Neurocomputing 102 (2013) 52–58

Contents lists available at SciVerse ScienceDirect

Neurocomputing

journal homepage: www.elsevier.com/locate/neucom

Parallel extreme learning machine for regression based on MapReduce

Qing He [a], Tianfeng Shang [a,b,*], Fuzhen Zhuang [a], Zhongzhi Shi [a]

[a] The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
[b] Graduate School of Chinese Academy of Sciences, Beijing 100049, China

ARTICLE INFO

Available online 5 June 2012

Keywords:
Data mining
Regression
ELM
MapReduce
PELM

ABSTRACT

Regression is one of the most basic problems in data mining. For regression problem, extreme learning machine (ELM) can get better generalization performance at a much faster learning speed. However, the enlarging volume of datasets makes regression by ELM on very large scale datasets a challenging task. Through analyzing the mechanism of ELM algorithm, an efficient parallel ELM for regression is designed and implemented based on MapReduce framework, which is a simple but powerful parallel programming technique currently. The experimental results demonstrate that the proposed parallel ELM for regression can efficiently handle very large datasets on commodity hardware with a good performance on different evaluation criterions, including speedup, scaleup and sizeup.

1. Introduction

With the rapid development of information technology, data mining has become one of the popular techniques for hidden knowledge discovery, which is helpful to decision making. Extreme learning machine (ELM) as emergent technology is one of the most important algorithms for classification or regression in data mining [1,2], which is one category of artificial neural networks. Many variants of ELM have been proposed in practical applications, such as basic ELM [3,4], random hidden layer feature mapping based ELM [5], incremental ELM [6,7] and kernel based ELM [8,9], etc.

ELM has been proposed by Huang in [3], which works for generalized single-hidden layer feed forward networks (SLFNs). However, in ELM, the hidden layer of SLFNs need not be iteratively tuned. Compared with those traditional computational intelligence techniques, ELM can provide better generalization performance at a much faster learning speed and with least human intervenes.

However, similar with most of the data mining algorithms, ELM algorithm is also memory resident. That is to say, all the data to process must be loaded into computer memory in advance. So it is a challenge to process large scale datasets, and it drives the increasing research of parallel algorithm and distributed processing.

The MapReduce model is introduced by Google as a software framework for parallel computing in a distributed environment [10–12]. Users specify the computation in terms of a map and a reduce function, and the underlying runtime system automatically parallelizes the computation across large-scale clusters of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks. Google and Hadoop both provide MapReduce running time with fault tolerance and dynamic flexibility support. Besides, Hadoop can be easily deployed on commodity hardware.

In previous work, some parallel algorithms have been implemented based on MapReduce framework [13,14], such as parallel ETL algorithms [15], parallel classification Each parallel algorithm [16,17] could get a good performance for different problems. However, for regression problems, there is no effective parallel regression algorithm for large datasets. Based on above analysis, we design the proper $\langle key, value \rangle$ pairs for ELM algorithm in this paper, and then parallel implementation of ELM for regression is achieved on MapReduce framework to mine large-scale datasets. Based on datasets of some practical regression problems, the experimental results demonstrate that our proposed parallel ELM for regression is effective and efficient for dealing with large-scale datasets.

The rest of the paper is organized as follows. Section 2 introduces MapReduce. Section 3 gives out some important theories of ELM. Parallel implementation of ELM algorithm based on MapReduce is proposed in Section 4. Experimental results and evaluations are showed in Section 5 with respect to speedup, scaleup and sizeup. Finally, Section 6 presents conclusions and future work.

2. MapReduce review

MapReduce is a programming model for data processing [13,14], as shown in Fig. 1. In MapReduce model, the underlying

* Corresponding author at: The Key Laboratory of Intelligent Information Processing, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.
E-mail address: shangtf@ics.ict.ac.cn (T. Shang).

