Memetic Comp. (2017) 9:183–197
DOI 10.1007/s12293-016-0190-5

REGULAR RESEARCH PAPER

An efficient parallel method for batched OS-ELM training using MapReduce

Shan Huang[1] · **Botao Wang**[1] · **Yuemei Chen**[1] · **Guoren Wang**[1] · **Ge Yu**[1]

Received: 27 September 2015 / Accepted: 20 April 2016 / Published online: 29 April 2016

Abstract In this era of big data, more and more models need to be trained to mine useful knowledge from large scale data. It has become a challenging problem to train multiple models accurately and efficiently so as to make full use of limited computing resources. As one of ELM variants, online sequential extreme learning machine (OS-ELM) provides a method to learn from incremental data. MapReduce, which provides a simple, scalable and fault-tolerant framework, can be utilized for large scale learning. In this paper, we propose an efficient parallel method for batched online sequential extreme learning machine (BPOS-ELM) training using MapReduce. Map execution time is estimated with historical statistics, where regression method and inverse distance weighted interpolation method are used. Reduce execution time is estimated based on complexity analysis and regression method. Based on the estimations, BPOS-ELM generates a Map execution plan and a Reduce execution plan. Finally, BPOS-ELM launches one MapReduce job to train multiple OS-ELM models according to the generated execution plan, and collects execution information to further improve estimation accuracy. Our proposal is evaluated with real and synthetic data. The experimental results show that the accuracy of BPOS-ELM is at the same level as those of OS-ELM and parallel OS-ELM (POS-ELM) with higher training efficiencies.

Keywords Parallel learning · Extreme learning machine · MapReduce · Sequential learning

✉ Botao Wang
wangbotao@ise.neu.edu.cn

1 School of Computer Science and Engineering, Northeastern University, Shenyang 110819, Liaoning, China

1 Introduction

With the development of technology and the widespread use of machine learning, more and more models need to be trained to mine useful knowledge from large scale data. It has become a challenging problem to train multiple models accurately and efficiently so as to make full use of limited computing resources. For example, in a machine learning organization where high performance computing cluster is a limited resource, researchers must schedule the jobs on the cluster legitimately to make full use of the cluster. For another example, resizable cloud hosting services such as Amazon elastic compute cloud (EC2) [1], which become more and more popular, enable their users to rent large amount of virtual machines by the hour at lower costs than operating a data center year-round. It is important for users to schedule multiple jobs running on this kind of environment as the rented virtual machines are charged by the used time.

Extreme learning machine (ELM), which was proposed based on single-hidden layer feed-forward neural networks (SLFNs) [2], has been verified to have high learning speed as well as high accuracy [3]. It has also been proved that ELM has universal approximation capability and classification capability [4]. As one of ELM variants, online sequential extreme learning machine (OS-ELM) [5] supports incremental learning.

MapReduce [6] is a well-known framework which supports large scale data processing and analyzing on a large cluster of commodity machines. As an open-source implementation of MapReduce framework, Apache Hadoop [7] has been used in various fields including machine learning. Recent research has studied on outsourcing calculations of ELM to resourceful workstation [8] or parallelizing ELM [9–12], however the strategies are not suitable to parallelize OS-ELM. POS-ELM [13] supports training one single OS-

ELM model in parallel with MapReduce, but it does not support training multiple OS-ELM models efficiently.

In this paper, we propose an efficient parallel method for batched online sequential extreme learning machine (BPOS-ELM) training using MapReduce. BPOS-ELM first estimates the execution time of Map and Reduce tasks for each OS-ELM model based on historical statistics. Then the estimations are employed to generate a Map and a Reduce execution plans based on the greedy strategy. After that, BPOS-ELM launches a MapReduce job to train multiple OS-ELM models. At the same time, BPOS-ELM collects execution information of selected Map tasks and Reduce tasks, and merges them to historical statistics to improve the accuracy of time estimation. The algorithm is evaluated with real and synthetic data. The accuracy is at the same level as those of OS-ELM and POS-ELM. The speedup reaches 10× on a cluster with maximum 32 cores.

The main contributions of this paper can be summarized as follows:

1. We propose an efficient parallel method for batched online sequential extreme learning machine (BPOS-ELM) training using MapReduce.
2. According to historical statistics, the costs of Map tasks are estimated with regression method and inverse distance weighted interpolation method, and the costs of Reduce tasks are estimated based on complexity analysis and regression techniques.
3. Map execution plan and Reduce execution plan are generated and executed to train multiple OS-ELM models efficiently.
4. BPOS-ELM algorithm is evaluated with synthetic and real data and the experimental results show that the speedup of it reaches 10× on a cluster with maximum 32 cores.

The remainder of this paper is organized as follows: Sect. 2 reviews related work. Section 3 briefly introduces MapReduce framework, ELM, OS-ELM and POS-ELM. The problem definition and basic idea are presented in Sect. 4. Section 5 describes the efficient parallel method for batched online sequential extreme learning machine using MapReduce in detail. An extensive experimental evaluation of BPOS-ELM is presented in Sect. 6. A brief conclusion is presented in Sect. 7.

2 Related work

ELM and its variants have shown powerful capability in handling large data. Cao et al. [14] reviewed recent applications that use ELM and its variants to solve problems including image processing, video processing and medical signal processing.

Existing approaches that improve training speed of ELM and its variants include outsourcing calculations to resourceful workstation [8] and parallelizing them [9–13] in a cluster. Lin et al. [8] proposed a method that reduces ELM training time by outsourcing to the cloud to handle large data applications. The results in [8] show that the method increases training speed of ELM dramatically. Different from this approach, our proposal focuses on improving training speed of multiple OS-ELM models.

There is also recent research on improving training speed of ELM by parallelizing the matrix calculations in a cluster. He et al. [9] proposed a parallel ELM algorithm which uses one MapReduce job to map training instances to hidden layer nodes and another MapReduce job to calculate the product of hidden layer output matrix and its transpose in parallel. Xiang et al. [10] used the algorithm in [9] for intrusion detection in big data environment. Xin et al. [11] proposed another MapReduce based ELM algorithm which uses one MapReduce job to map training instances to hidden layer nodes and calculate the product of hidden layer output matrix and its transpose. Heewijk et al. [12] accelerated the training speed of ELM using CUDA [15] technique and MATLAB parallel computing toolbox [16]. It can be observed from experimental results in [12] that the speedup of the algorithm in [12] reaches 3.4× on 4 machines. These algorithms support large scale learning, but they do not support learning from incremental data that commonly generated in our daily life. As the calculation procedure of OS-ELM is different from that of ELM, the methods above are not suitable to parallelize OS-ELM. POS-ELM [13] supports training one single OS-ELM model in parallel with MapReduce, but it does not support training multiple OS-ELM models efficiently.

3 Preliminaries

In this section, we briefly introduce MapReduce, ELM, OS-ELM and POS-ELM.

3.1 MapReduce

MapReduce [6], which was first proposed by Google, is a framework for large scale data processing and analyzing. As one open-source version MapReduce framework, Hadoop [7] has been used by many companies and organizations. MapReduce hides the details of the complex processing in distributed computing such as load balancing, network performance and fault tolerance, so it allows users to implement parallel algorithms on a large cluster of commodity machines. In a cluster of MapReduce, one machine works as Master node and the others work as Slave nodes. The Master node is responsible for task scheduling and the Slave nodes are responsible for executing tasks that are assigned by the Master node.

MapReduce provides two main procedures for users to implement their logics, the $map()$ procedure and the $reduce()$ procedure. The types involved in these two procedures are listed as below:

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$reduce(k_2, list(v_2)) \rightarrow list(k_3, v_3)$$

The $map()$ procedure takes a key-value pair (k_1, v_1) as input, processes it with user's logic and generates zero or more output key-value pairs (k_2, v_2). The $reduce()$ procedure combines all the key-value pairs with the same key, iterates through the values that are associated with that key and produces zero or more outputs.

3.2 ELM and OS-ELM

Extreme learning machine (ELM) is designed based on single-hidden layer feed-forward neural networks (SLFNs) [2]. Given N distinct arbitrary instances $(\mathbf{x}_j, \mathbf{t}_j)$, where $\mathbf{x}_j = [x_{j1}, x_{j2}, \ldots, x_{jn}]^T \in \mathbf{R}^n$ is the attribute matrix and $\mathbf{t}_j = [t_{j1}, t_{j2}, \ldots, t_{jm}]^T \in \mathbf{R}^m$ is the tag matrix, ELM is defined as Formula (1).

$$\mathbf{H}\boldsymbol{\beta} = \mathbf{T} \tag{1}$$

where

$$\mathbf{H}(\mathbf{w}_1, \ldots, \mathbf{w}_{\tilde{N}}, b_1, \ldots, b_{\tilde{N}}, \mathbf{x}_1, \ldots, \mathbf{x}_N) = \begin{bmatrix} g(\mathbf{w}_1 \cdot \mathbf{x}_1, b_1) & \ldots & g(\mathbf{w}_{\tilde{N}} \cdot b_{\tilde{N}} + \mathbf{x}_1) \\ \vdots & \ldots & \vdots \\ g(\mathbf{w}_1 \cdot \mathbf{x}_N, b_1) & \ldots & g(\mathbf{w}_{\tilde{N}} \cdot b_{\tilde{N}} + \mathbf{x}_N) \end{bmatrix}_{N \times \tilde{N}} \tag{2}$$

$\mathbf{w}_i = [w_{i1}, w_{i2}, \ldots, w_{in}]^T$ is the weights vector between the ith hidden node and the input nodes, b_i is the threshold of the ith hidden node, $\boldsymbol{\beta} = [\boldsymbol{\beta}_1^T, \ldots, \boldsymbol{\beta}_{\tilde{N}}^T]^T_{m \times \tilde{N}}$, $\mathbf{T} = [\mathbf{t}_1^T, \ldots, \mathbf{t}_N^T]^T_{m \times N}$ and $\tilde{N}$ is the number of hidden layer nodes. $\mathbf{H}$ is called the hidden layer output matrix of the neural network. $\boldsymbol{\beta}$ is called output weights matrix. It has been proved in [17] that the hidden layer parameters can be randomly generated if the activation function g is infinitely differentiable in any interval. It has also been proved that ELM has universal approximation capability and classification capability [4].

