Code Generation and Optimization, CGO 2012*, pp. 33–43, USA, April 2012.

[20] Z. Yu, Z. Bei, H. Zhang et al., "RFHOC: A Random-Forest Approach to Auto-Tuning Hadoop's Configuration," *IEEE Xplore: IEEE Transactions on Parallel and Distributed*, vol. PP, no. 99, p. 1, 2015.

[21] H. Herodotou, "Hadoop Performance Models," Technical Report, CS-2011-05, 2011, http://www.cs.duke.edu/starfish/files/hadoop-models.pdf.

[22] R. Vallée-Rai, L. Hendren, P. Co, P. Lam, E. Gagnon, and V. Sundaresan, "Soot - A Java bytecode optimization framework," in *Proceedings of the 20th Annual CASCON Conference, CASCON 2010*, pp. 214–224, Canada, November 2010.

[23] Y. Li, K. Wang, Q. Guo et al., "Breaking the boundary for whole-system performance optimization of big data," in *Proceedings of the 2013 ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 2013*, pp. 126–131, China, September 2013.

[24] M. Khan, Y. Jin, M. Li, Y. Xiang, and C. Jiang, "Hadoop Performance Modeling for Job Estimation and Resource Provisioning," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 2, pp. 441–454, 2015.

[25] M. Khan, Z. Huang, M. Li, G. A. Taylor, and M. Khan, "Optimizing hadoop parameter settings with gene expression

programming guided PSO," *Concurrency Computation*, vol. 29, no. 3, Article ID e3786, 2016.

[26] K. Kambatla, A. Pathak, and H. Pucha, "Towards Optimizing Hadoop Provisioning in the Cloud," in *Proceedings of the in Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, 2009.

[27] T. White, *Hadoop:The Definitive Guide*, Yahoo press, 4th edition, 2015.

[28] S. Babu, "Towards automatic optimization of MapReduce programs," in *Proceedings of the Proceedings of the 1st ACM Symposium on Cloud Computing*, pp. 137–142, ACM, New York, NY, USA, June 2010.

[29] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*, Oxford University Press, Oxford, UK, 1996.

[30] J. H. Holland, *Adaptation in Natural and Artificial Systems*, MIT Press, Cambridge, Mass, USA, 1992.

[31] J. R. Koza, *Genetic Programming: on the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, Mass, USA, 1992.

[32] OpenPDC., "CodePlex," http://openpdc.codeplex.com/.

[33] Oracle Virtual Box., https://www.virtualbox.org/.

[34] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the 24th ACM Symposium on Operating Systems Principles, SOSP 2013*, pp. 423–438, USA, November 2013.