0925-2312/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.neucom.2012.01.040

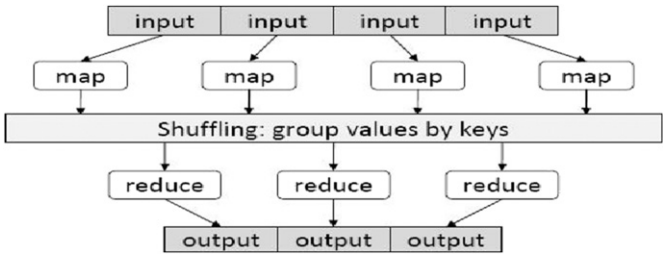

Fig. 1. Illustration of the MapReduce framework: the "map" is applied to all input records, which generates intermediate results that are aggregated by the "reduce".

runtime system automatically parallelizes the computation across large-scale cluster of machines, handles machine failures, and schedules inter-machine communication to make efficient use of the network and disks.

Generally, the computation in one MapReduce job contains two main phases, named a map function and a reduce function, respectively. As shown in Fig. 1, in the first phase, the input data are processed by the map function, generating some intermediate results as the input of the reduce function in the second phase. What users just need to do is to design the computation in the map function and the reduce function, they need not care about communication between the map function and the reduce function.

More specifically, the $\langle key,value\rangle$ pair is the basic data structure in MapReduce, and all data processed in MapReduce are in terms of $\langle key,value\rangle$ pairs. In the first phase, for each input $\langle key,value\rangle$ pair, a map function is invoked once, and it generates some intermediate pairs. Before the execution of a reduce function, these intermediate pairs are shuffled by the underlying system, including merge, group and sort values by keys. In the second phase, a reduce function is called once for each shuffled intermediate pair, generating output $\langle key,value\rangle$ pairs as final results.

To be noted, the type of output $\langle key,value\rangle$ pair of a map function or a reduce function may be different from the type of its input $\langle key,value\rangle$ pair, but in one MapReduce job, the type of input $\langle key,value\rangle$ pair of a reduce function must be the same with the type of output $\langle key,value\rangle$ pair of the map function, which can be represented as follows:

$$map :: (key_1,value_1) \Rightarrow list(key_2,value_2) \tag{1}$$

$$reduce :: (key_2,list(value_2)) \Rightarrow (key_2,value_3) \tag{2}$$

The MapReduce runtime system executes each map function independently, that is, all the map functions can be fully parallelized, so does the reduce function. Thus, the MapReduce model provides sufficient high-level parallelization. Due to the highly transparency of MapReduce, its parallelism can be easily understood in actual programming, especially for novice programmers.

3. Extreme learning machine for multiple regression

ELMs are first articulated by Huang in his 2004 paper [3]. They are originally derived from the single-hidden layer feedforward neural networks (SLFNs) and then extended to the "generalized" SLFNs. The essence of ELM is that: if the feature mapping of hidden layer satisfies the universal approximation condition, the hidden layer of SLFNs need not be iteratively tuned, which is different from the common SLFNs. One of typical implementation of ELMs is to generate the weight matrix randomly between the input layer and the hidden layer, and then calculate the output weights by the least-square method.

It is worth mentioning that ELMs not only tend to reach the smallest training error but also the smallest norm of output weights. According to the neural network theory, for feedforward neural networks, the smaller norm of weights are, the better generalization performance the networks tend to have (Fig. 2).

SLFN network functions with L hidden nodes can be represented by

$$f_L(\mathbf{x}) = \sum_{i=1}^{L} \beta_i g_i(\mathbf{x}) = \sum_{i=1}^{L} \beta_i G(\mathbf{a}_i,b_i,\mathbf{x}), \mathbf{x} \in \mathbf{R}^d, \beta_i \in \mathbf{R}^m \tag{3}$$

where g_i denotes the output function $G(\mathbf{a}_i,b_i,\mathbf{x})$ of the ith hidden node. For N arbitrary distinct samples $(\mathbf{x}_i,\mathbf{t}_i) \in \mathbf{R}^d \times \mathbf{R}^m$, SLFNs with L hidden nodes can approximate these N samples with zero error means that there exist $(\mathbf{a}_i,b_i)$ and β_i such that

$$\sum_{i=1}^{L} \beta_i G(\mathbf{a}_i,b_i,\mathbf{x}_j) = \mathbf{t}_j, \quad j = 1,\ldots,N \tag{4}$$