As a variant of ELM, online sequential extreme learning machine (OS-ELM) [5] has the ability to learn data chunk by chunk with fixed or varying sizes instead of batch learning. OS-ELM algorithm is divided into two phases, initialization phase and sequential learning phase.

Initialization phase uses a small chunk of training data to initialize the learning machine. First, initial hidden layer output matrix $\mathbf{H}_0$ is calculated according to Formula (2). Then $\boldsymbol{\beta}_0$ and $\mathbf{P}_0$ are calculated according to Formula (3) and (4) respectively.

$$\boldsymbol{\beta}_0 = {\mathbf{H}_0}^{\dagger}\mathbf{T}_0 \tag{3}$$
$$\mathbf{P}_0 = (\mathbf{H}_0^T\mathbf{H}_0)^{-1} \tag{4}$$

where ${\mathbf{H}_0}^{\dagger}$ denotes the Moore-Penrose generalized inverse of matrix $\mathbf{H}_0$.

Sequential learning phase updates $\boldsymbol{\beta}_k$ and $\mathbf{P}_k$ for the kth chunk of training data with N_k distinct arbitrary instances according to Formula (5) and (6) respectively.

$$\boldsymbol{\beta}_k = \boldsymbol{\beta}_{k-1} + \mathbf{P}_k\mathbf{H}_k^T(\mathbf{T}_k - \mathbf{H}_k\boldsymbol{\beta}_{k-1}) \tag{5}$$
$$\mathbf{P}_k = \mathbf{P}_{k-1} - \mathbf{P}_{k-1}\mathbf{H}_k^T(\mathbf{I} + \mathbf{H}_k\mathbf{P}_{k-1}\mathbf{H}_k^T)^{-1}\mathbf{H}_k\mathbf{P}_{k-1} \tag{6}$$

For more information about ELM and OS-ELM, please refer to [2,5].

3.3 POS-ELM

Parallel online sequential extreme learning machine (POS-ELM) algorithm [13] was designed based on OS-ELM and implemented on MapReduce framework. The basic idea of

Algorithm 1: POS-ELM Map()

```
Input: (Key, Value): Key is the offset in bytes, value is a sample
       pair $(x_i, t_i) \in (\mathbf{X}, \mathbf{T})$ where $0 \le i \le| (\mathbf{X}, \mathbf{T}) |$;
Result:
$blockID$: Output key;
$\mathbf{H}_b$:Output weight;
$\mathbf{T}_b$:Observation value vector;
1 $blocknum$=0;
2 for $blocknum \le BLOCKSIZE$ do
3     merge a sample pair into $block$;
4     $blocknum$++;
5 calculate hidden layer output matrix $\mathbf{H}_b$;
6 output ( $blockID$++, $(\mathbf{H}_b, \mathbf{T}_b)$ );
```

Algorithm 2: POS-ELM Reduce()

```
Input:
Set of $(b, value)$: $b$ is the identifier of a block, $value$ is a vector
pair $(\mathbf{H}_b, \mathbf{T}_b)$;
Result:
$\boldsymbol{\beta}$: output weight vector
1 $b$=0;
2 for $b \le B$ do
3     calculate
      $\mathbf{P}_{b+1} = \mathbf{P}_b - \mathbf{P}_b\mathbf{H}_{b+1}^{\mathbf{T}}(\mathbf{I} + \mathbf{H}_{b+1}\mathbf{P}_b\mathbf{H}_{b+1}^{\mathbf{T}})^{-1}\mathbf{H}_{b+1}\mathbf{P}_b$;
4     calculate output weight
      $\boldsymbol{\beta}_{b+1} = \boldsymbol{\beta}_b + \mathbf{P}_{b+1}\mathbf{H}_{b+1}^{\mathbf{T}}(\mathbf{T}_{b+1} - \mathbf{H}_{b+1}\boldsymbol{\beta}_b)$;
5     $b$=$b$+1;
```

POS-ELM is to parallelize the calculation of hidden layer output matrix of original OS-ELM. Algorithm 1 and Algorithm 2 show the Map phase and Reduce phase algorithms of POS-ELM, respectively. For more details of POS-ELM please refer to [13].

4 Problem definition and basic idea

POS-ELM [13] supports training one single OS-ELM model in parallel with MapReduce, but it does not support training multiple OS-ELM models efficiently. For the training of multiple OS-ELM models, there are three main challenges to be solved.

1. Estimating the execution time of training POS-ELM in Map phase and Reduce phase accurately, which is the basis of execution plan generation, is a challenging problem.
2. Generating the most optimized execution plan that all the jobs complete in minimum possible time is an NP-hard problem, so it is necessary to find heuristic rules to generate an approximate optimal execution plan. The problem of generating the most optimized execution plan is the same as "multiprocessor scheduling" problem which has been proven to be an NP-complete problem in [18]. Because NP-complete problems are included in NP-hard problems, generating the most optimized execution plan is an NP-hard problem.
3. How to reorganize execution procedure of POS-ELM algorithm to make it possible to train multiple OS-ELMs in one MapReduce job. So the problems are how to estimate the cost of Map task and Reduce task of each OS-ELM model, how to create an execution plan for the training of multiple models and how to reorganize the POS-ELM algorithm.

The basic idea of BPOS-ELM algorithm is to generate an execution plan which trains multiple OS-ELM models in one MapReduce job according to the estimations of Map execution time and Reduce execution time. The cost of calculation of POS-ELM in Map phase and Reduce phase is associated with parameters as shown in Table 1, and the Map execution time and Reduce execution time are estimated according to historical statistics.

5 BPOS-ELM

The main procedures of BPOS-ELM algorithm are first described in Sect. 5.1. Then ID assigning is introduced in Sect. 5.2. Execution time estimation is introduced in detail in Sect. 5.3. Section 5.4 describes execution plan generation in detail. Section 5.5 introduces plan execution and the execution information collection is introduced in Sect. 5.6.

Table 1 Notations used in BPOS-ELM

Parameter	Description
B	Block size
N	Number of training data
D	Number of attributes in training data
M	Number of training models
M_s	Number of actual execution information
$\tilde{N}$	Number of hidden layer nodes
C	Number of classifications

5.1 Overview

As shown in Fig. 1, BPOS-ELM algorithm uses the following steps to train multiple OS-ELM models in one MapReduce job.

Step 1 Each model is assigned with a unique ID which is used to specify it from the other training models. The ID of each model is associated with the model all over the following steps until the job execution completes.
Step 2 The Map execution time and Reduce execution time are estimated according to historical statistics described with parameters shown in Table 1.
Step 3 A job execution plan is generated according to the estimations of Map execution time and Reduce execution time in previous step. The Map tasks that have short execution time are treated as unit executions and the Map tasks that have long execution time are split into multiple unit executions. Each Reduce task is treated as one unit execution. The Map execution plan and Reduce execution plan are generated based on the unit executions.
Step 4 The generated execution plans are executed to train the models.
Step 5 The actual execution time of the selected Map tasks and Reduce tasks is collected for future time estimation.

5.2 ID assigning

In this step, each OS-ELM model is assigned with a unique ID. In the following steps, the BPOS-ELM algorithm uses these IDs to distinguish one model from the other models.

To facilitate the description of BPOS-ELM, we assume that there are M OS-ELM models to be trained and each model is associated with an ID from 1 to M.

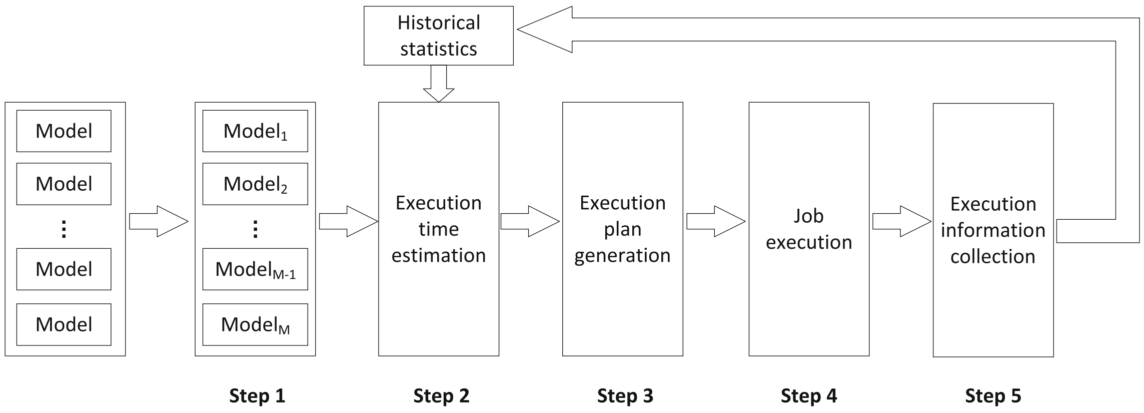


Fig. 1 Execution framework of BPOS-ELM

5.3 Execution time estimation

The Map execution time and Reduce execution time are estimated according to historical job execution time statistics. At the same time, the actual execution time of selected Map tasks and Reduce tasks are also collected for future time estimation. In the following, the estimations of Map execution time and Reduce execution time are introduced in detail.

5.3.1 Map execution time estimation

We provide two methods to estimate Map execution time, (1) regression method and (2) Inverse Distance Weighted (IDW) interpolation method.

(1) Regression method

One way to estimate Map execution time is to use regression method. We use B, N, D, $\tilde{N}$ and C as regression parameters and Map execution time as regression target. So the Map execution time estimation is a multi-parameter regression problem. OS-ELM is used as the regression model as it has the ability to learn statistics incrementally.

Algorithm 3 shows the regression method. There are two procedures in this method, **MSMapRegression** procedure (lines 1–5) and **EMapRegression** procedure (lines 6–9).

MSMapRegression procedure is used to merge actual execution information to Map execution time statistics. There are two inputs of this procedure, the actual execution information and current Map execution time statistics. First, hidden layer output matrix is calculated according to Formula (2) (line 2). Then matrix $\mathbf{P}_k$ is updated according to Formula (6) (line 3). Finally, matrix $\boldsymbol{\beta}_k$ is updated according to Formula 5 (line 4). Matrices $\mathbf{P}_k$ and $\boldsymbol{\beta}_k$ represent the current Map execution time statistics.

EMapRegression procedure is used to estimate Map execution time according to the Map execution time statistics.

Algorithm 3: Map execution time estimation using regression method

```
Input: $(\mathbf{x}_j, \mathbf{t}_j)(1 \le j \le M_s)$ : $M_s$ distinct actual execution
       information, where $\mathbf{x}_j = [x_{j1}, x_{j2}, ..., x_{j5}]^T$, $x_{j1}, x_{j2}$,
       $x_{j3}, x_{j4}$ and $x_{j5}$ represent $B_j$,$N_j$,$D_j$,$\tilde{N}_j$ and $C_j$,
       respectively. $\mathbf{t}_j$ represents the actual Map execution time.
$(\mathbf{P}_{k-1}, \boldsymbol{\beta}_{k-1})$: current Map execution time statistics.
Result: $(\mathbf{P}_k, \boldsymbol{\beta}_k)$: updated Map execution time statistics.
1 MSMapRegression()
2     calculate $\mathbf{H}_k$ with Formula (2);
3     update $\mathbf{P}_k$ with Formula (6);
4     update $\boldsymbol{\beta}_k$ with Formula (5);
5     return $(\mathbf{P}_k, \boldsymbol{\beta}_k)$;
Input: $\mathbf{x}_j(1 \le j \le M)$ : $M$ distinct actual execution parameter
       vector, where $\mathbf{x}_j = [x_{j1}, x_{j2}, ..., x_{j5}]^T$, $x_{j1}, x_{j2}, x_{j3}, x_{j4}$
       and $x_{j5}$ represent $B_j$,$N_j$,$D_j$,$\tilde{N}_j$ and $C_j$, respectively.
$\boldsymbol{\beta}_k$: current Map execution time statistics.
Result: $\mathbf{T} = [t_1, t_2, ..., t_M]^T$ :estimated Map execution time.
6 EMapRegression()
7     calculate $\mathbf{H}$ with Formula (2);
8     $\mathbf{T} = \mathbf{H}\boldsymbol{\beta}_k$;
9     return $\mathbf{T}$;
```

There are two inputs of this procedure, the current Map execution time statistics and the actual execution parameter vector. First, hidden layer output matrix is calculated according to Formula (2) (line 7). Then the Map execution time is estimated according to the Map execution time statistics (line 8).

(2) Inverse distance weighted interpolation method

Inverse distance weighted (IDW) [19] interpolation method is another way to estimate Map execution time. First, each parameter of job execution information is mapped to one dimension at a multi-dimensional space, so the historical statistics are mapped to a set of points in the space. Then k nearest neighbour points of the point to be estimated in the space are selected and used to estimate Map execution time. After that, IDW interpolation method shown as Formula (7) is used to estimate Map execution time.

$$t_{map}(\mathbf{x}) \approx \begin{cases} \dfrac{\sum_{i=1}^{k} w_i(\mathbf{x}) t_i}{\sum_{i=1}^{k} w_i(\mathbf{x})}, & \text{if } d(\mathbf{x}, \mathbf{x}_i) \neq 0 \quad \text{for all } i \\ t_i, & \text{if } d(\mathbf{x}, \mathbf{x}_i) = 0 \quad \text{for some } i \end{cases} \tag{7}$$

where $w_i(\mathbf{x}) = \dfrac{1}{d(\mathbf{x}, \mathbf{x}_i)^p}$ is a simple IDW weighting function, as defined by Shepard [19], $\mathbf{x}$ denotes the parameter vector of point to be predict, $\mathbf{x_i}$ is the selected k nearest neighbour points, d is a given distance from the point $\mathbf{x_i}$ to point $\mathbf{x}$ and p is a positive real number, called the power parameter. Euclidean distance is used to measure the distance between two points.

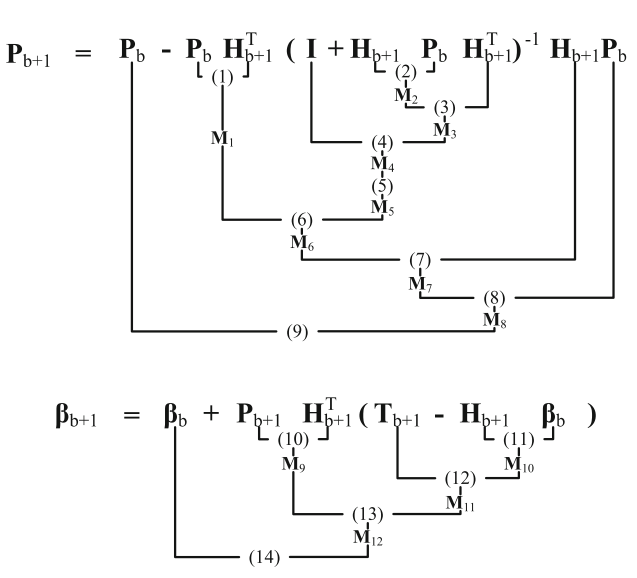
Algorithm 4 shows the Inverse Distance Weighted (IDW) interpolation method. There are two procedures in this method, **MSMapIDW** procedure (lines 1–4) and **EMapIDW** procedure (lines 5–9).

MSMapIDW procedure is used to merge actual execution information to Map execution time statistics. Similar to **MSMapRegression**, there are two inputs of this procedure, the actual execution information and current Map execution time statistics. Each of the actual execution information is added to the Map execution time statistics (lines 2–3).

EMapIDW procedure is used to estimated Map execution time according to the Map execution time statistic. Similar to **EMapRegression**, there are two inputs of this procedure, the current Map execution time statistics and the actual execution parameter vector. For each of the parameter vector, the k nearest neighbour statistics of the Map execution time statistics are found (line 7). Then the Map execution time is estimated according with IDW method according to Formula (7) (line 8).