The above N equations can be written compactly as

$$\mathbf{H}\beta = \mathbf{T} \tag{5}$$

where

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}(\mathbf{x}_1) \\ \vdots \\ \mathbf{h}(\mathbf{x}_N) \end{bmatrix} = \begin{bmatrix} G(\mathbf{a}_1,b_1,\mathbf{x}_1) & \cdots & G(\mathbf{a}_L,b_L,\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ G(\mathbf{a}_1,b_1,\mathbf{x}_N) & \cdots & G(\mathbf{a}_L,b_L,\mathbf{x}_N) \end{bmatrix}_{N\times L}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_L^T \end{bmatrix}_{L\times m} \quad \text{and} \quad \mathbf{T} = \begin{bmatrix} \mathbf{t}_1^T \\ \vdots \\ \mathbf{t}_N^T \end{bmatrix}_{N\times m}$$

The above equations can be seen as a multiple regression system, where $\mathbf{H}$ is the independent variables matrix of N samples, β is regression coefficient vector to be solved, and $\mathbf{T}$ is sample observation value matrix. The smallest norm least-squares solution of the above multiple regression system is:

$$\widehat{\beta} = \mathbf{H}^{\dagger}\mathbf{T} \tag{6}$$

where $\mathbf{H}^{\dagger}$ is the Moore–Penrose generalized inverse of matrix $\mathbf{H}$. So the regression estimation value matrix $\mathbf{Y}$ can be expressed by

$$\mathbf{Y} = \mathbf{H}\widehat{\beta} = \mathbf{H}\mathbf{H}^{\dagger}\mathbf{T} \tag{7}$$

The error matrix of the multiple regression system is

$$\|\mathbf{e}\|^2 = \|\mathbf{Y}-\mathbf{T}\|^2 = \|\mathbf{H}\mathbf{H}^{\dagger}\mathbf{T}-\mathbf{T}\|^2 \tag{8}$$

To solve the multiple regression system, the key point is to solve the matrix $\mathbf{H}^{\dagger}$. Generally, there are several methods to solve the Moore–Penrose generalized inverse of a matrix, including orthogonal projection method, orthogonalization method, iterative

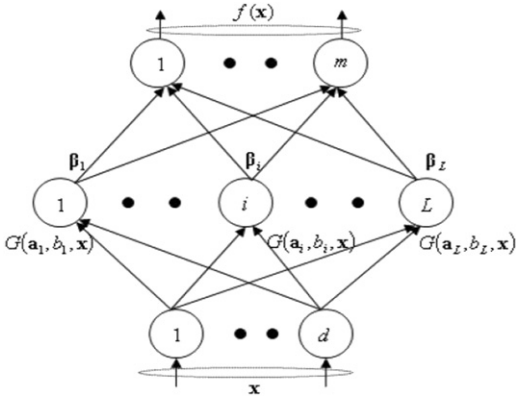

Fig. 2. Illustration of SLFN.

method, and singular value decomposition (SVD) method [18,19]. In this paper, for the convenience of programming, we adopt SVD method to calculate the Moore–Penrose generalized inverse of matrix $\mathbf{H}$.

However, when the number of samples N is large, the $N \times L$ dimension matrix $\mathbf{H}$ cannot be loaded into computer memory, the above multiple regression system cannot be solved in the memory-resident mode. In this situation, development of disk-resident parallel ELM algorithm for the multiple regression system becomes a necessary and significant research.

4. Parallel extreme learning machine (PELM) for regression based on MapReduce

As discussed in Section 3, ELM algorithm can be described in Algorithm 1.

Algorithm 1. ELM.

Input: a training set $\aleph = \{(\mathbf{x}_i, \mathbf{t}_i) | \mathbf{x}_i \in \mathbf{R}^d, \mathbf{t}_i \in \mathbf{R}^m, i = 1, \ldots, N\}$, hidden node number L and hidden node transfer function $G(\mathbf{a}_j, b_j, \mathbf{x}), j = 1, \cdots, L$.
Output: a trained instance of ELM.
(1) Randomly generate hidden node parameters $(\mathbf{a}_j, b_j), \mathbf{a}_j \in \mathbf{R}^d, b_j \in \mathbf{R}$;
(2) Calculate hidden node output matrix $\mathbf{H}$;
(3) Calculate output weight vector $\widehat{\beta} = \mathbf{H}^{\dagger}\mathbf{T}$.