Algorithm 4: Map execution time estimation using kNN and IDW

```
Input: $(\mathbf{x}_j, \mathbf{t}_j)(1 \le j \le M_s)$ : $M_s$ distinct actual execution
       information, where $\mathbf{x}_j = [x_{j1}, x_{j2}, ..., x_{j5}]^T$, $x_{j1}, x_{j2}$,
       $x_{j3}, x_{j4}$ and $x_{j5}$ represent $B_j, N_j, D_j, \tilde{N}_j$ and $C_j$,
       respectively. $\mathbf{t}_j$ represents the actual Map execution time.
$list$: current Map execution time statistics.
Result: $list$: updated Map execution time statistics.
1 MSMapIDW()
2     for $i = 1$ to $M_s$ do
3         list.add(($\mathbf{x}_i, \mathbf{t}_i$));
4     return $list$;
Input: $\mathbf{x}_j(1 \le j \le M)$ : $M$ distinct actual execution parameter
       vector, where $\mathbf{x}_j = [x_{j1}, x_{j2}, ..., x_{j5}]^T$, $x_{j1}, x_{j2}, x_{j3}, x_{j4}$
       and $x_{j5}$ represent $B_j, N_j, D_j, \tilde{N}_j$ and $C_j$, respectively.
($list$): current Map execution time statistics.
Result:
$\mathbf{T} = [t_1, t_2, ..., t_M]^T$ :estimated Map execution time.
5 EMapIDW()
6     for $i = 1$ to $M_s$ do
7         ($\mathbf{x'}_i, \mathbf{t'}_i$)= kNN(list);
8         $\mathbf{t}_i$ = IDW($\mathbf{x'}_i, \mathbf{t'}_i$);
9     return $\mathbf{T}$;
```



$\mathbf{P}_{b+1} = \mathbf{P}_b - \mathbf{P}_b \mathbf{H}^T_{b+1} (\mathbf{I} + \mathbf{H}_{b+1} \mathbf{P}_b \mathbf{H}^T_{b+1})^{-1} \mathbf{H}_{b+1}\mathbf{P}_b$
(1) M1
(2) M2
(3) M3
(4) M4
(5) M5
(6) M6
(7) M7
(8) M8
(9)
$\boldsymbol{\beta}_{b+1} = \boldsymbol{\beta}_b + \mathbf{P}_{b+1} \mathbf{H}^T_{b+1} (\mathbf{T}_{b+1} - \mathbf{H}_{b+1} \boldsymbol{\beta}_b)$
(10) M9
(11) M10
(12) M11
(13) M12
(14)

Fig. 2 Calculation steps in Algorithm 2. $\boldsymbol{M}_n(1 \le n \le 12)$ are temporary matrices during the calculation procedure

5.3.2 Reduce execution time estimation

Reduce execution time is estimated based on complexity analysis and regression techniques. Figure 2 shows the calculation steps in Algorithm 2 and the calculation complexity of each step is listed in Table 2. There are three main kinds of calculations in Reduce phase of POS-ELM, matrix multiplication, matrix addition and matrix inversion. The calculations in Table 2 repeats $\dfrac{N}{B}$ times, so the Reduce execution time is estimated by Formula (8). In Formula (8), $\alpha_n(1 \le n \le 5)$ are the factors that need to be determined using historical statistics. According to the above analysis, the Reduce execution time estimation transforms to a multi-parameter regression problem so it can be solved with regression techniques. In this paper, OS-ELM is used as the regression model as it has the ability to learn statistics incrementally.

$$\begin{aligned} t_{red} &\approx \frac{N}{B}(\alpha_1 B^3 + \alpha_2 B^2 \tilde{N} + \alpha_3 B \tilde{N}^2 + \alpha_4 B \tilde{N} C \\ &\quad + \alpha_5(B^2 + BC + \tilde{N}^2 + \tilde{N}C)) \\ &= N\left(\alpha_1 B^2 + \alpha_2 B\tilde{N} + \alpha_3 \tilde{N}^2 + \alpha_4 \tilde{N} C \right. \\ &\quad \left. + \alpha_5\left(B + C + \frac{\tilde{N}^2}{B} + \frac{\tilde{N}C}{B}\right)\right) \end{aligned} \tag{8}$$

Algorithm 5 shows the algorithm of Reduce execution time estimation. There are two procedures in this algorithm,

Table 2 Calculation complexities in Reduce phase of POS-ELM

Step	Calculation	Complexity	Calculation type
(1)	$\mathbf{M}_1 = \mathbf{P}_b\mathbf{H}_{b+1}^{\mathbf{T}}$	$O(B\tilde{N}^2)$	Matrix multiplication
(2)	$\mathbf{M}_2 = \mathbf{H}_{b+1}\mathbf{P}_b$	$O(B\tilde{N}^2)$	Matrix multiplication
(3)	$\mathbf{M}_3 = \mathbf{M}_2\mathbf{H}_{b+1}^{\mathbf{T}}$	$O(B^2\tilde{N})$	Matrix multiplication
(4)	$\mathbf{M}_4 = \mathbf{I} + \mathbf{M}_3$	$O(B^2)$	Matrix addition
(5)	$\mathbf{M}_5 = \mathbf{M}_4^{-1}$	$O(B^3)$	Matrix inversion
(6)	$\mathbf{M}_6 = \mathbf{M}_1\mathbf{M}_5$	$O(B\tilde{N}^2)$	Matrix multiplication
(7)	$\mathbf{M}_7 = \mathbf{M}_6\mathbf{H}_{b+1}$	$O(B^2\tilde{N})$	Matrix multiplication
(8)	$\mathbf{M}_8 = \mathbf{M}_7\mathbf{P}_b$	$O(B\tilde{N}^2)$	Matrix multiplication
(9)	$\mathbf{P}_{b+1} = \mathbf{P}_b - \mathbf{M}_8$	$O(\tilde{N}^2)$	Matrix addition
(10)	$\mathbf{M}_9 = \mathbf{P}_{b+1}\mathbf{H}_{b+1}^{\mathbf{T}}$	$O(B\tilde{N}^2)$	Matrix multiplication
(11)	$\mathbf{M}_{10} = \mathbf{H}_{b+1}\boldsymbol{\beta}_b$	$O(B\tilde{N}C)$	Matrix multiplication
(12)	$\mathbf{M}_{11} = \mathbf{T}_{b+1} - \mathbf{M}_{10}$	$O(BC)$	Matrix addition
(13)	$\mathbf{M}_{12} = \mathbf{M}_9\mathbf{M}_{11}$	$O(B\tilde{N}C)$	Matrix multiplication
(14)	$\boldsymbol{\beta}_{b+1} = \boldsymbol{\beta}_b + \mathbf{M}_{12}$	$O(\tilde{N}C)$	Matrix addition

MSReduce procedure (lines 1–13) and **EReduce** procedure (lines 14–26).

MSReduce procedure takes actual execution information as input and merges them to the Reduce execution time statistics. The attributes for regression are first calculated for each instance of the actual execution information (lines 2–9). Then matrix $\mathbf{H}_k$ is calculated according to Formula (2) (line 10). Finally, matrices $\mathbf{P}_k$ and $\boldsymbol{\beta}_k$ are updated according to Formula (5) and (6) (lines 11–12), respectively.

EReduce procedure takes actual execution parameters and estimates Reduce execution time as output. The attributes are first generates for each of the parameter vector instances (lines 15–21). Then matrix $\mathbf{H}$ is calculated according to Formula (2) (line 22). Matrix $\mathbf{T}$ is calculated with current Reduce execution time statistic according to Formula (1) (line 23). Finally, the estimated Reduce execution time is generated by multiplying each element of $\mathbf{T}$ with corresponding N (lines 24–25).

5.4 Execution plan generation

BPOS-ELM generates a Map execution plan and a Reduce execution plan with greedy strategies. The details of Map execution plan generation and Reduce execution plan generation are described in Sects. 5.4.1 and 5.4.2 respectively.

5.4.1 Map execution plan generation

The execution plan generation algorithm of Map phase is shown in Algorithm 6. The algorithm needs an array of OS-ELM models as input and generates the execution plan of Map phase of BPOS-ELM as output. It first calculates the predictable average execution time of Map tasks (lines 1–3). Then it scans the OS-ELM models and processes them differently according to the estimated Map execution time. The models whose estimated Map execution time is less than average time are treated as unit executions during the execution plan generation (lines 5–6). The models whose estimated Map execution time is more than average time are split to multiple unit executions (lines 7–11). Relax factor α which is more than 1 is used to make the models whose Map execution time is at the same level as average time not split into more unit executions. After generating the list of unit executions, heuristic algorithm GeneratePlan is executed to generate Map execution plan.

The GeneratePlan algorithm is shown in Algorithm 7, which is used in both Map execution plan generation and Reduce execution plan generation. There are three inputs, a list of unit executions, the number of tasks and expected execution time for each task. The output of this algorithm is the execution plan for Map phase or Reduce phase. When the number of unit executions in the list is less than that of tasks, each of the unit execution is assigned to each task (lines 1–3). Otherwise, greedy strategy is used to generate execution plan. $Unassigned$ is initialized and used to count the number of unassigned unit executions in the list (line 5). First, the list of unit executions is sorted by estimated execution time in descending order (line 6). Then the sorted list is scanned and the unit executions in it are added to the execution plan (lines 7–16). The assigned unit executions are skipped (lines 8–9) and the loop is broken when $Count$ exceeds the number of tasks (lines 10–11). After that, the unassigned unit execution which has the longest execution time is added to execution plan (line 12) and the algorithm scans the remaining list to find the suitable unit execution and add it to execution plan recursively (lines 14–15). Finally, the algorithm scans the

Algorithm 5: Reduce execution time estimation

```
Input: $(B_j, N_j, D_j, \tilde{N}_j, C_j, \mathbf{t}_j)(1 \le j \le M_s)$ : $M_s$ distinct
       actual execution information, where $\mathbf{t}_j$ represents the
       actual Reduce execution time.
($\mathbf{P}_{k-1}, \boldsymbol{\beta}_{k-1}$): current Reduce execution time statistics.
Result: ($\mathbf{P}_k, \boldsymbol{\beta}_k$): updated Reduce execution time statistics.
MSReduce()
    for $i = 1$ to $M_s$ do
        $x_{i1} = (B_i)^2$;
        $x_{i2} = B_i\tilde{N}_i$;
        $x_{i3} = (\tilde{N}_i)^2$;
        $x_{i4} = \tilde{N}_i C_i$;
        $x_{i5} = B_i + C_i + \frac{\tilde{N}_i^2}{B_i} + \frac{\tilde{N}_i C_i}{B_i}$);
        $\mathbf{x}_i = [x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}]^T$;
        $t_i = \frac{t_i}{N_i}$;
    calculate $\mathbf{H}_k$ with Formula (2);
    update $\mathbf{P}_k$ with Formula (6);
    update $\boldsymbol{\beta}_k$ with Formula (5);
    return ($\mathbf{P}_k, \boldsymbol{\beta}_k$);
Input: $(B_j, N_j, D_j, \tilde{N}_j, C_j)(1 \le j \le M)$ : $M$ distinct actual
       execution parameter vector instances.
$\boldsymbol{\beta}_k$: current Reduce execution time statistics.
Result: $\mathbf{T} = [t_1, t_2, ..., t_M]^T$ :estimated Reduce execution time.
EReduce()
    for $i = 1$ to $M$ do
        $x_{i1} = (B_i)^2$;
        $x_{i2} = B_i\tilde{N}_i$;
        $x_{i3} = (\tilde{N}_i)^2$;
        $x_{i4} = \tilde{N}_i C_i$;
        $x_{i5} = B_i + C_i + \frac{\tilde{N}_i^2}{B_i} + \frac{\tilde{N}_i C_i}{B_i}$);
        $\mathbf{x}_i = [x_{i1}, x_{i2}, x_{i3}, x_{i4}, x_{i5}]^T$;
    calculate $\mathbf{H}$ with Formula (2);
    $\mathbf{T} = \mathbf{H}\boldsymbol{\beta}_k$;
    for $i = 1$ to $M$ do
        $t_i = t_i * N_i$ ;
    return $\mathbf{T}$;
```

list of unit executions again and adds the unassigned unit execution to the expected shortest task (lines 17–20).

5.4.2 Reduce execution plan generation

The execution plan generation algorithm of Reduce phase is shown in Algorithm 8. This algorithm needs an array of OS-ELM models as input and generates the execution plan of Reduce phase of BPOS-ELM as output. The algorithm first calculates the expected average execution time of Reduce tasks (lines 1–3). Then the algorithm scans the OS-ELM models and adds them to the list of unit executions (lines 4–5). As the calculations of POS-ELM algorithm in Reduce phase is indivisible, each Reduce task is treated as a unit execution. Since *start* and *end* are not used in Reduce execution plan generation, they are set to 0 to be compatible with GeneratePlan algorithm. After generating the list of unit executions, heuristic algorithm GeneratePlan introduced in Sect. 5.4.1 is executed to generate Reduce execution plan (line 6). At last, the algorithm scans the execution plan and assigns the OS-ELM models in the plan with correct *ReduceKeys* (lines 7–10). The *ReduceKey* is used to mark which Reduce task that intermediate results should pass to.

Algorithm 6: Map execution plan generation

```
Input: $models$ [ ] : array of OS-ELM models.
       $MapNum$ : the maximum number of Map tasks in the
cluster.
Result: $MapPlan < List < ID, start, end >>$[ ]: array
        represents Map execution plan, in which each element is
        a list of triples. In each triple $ID$ is the OS-ELM model
        ID, $start$ and $end$ are the start and end offsets of the
        training input file respectively.
for $m$ = 1 to sizeof($models$) do
    $TimeSum = TimeSum + model[i].EstimatedMapTime$;
$AvgTime = \frac{TimeSum}{MapNum}$;
for $m$= 1 to sizeof($models$) do
    if $models[m].EstimatedMapTime \le AvgTime * \alpha$ then
        list.add($< models[m].id$, 0 , $models[m].InputSize$,
        $models[m].EstimatedMapTime >$);
    else
        $splits = \frac{models[m].EstimatedMapTime}{AvgTime}$ ;
        $splitsize = \frac{models[m].InputSize}{splits}$;
        for $i$ =1 to $splits$ do
            list.add($< models[m].id$,
            $i * splitsize, (i+1) * splitsize$,
            $\frac{models[m].EstimatedMapTime}{splits}$ $>$);
$MapPlan$ = GeneratePlan($list.toArray()$, $MapNum$ ,
$AvgTime$ );
```

Figure 3 shows the execution procedure of BPOS-ELM. Each Map task is responsible for calculating **H** for multiple OS-ELM models, one OS-ELM model or part of one OS-ELM model according to the execution plan generated in Algorithm 6. Each Reduce task is responsible for calculating $\boldsymbol{\beta}$ for multiple OS-ELM models or one OS-ELM model according to the execution plan generated in Algorithm 8. The pseudo codes of BPOS-ELM job execution in Map phase and Reduce phase are shown in Algorithms 9 and 10 respectively.

5.5 Job execution

The pseudo codes of Map procedure are shown in Algorithm 9. The input is a Key-Value pair in which Key is the OS-ELM model ID and *value* represents data chunk (X_m, T_m). The algorithm first initializes the parameters such as w_i and

Algorithm 7: GeneratePlan()

```
Input: $Tasks < ID, start, end, time >$ [ ] : array of
       quadruples, in which each quadruple represents task
       information of OS-ELM model.
       $TaskNum$ : the maximum number of tasks that the
       cluster can hold.
       $AvgTime$ : the expected average execution time for each task.
Result: $Plan < List < ID, start, end >>$[ ] : array
        represents execution plan, in which each element is a list
        of triples. $ID$ is the OS-ELM model ID, $start$ and $end$
        are the start and end offsets of the training input file
        respectively, which are omitted for Reduce execution
        plan generation.
1  if $sizeof(Tasks) \le TaskNum$ then
2      for $m$ = 1 to sizeof($Tasks$) do
3          $Plan$[m].add( $< Tasks$[m].$id$, $Tasks$[m].$start$,
           $Tasks$[m].$end >$);
4  else
5      $Unassigned = Size$ = sizeof($Tasks$);
6      SortByTimeInDescendingOrder($Tasks$);
7      for $i$ =1 to $Size$ do
8          if $used[i] == true$ then
9              continue;
10         if $Count \ge TaskNum$ then
11             break;
12         addToPlan($Count, i$)
13         $Start = i$+1;
14         for $Start \le Size$ do
15             $Start$ = FindAndAdd($Start$);
16         $Count = Count$+1;
17     for $i$= 1 to $Size$ do
18         if $Unassigned$ >0 && $used[i]$==false then
19             $insertIndex$ = findMinTimeIndex($Time$);
20             AddToMapPlan($insertIndex, i$);
21 FindMinTimeIndex($Time$)
22     for $i$=1 to $Size$ do
23         if $Time[i] < MinTime$ then
24             $MinTime = Time[i]$;
25             $MinIndex = i$;
26     return $MinIndex$;
27 FindAndAdd($Start$)
28     for $j$=Start to Size do
29         if $used[j]$==false && Tasks[j].$time$+$Time[Count] \le$
           $AvgTime * \alpha$ then
30             addToMapPlan($Count, j$);
31             return $j$;
32     return $j$;
33 AddToPlan($P\_I, T\_I$)
34     $used[T\_I]$=$true$;
35     $Time[P\_I]$=$Time[P\_I]$+$Tasks[T\_I].time$
36     $Plan[P\_I]$.add(< $Tasks[T\_I].id$, $Tasks[T\_I].start$,
       $Tasks[T\_I].end$ >)
37     $Unassigned$ =$Unassigned$-1;
```

Algorithm 8: Reduce execution plan generation

```
Input: $models$ [ ] : array of OS-ELM models.
       $ReduceNum$ : the maximum number of Reduce tasks in
       the cluster.
Result: $ReducePlan < List < ID, start, end >>$[ ]: array
        represents Reduce execution plan, in which each element
        is a list of triples. In each triple, $ID$ is the OS-ELM ID,
        $start$ and $end$ are set to 0 to be compatible with
        GeneratePlan algorithm.
1  for $m$= 1 to sizeof($models$) do
2      TimeSum=TimeSum+model[i].EstimatedRedTime;
3  $AvgTime = \frac{TimeSum}{ReduceNum}$;
4  for $m$= 1 to sizeof($models$) do
5      list.add(< $models[m].id$, 0 , 0,
       $models[m].EstimatedRedTime$ >);
6  $ReducePlan$ = GeneratePlan($list.toArray()$, $ReduceNum$,
   $AvgTime$ );
7  for $i$=1 to sizeof(ReducePlan) do
8      for $j$=1 to sizeof (ReducePlan[i]) do
9          Index = FindByID(models, ReducePlan [i][j].ID);
10         models [Index].ReduceKey = i ;
11 FindByID($list, ID$)
12     for $i$=1 to sizeof(list) do
13         if $list[i].ID$ == ID then
14             return $i$;
```

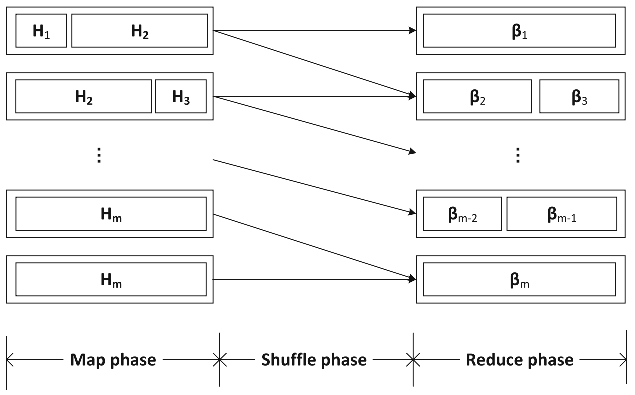


Fig. 3 Job execution of BPOS-ELM

b_i if they have not been initialized (lines 2–4). Then it collects $BLOCK_m$ training instances into a buffer $block_m$ (lines 5–6). After $BLOCK_m$ training instances are collected (line 7), matrix $\mathbf{H}_{m,k}$ is calculated according to Formula (2) (line 8) and $\mathbf{T}_{m,k}$ is also generated (line 9). After that, a key-value pair is generated as output (line 10). key is composed with OS-ELM model ID m, block ID k and $ReduceKey_m$ while $value$ includes $\mathbf{H}_{m,k}$ and $\mathbf{T}_{m,k}$. Finally, the counter is cleared (line 11) and the block ID k is increased by one (line 12).