In Algorithm 1, Step (1) is to generate a $d \times L$ dimension matrix, which can usually be completed within a very short time. Step (2) is actually a matrix multiplication, which can be transferred into parallel mode easily. Step (3) is mainly to calculate the Moore–Penrose generalized inverse of matrix $\mathbf{H}$, which can be achieved through a series of matrix operations, including the multiplication of matrix $\mathbf{H}$ and its transposed matrix $\mathbf{H}^T$, the calculation of matrix eigenvalues and eigenvectors by Jacobi and the calculation of Moore–Penrose generalized inverse by SVD.

Note that, $\mathbf{H}$ is a $N \times L$ dimension matrix, thus $\mathbf{H} \times \mathbf{H}^T$ is a $N \times N$ dimension matrix. However, in massive data mining, N is usually a very large number, so $\mathbf{H} \times \mathbf{H}^T$ will be a too large matrix for memory, which also makes it impossible to execute ELM algorithm in the way of memory-residence. However, in most cases, the number of hidden nodes is much less than the number of training samples, $L \ll N$. According to matrix theory, in SVD method, a small matrix $\mathbf{H}^T \times \mathbf{H}$ could be calculated instead of the large matrix $\mathbf{H} \times \mathbf{H}^T$. Thus, $\mathbf{H}^{\dagger} = inv(\mathbf{H}^T \times \mathbf{H}) \times \mathbf{H}^T$, and $\widehat{\beta} = inv(\mathbf{H}^T \times \mathbf{H}) \times \mathbf{H}^T \times \mathbf{T}$. Hence, the bottleneck operations $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$ need to be parallelized using MapReduce. The remaining matrix multiplication in $inv()$ are performed sequentially.

As the analysis above, there need to be two MapReduce jobs in parallel ELM algorithm: calculation of hidden node output matrix $\mathbf{H}$ and calculation of matrix $\mathbf{H}^T \times \mathbf{H}$ and matrix $\mathbf{H}^T \times \mathbf{T}$. The key to the parallel implementation is the design of $\langle key, value \rangle$ pairs, including the input $\langle key, value \rangle$ pair of map function, the intermediate $\langle key, value \rangle$ pair (The input $\langle key, value \rangle$ pair of reduce function is the same with the output $\langle key, value \rangle$ pair of map function, we refer to them as the intermediate $\langle key, value \rangle$ pair.) and the output $\langle key, value \rangle$ pair of reduce function. Then the $\langle key, value \rangle$ pairs of the two MapReduce jobs are analyzed and designed in Sections 4.1 and 4.2, respectively.

4.1. MapReduce job on hidden layer mapping

MapReduce job on hidden layer mapping is just mapping samples to space represented by hidden layer nodes. In the beginning, the input dataset is stored on Hadoop Distributed File System (HDFS) as a sequence file of $\langle key, value \rangle$ pairs, each of which represents a sample in the dataset, so the input key of map function is the offset in bytes of each sample to the start point of the data file and the input value of map function is the content of the sample, including the value of independent and dependent variables. In the map function, samples are randomly assigned to each mapper, so the original samples should be saved together with its hidden node output results, thus the intermediate key is the hidden node output vector of the sample and the intermediate value is the content of the sample. Algorithm 2 gives the pseudo code of the map function of MapReduce job on hidden layer mapping.

Algorithm 2. hiddenMapper(key, value).

Input: $\langle key, value \rangle$ pair, where *key* is the offset in bytes, and *value* is the content of a sample.
Output: $\langle key', value' \rangle$ pair, where key' is the hidden node output vector, and $value'$ is the content of sample.
(1) Parse the string *value* to an array, named *onesample*;
(2) Normalize the independent variables of each input sample data;
(3) Initiate string *outkey* as a null string;
(4) for $i=1$ to hidden node number
(5) $\mathbf{x} = 0$;
(6) for $j=1$ to *onesample*.length
(7) $\mathbf{x} += \mathbf{A}[i][j] \times onesample[j]$; ($\mathbf{A}[i][j]$ is the weight between the j^{th} input node and the i^{th} hidden node.)
(8) endfor
(9) *outkey*.append($G(\mathbf{a}_i, b_i, \mathbf{x})$); ($\mathbf{a}_i$ is the i^{th} row of $\mathbf{A}$.)
(10) *outkey*.append(“,”);
(11) endfor
(12) output(*outkey*, *value*);

In Algorithm 2, Steps (1)–(3) are some preparation for the calculation, including parsing the input *value*, normalizing the independent variables and initiating the intermediate *key* string, *outkey*. Steps (4)–(11) calculate the hidden layer output vector, and append all its elements to string *outkey*, spaced with “,”. Step (12) is to output the intermediate $\langle key, value \rangle$ pair, $\langle outkey, value \rangle$.