Algorithm 9: BPOS-ELM map()

```
Input: (Key, Value): Key is the OS-ELM model ID, Value is a
       sample pair $(x_i, t_i) \in (X_k, T_k)$ where $0 \le i \le |(X_k, T_k)|$
       for the model.
Result: $m$ : OS-ELM model ID;
        $k$ : blockID;
        $ReduceKey_m$ : Key that marks which Reduce task trains
the model;
        $\mathbf{H}_{m,k}$ : Output weight;
        $\mathbf{T}_{m,k}$ : Observation value vector;
1  $m = Key$;
2  if $init_m$ ==false then
3      init(m);
4      $init_m = true$;
5  add to $block_m$;
6  $count_m$ ++;
7  if $count \ge BLOCK_m$ then
8      $\mathbf{H}_{m,k}$=calcH($block_m$);
9      $\mathbf{T}_{m,k}$=calcT($block_m$);
10     output(($m$,$k_m$,$ReduceKey_m$), ($\mathbf{H}_{m,k}$, $\mathbf{T}_{m,k}$));
11     $count_m = 0$;
12     $k_m$++;
```

Algorithm 10: BPOS-ELM reduce()

```
Input: Set of $(key, value)$: $key$ is a combination of OS-ELM
       model ID $m$, blockID $k$ and $ReduceKey$. $value$ is a
       vector pair $(H_{kb}, T_{kb})$;
Result: $\beta_m$: output weight vector (corresponding to $\beta_{m,k}$).
1  $m = getm(key)$;
2  if $firstRun_m$==true then
3      init($m$);
4      $firstRun_m$=false;
5  $\mathbf{H}_{m,k+1} = getH(value)$;
6  $\mathbf{T}_{m,k+1} = getT(value)$;
7  $\mathbf{P}_{m,k+1} =$
   $\mathbf{P}_{m,k} - \mathbf{P}_{m,k}\mathbf{H}_{m,k+1}^T(\mathbf{I} + \mathbf{H}_{m,k+1}\mathbf{P}_{m,k}\mathbf{H}_{m,k+1}^T)^{-1}\mathbf{H}_{m,k+1}\mathbf{P}_{m,k}$;
8  $\beta_{m,k+1} = \beta_{m,k} + \mathbf{P}_{m,k+1}\mathbf{H}_{m,k+1}^T(\mathbf{T}_{m,k+1} - \mathbf{H}_{m,k+1}\beta_{m,k})$;
```

The pseudo codes of Reduce procedure are shown in Algorithm 10. The output results of Map phase which have the same $ReduceKey$ are partitioned to the same Reducer and then sorted by m and k. When the set of key-value pairs are transferred to Reduce procedure, the OS-ELM model ID m is first resolved (line 1). The parameters for OS-ELM model m are initialized if they have not been initialized (lines 2–4) and then $\mathbf{H}_{m,k}$ and $\mathbf{T}_{m,k}$ included in $value$ are resolved (lines 5–6). Finally, the $\mathbf{P}_{m,k}$ and $\beta_{m,k}$ are updated according to the formulas (lines 7–8).

5.6 Execution information collection

After job completes execution, the execution information of selected tasks is collected and merged to historical execution statistics. The execution information of the tasks that process one OS-ELM model is collected and merged to historical statistics. The information is helpful to further improve the execution time estimation accuracies of Map tasks and Reduce tasks.

6 Experimental evaluation

The setup of evaluations is firstly introduced in Sect. 6.1. Then the estimation algorithms of BPOS-ELM are evaluated in Sect. 6.2. Section 6.3 evaluates the accuracy of BPOS-ELM with real data. The training speed evaluation with real and synthetic data is introduced in Sect. 6.4. Finally, the scalability of BPOS-ELM is evaluated with synthetic data in Sect. 6.5.

6.1 Experimental setup

POS-ELM indicates parallel online sequential learning machine algorithm in [13] that trains each OS-ELM model one by one. BPOS-ELM is compared with POS-ELM and OS-ELM algorithms. All the three algorithms are implemented in Java 1.6. Universal java matrix package (UJMP) [20] with version 0.2.5 is used for matrix storage and processing. The activation function of OS-ELM, POS-ELM and BPOS-ELM algorithm is $g(x) = \dfrac{1}{1+e^{-x}}$.

The size of the memory that is used to train an OS-ELM model increases as the number of nodes in hidden layer increases. So is the size of the memory on the number of attributes. In accuracy evaluation, because the number of attributes is not very large (maximum 780), the number of hidden nodes is set to 128 in our experimental environment. In training speed evaluation and scalability evaluation, because synthetic data with maximum number of attributes 1024 are used, only training speed and scalability are evaluated, the number of hidden nodes is set to 64 in our environment.

Hadoop-0.20.2-cdh3u3 is used as our evaluation platform. The Hadoop cluster is deployed on 9 commodity PCs in a high speed Gigabit network, with one PC as the Master node and the others as the Slave nodes. Each PC has an Intel Quad Core 2.66 GHZ CPU, 4 GB memory and CentOS Linux 5.6 operating system. Each PC is set to hold maximum 4 Map or Reduce tasks running in parallel and the cluster is set to hold maximum 32 tasks running in parallel. Each task is configured with 1024M java heap. Other parameters are using the default values of Hadoop.

BPOS-ELM algorithm is evaluated with real data and synthetic data. The real data sets (MNIST[1], DNA[1], and KDD-Cup99[2]) are mainly used to evaluate training accuracy and

[1] Downloaded from http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[2] Downloaded from http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

Table 3 Specifications of real data

Data set	#attributes	#class	#training data	#testing data	Size of test data (KB)
MNIST	780	10	60,000	10,000	176001.138
DNA	180	3	2000	1186	1126.301
KDDCup99	41	2	4,898,431	292,300	708197.916

Table 4 Specifications of synthetic data and running parameters for scalability evaluation

Parameter	Value range	Default value
Data set type	F (Flower), C (CIFAR-10)	F
#training data	10k, 20k, 40k, 80k, 160k,320k, 640k, 1280k, 2560k, 5120k, 10240k	640k
#attributes	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024	64
#cores	1, 2, 4, 8, 16, 32	32
#data per block	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024	64
#neurons	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024	64
#classifications	1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024	2
#model groups	3, 6, 9, 12, 15, 18, 21, 24, 27, 30	6

testing accuracy. Some attributes of KDDCup99 data set are symbolic-valued attributes which cannot be directly used for BPOS-ELM, POS-ELM or OS-ELM, so we preprocess the data set by mapping symbolic-valued attributes to numeric-valued attributes with the method in [10]. For testing data, we use the KDDcup99 (corrected) evaluation data set by excluding those attack instances which do not belong to the set of attack types in the training data set. The specifications of real data are shown in Table 3.

The synthetic data sets are used for training speed evaluation and scalability evaluation. Two kinds of synthetic data are used, which are generated based on Flower[3] and CIFAR-10[4] respectively. For the data set that generates based on Flower, the attributes and volume are extended by duplicating the original data in a round-robin way. For the data set that generates based on CIFAR-10, the attributes of synthetic data are extracted from the original data and the volume is extended by duplicating the original data in a round-robin way. In one training model group, there are 11 OS-ELM models training with synthetic data sets with N varies from $2^0 \times 10^4$ to $2^{10} \times 10^4$. The parameters used in scalability evaluation are summarized in Table 4. In the experiments, all the parameters use default values unless otherwise specified.

6.2 Evaluation of execution time estimation

Since the time estimation accuracy is the basis of execution plan generation, the accuracy of time estimation is first evaluated. The Map execution time and Reduce execution time are estimated with methods introduced in Sect. 5.3 and then compared with the actual execution time. The estimation time is evaluated with different B, C, D, $\tilde{N}$ and N.

Figure 4 shows the evaluation of Map execution time estimation with different parameters. Compared with regression method, the IDW method has higher Map estimation accuracy. The reason for this is that noises are superimposed on Map execution time due to the local/remote data accesses that generated in Map phase. These noises reduce the regression feature of Map execution time statistics. Because the IDW method does not use regression model, it avoids this problem. It can be found from Fig. 4 that the estimated Map execution time is almost the same with the actual execution time. This shows that IDW and k nearest neighbour methods are effective to estimate Map execution time with historical statistics.

Figure 5 shows the evaluation of Reduce execution time estimation. It can be found from Fig. 5 that the estimated Reduce execution time is almost the same with the actual execution time. This shows that it is effective to use complexity analysis and regression model to estimate Reduce execution time with historical statistics.

The high accuracy of execution time estimation also benefits from the execution information collected from the actual BPOS-ELM tasks. The accurate execution time estimation lays the foundation for the execution plan generation.

6.3 Accuracy evaluation

We use one MapReduce job to train three OS-ELM models with DNA, MNIST and KDDCup99 data sets using BPOS-

[3] Downloaded from http://www.datatang.com/data/13152.

[4] Downloaded from https://www.cs.toronto.edu/~kriz/cifar.html.

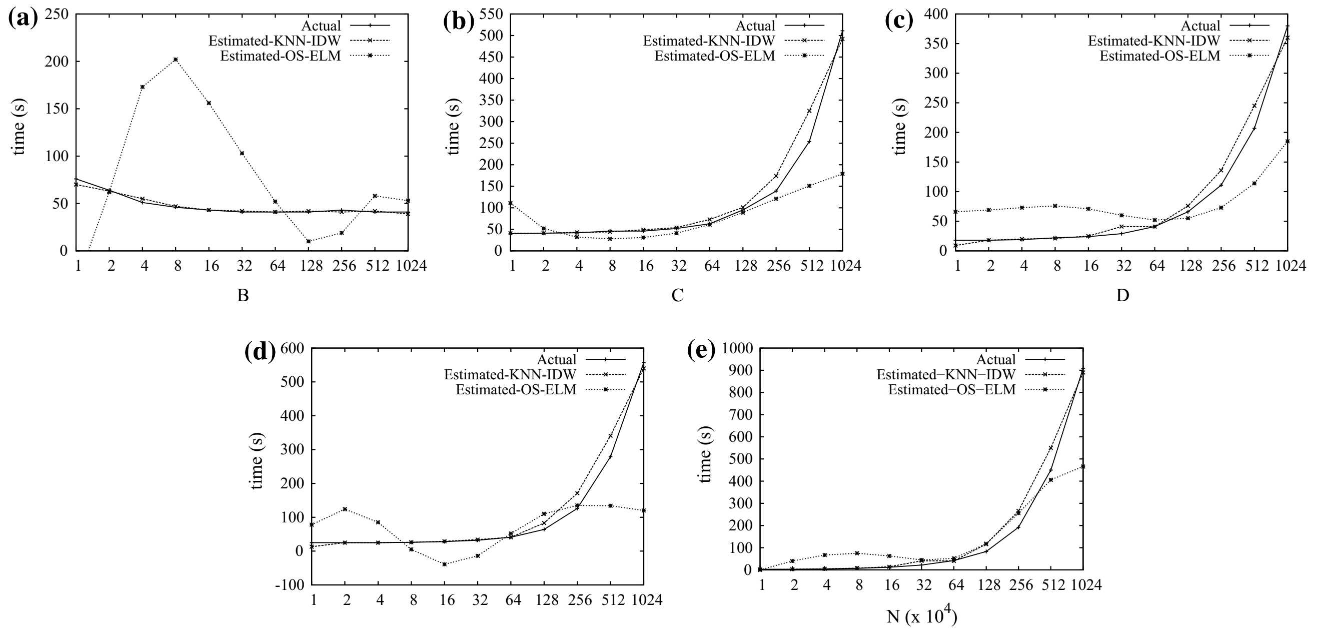


Fig. 4 Evaluation of Map execution time estimation

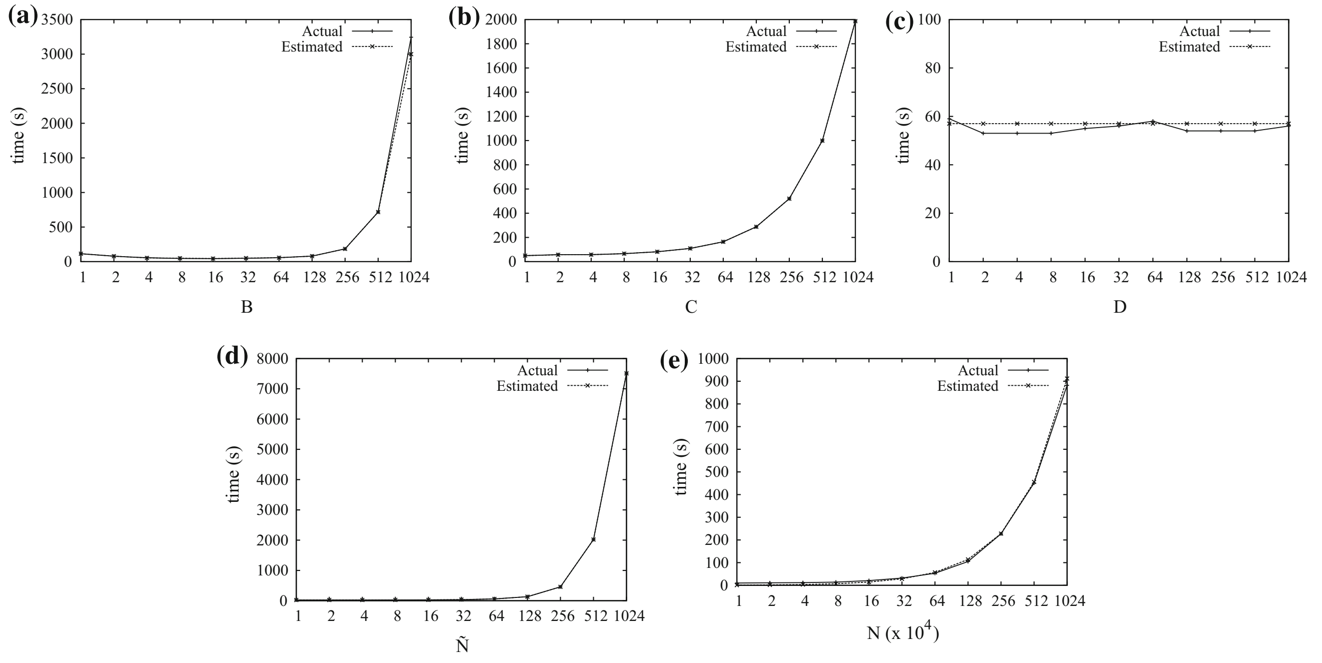


Fig. 5 Evaluation of Reduce execution time estimation

ELM. We also train the three models using OS-ELM and POS-ELM algorithm one by one as comparisons. Table 5 shows the results of accuracy evaluations with real data. It can be found that the training accuracy and testing accuracy of BPOS-ELM algorithm are at the same level with those of POS-ELM and OS-ELM. The reason for this is that BPOS-ELM algorithm does not change the computation sequence of matrices calculation of OS-ELM.

6.4 Training speed evaluation

Table 6 shows the results of training speed evaluation with real data and synthetic data. The training time in Table 6 includes the time of training multiple OS-ELM models with different algorithms. The training time of BPOS-ELM includes time of ID assignation, execution time estimation, execution plan generation and job execution. The training

Table 5 Accuracy evaluation with real data

Data set	Algorithm	Training accuracy	Testing accuracy
MNIST	OS-ELM	0.824	0.831
	POS-ELM	0.823	0.830
	BPOS-ELM	0.825	0.831
DNA	OS-ELM	0.845	0.779
	POS-ELM	0.846	0.781
	BPOS-ELM	0.844	0.780
KDDCup99	OS-ELM	0.992	0.856
	POS-ELM	0.991	0.856
	BPOS-ELM	0.992	0.855

Table 6 Execution time evaluation with real data and synthetic data

Data set	Algorithm	Training time (s)
Real	OS-ELM	1146
	POS-ELM	1025
	BPOS-ELM	1021
Synthetic (F)	OS-ELM	20701
	POS-ELM	10651
	BPOS-ELM	2130
Synthetic (C)	OS-ELM	20105
	POS-ELM	10501
	BPOS-ELM	2103

time of OS-ELM and POS-ELM includes time of training multiple OS-ELM models one by one.

As shown in Table 6, the training speed of BPOS-ELM is faster than the training speed of POS-ELM and OS-ELM. For the models training with real data sets, the training speed of BPOS-ELM is only a little faster than that of POS-ELM. The reason for this is that most of the cores are idle in Reduce phase of BPOS-ELM since the number of the training models is less than that of cores and the Reduce tasks are indivisible. For the models training with synthetic data sets, the training speed of BPOS-ELM is much faster than that of POS-ELM and OS-ELM. This is because the cores of the cluster are fully utilized in the Reduce phase of BPOS-ELM algorithm. This result also shows that BPOS-ELM trains large scale multiple OS-ELM models efficiently.

The results in Table 6 also reveal that BPOS-ELM has the same training speed when training with Flower and CIFAR-10 based data sets. This is because the costs of calculation and job execution are almost the same to process these two data sets.

6.5 Scalability evaluation

BPOS-ELM algorithm is also evaluated by the speedup. Speedup a metric for improvement in performance between

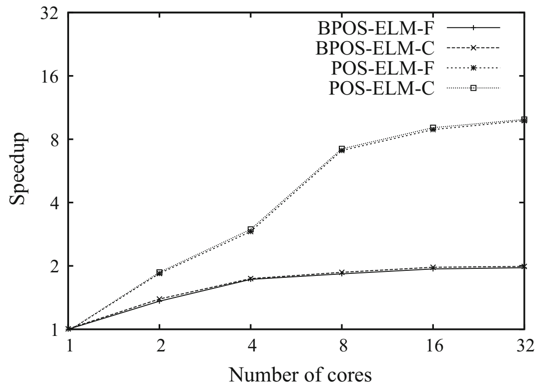


Fig. 6 Speedup with different number of cores

two systems processing the same problem. The speedup of parallel algorithm is defined in Formula (9).

$$Speedup = \frac{Time\ of\ one\ core}{Time\ of\ N\ cores} \tag{9}$$

Figure 6 shows the scalability (speedup) of BPOS-ELM compared with that of POS-ELM. Since OS-ELM does not support training in parallel, we do not compare BPOS-ELM with it here. The speedup of BPOS-ELM reaches 10× when the number of cores increases to 32. It means that BPOS-ELM has good scalability. It benefits from accurate estimations of Map and Reduce execution time and the execution plan which is suitable for parallel processing. It can also be found that the speedup of BPOS-ELM reaches 10× whereas the speedup of POS-ELM only reaches 1.96×. The reason is that BPOS-ELM calculates $\boldsymbol{\beta}_{m,k}$ for different models in parallel instead of calculating them sequentially. It is shown in Fig. 6 that the speedups of BPOS-ELM which train with different data sets are at the same level. It also shows that data set type has little effect on BPOS-ELM algorithm.

There are several reasons for the changing trend of speedup decreases as the number of cores increases. First, since the Reduce tasks cannot be further split into smaller ones, the execution time of the OS-ELM model which has the longest Reduce execution time does not decrease as the number of cores increases. In this case, the MapReduce job has to wait for the completion of the slowest task. Second, the cost of scheduling tasks among multiple cores increases as the number of cores increases. Third, the memory and the number of I/Os become bottlenecks as the number of cores increases since all the Map tasks and Reduce tasks running on a physical machine share the same memory and disks.

Figure 7 shows the training time of BPOS-ELM compared with that of POS-ELM. The training time of BPOS-ELM is a little longer than that of POS-ELM on one core due to the overhead derived from task scheduling. However, as the number of cores increases, the training time drops signifi-

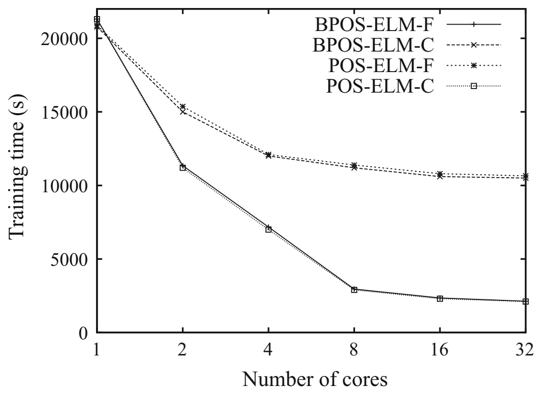


Fig. 7 Training time with different number of cores

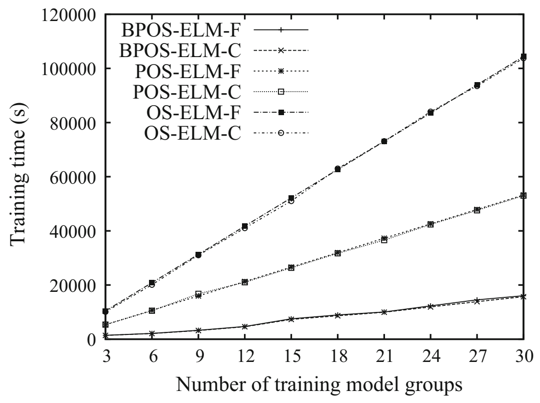


Fig. 8 Training time with different number of model groups

cantly and becomes much shorter than that of POS-ELM. It means that BPOS-ELM is more efficient than POS-ELM for training multiple models for the reason that BPOS-ELM trains multiple models in parallel in Reduce phase.

Figure 8 shows the training time of BPOS-ELM with different number of model groups compared with that of POS-ELM and OS-ELM. As shown in Fig. 8, the training time increases much more slowly than that of POS-ELM and OS-ELM. The reason for this is that BPOS-ELM trains multiple OS-ELM models in parallel in both Map phase and Reduce phase whereas POS-ELM only parallelizes the training in Map phase and OS-ELM does not parallelize the training. It means that BPOS-ELM utilizes computing resources efficiently.

7 Conclusions

More and more models need to be trained in this era of big data, and it has become a challenging problem to train multiple models efficiently. In this work, we proposed an efficient parallel method for batched online sequential extreme learning machine (BPOS-ELM) training using MapReduce. The execution time of Map and Reduce tasks was estimated with historical statistics. We proposed two methods to estimate Map execution time, regression method and inverse distance weighted interpolation method. We estimated Reduce execution time based on complexity analysis and regression techniques. A Map execution plan and a Reduce execution plan were generated with greedy strategy based on the estimations. A MapReduce job was launched to train multiple OS-ELM models according to the execution plans. The algorithm also collected information of selected tasks in the job and merged it to historical statistics to help to improve the estimation accuracy. BPOS-ELM algorithm was evaluated with real and synthetic data. The experimental results showed that the accuracy of BPOS-ELM was at the same level as those of POS-ELM and OS-ELM. The speedup of BPOS-ELM reached 10× on a cluster with maximum 32 cores. Compared with OS-ELM and POS-ELM, BPOS-ELM trains multiple OS-ELM models more efficiently.

Acknowledgements This research was partially supported by the National Natural Science Foundation of China under Grant Nos. 61173030, 61272181, 61272182, 61173029, 61332014; and the Fundamental Research Funds for the Central Universities (N120816001).

References

1. Amazon elastic compute cloud (2015). http://aws.amazon.com/cn/ec2/
2. Huang GB, Zhu QY, Siew CK (2004) Extreme learning machine. In: Technical Report ICIS/03/2004. School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore
3. Huang GB, Chen L (2007) Convex incremental extreme learning machine. Neurocomputing 70(16):3056–3062
4. Huang GB, Zhou H, Ding X, Zhang R (2012) Extreme learning machine for regression and multiclass classification. IEEE Trans Syst Man Cybern Part B Cybern 42(2):513–529
5. Liang NY, Huang GB, Saratchandran P, Sundararajan N (2006) A fast and accurate online sequential learning algorithm for feedforward networks. Neural Netw IEEE Trans 17(6):1411–1423
6. Dean J, Ghemawat S (2008) MapReduce: Simplified data processing on large clusters. Commun ACM 51(1):107–113
7. White T (2012) Hadoop: The definitive guide. O'ReillyMedia Inc, Sebastopol, CA, USA
8. Lin J, Yin J, Cai Z, Liu Q, Li K, Leung V (2013) A secure and practical mechanism for outsourcing ELMs in cloud computing. IEEE Intell Syst 28(6):35–38
9. He Q, Shang T, Zhuang F, Shi Z (2013) Parallel extreme learning machine for regression based on MapReduce. Neurocomputing 102:52–58
10. Xiang J, Westerlund M, Sovilj D, Pulkkis G (2014) Using extreme learning machine for intrusion detection in a big data environment. In: Proceedings of the 2014 workshop on artificial intelligent and security workshop, AISec '14ACM, New York, pp 73–82
11. Xin J, Wang Z, Chen C, Ding L, Wang G, Zhao Y (2013) ELM*: distributed extreme learning machine with MapReduce. World Wide Web, pp. 1–16
12. van Heeswijk M, Miche Y, Oja E, Lendasse A (2011) GPU-accelerated and parallelized ELM ensembles for large-scale regression. Neurocomputing 74(16):2430–2437

13. Wang B, Huang S, Qiu J, Liu Y, Wang G (2015) Parallel online sequential extreme learning machine based on MapReduce. Neurocomputing 149, Part A:224–232
14. Cao J, Lin Z (2015) Extreme learning machines on high dimensional and large data applications: a survey. Math Probl Eng 2015:1–12
15. NVIDIA CUDA home page (2015). http://www.nvidia.com/object/cuda_home_new.html
16. Matlab parallel computing toolbox (2015). http://www.mathworks.com/products/parallel-computing/index.html
17. Huang GB, Zhu QY, Siew CK (2006) Extreme learning machine: theory and applications. Neurocomputing 70(1):489–501
18. Garey MR, Johnson DS (1990) Computers and intractability; a guide to the theory of NP-completeness. W. H. Freeman & Co, New York
19. Shepard D (1968) A two-dimensional interpolation function for irregularly-spaced data. In: Proceedings of the 1968 23rd ACM national conference, ACM '68ACM, New York, pp 517–524
20. Arndt H, Bundschus M, Naegele A (2009) Towards a next-generation matrix library for java. In: Computer software and applications conference, 2009. COMPSAC'09. 33rd Annual IEEE International, vol 1. IEEE, pp 460–467