4.2. MapReduce job on matrix generalized inverse

MapReduce job on matrix generalized inverse is used to calculate the matrix multiplication $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$ in parallel. So the input key of map function is the offset in bytes of each sample to the start point of the data file and the input value of map function is a string containing the hidden node output vector of each sample and the original sample. In map function, the elements of hidden node output vector are firstly parsed, and then they multiply each other together to form the intermediate results. Besides, the dependent variable is also multiplied with the corresponding element of hidden node output vector. Thus the intermediate key could be a random string for the key will not use in the following reduce function and the intermediate value could be a string of the intermediate results together with multiplication results of the dependent variable. In reduce function, the intermediate results are merged, sorted and summed to achieve the matrix multiplication $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$, so the output key of reduce function could be a string containing all the elements in matrix $\mathbf{H}^T \times \mathbf{H}$ and $\mathbf{H}^T \times \mathbf{T}$ and the output value of reduce function could be any string you like. Algorithms 3 and 4 give the pseudo code of the map and reduce functions of MapReduce job on matrix generalized inverse, where for N samples dataset, each term of $\mathbf{X} = \mathbf{H}^T \times \mathbf{H}$ and $\mathbf{Y} = \mathbf{H}^T \times \mathbf{T}$ can be

expressed as follows:

$$\mathbf{X}[i][j] = \sum_{k=1}^{N} H[k][i] \times H[k][j] \tag{9}$$

$$\mathbf{Y}[i][j] = \sum_{k=1}^{N} H[k][i] \times T[k][j] \tag{10}$$

Algorithm 3. HTHMapper(key, value).

Input: $\langle key, value \rangle$ pair, where *key* is the offset in bytes, and *value* is a string containing the hidden node output vector of each sample and the original sample.
Output: $\langle key', value' \rangle$ pair, where key' is a random string, and $value'$ is a string of the intermediate results.

```
(1)  Parse the string value to an array, named onesample;
(2)  Normalize the dependent variables of each input sample
     data;
(3)  Initiate string outvalue as a null string;
(4)  Initiate string depVarMatrix as a null string;
(5)  for i=1 to onesample.length
(6)   for j=1 to onesample.length
(7)    outvalue.append(onesample[i] × onesample[j]);
(8)    outvalue.append(",");
(9)   endfor
(10) for j=1 to depVariables.length
(11)   depVarMatrix.append(onesample[i] × depVariables[j]);
(12)   depVarMatrix.append(",");
(13) endfor
(14) endfor
(15) outvalue.append(depVarMatrix);
(16) output("HTH",outvalue);
```

Algorithm 4. HTHReducer(key, value).

Input: $\langle key, value \rangle$ pair, where *key* is a random string, and *value* is a string of the intermediate results.
Output: $\langle key', value' \rangle$ pair, where key' is a string containing all the elements in matrix $H^T \times H$, and $value'$ is any string you like.

```
(1)  Initiate matrix sumMatrix as a L × L dimension matrix
     with all elements are zero; (L is hidden node number)
(2)  Initiate matrix depVarMatrix as a L × M dimension matrix
     with all elements are zero; (M is dependent variable
     number)
(3)  while(value.hasNext())
(4)   oneValue=value.next();
(5)   Parse the string oneValue to an array, named onesample;
(6)   for i=1 to L
(7)    for j=1 to L
(8)     sumMatrix[i][j]+=onesample[i × L+j];
(9)    endfor
(10)   for j=1 to M
(11)    depVarMatrix[i][j]+=onesample[L × L+i × M+j];
(12)   endfor
(13)  endfor
(14) endwhile
(15) Initiate string outkey as a null string;
(16) for i=1 to L
(17)  for j=1 to L
(18)   outkey.append(sumMatrix[i][j]);
(19)   outkey.append(",");
(20)  endfor
(21) endfor
(22) for i=1 to L
(23)  for j=1 to M
(24)   outkey.append(depVarMatrix[i][j]);
(25)   outkey.append(",");
(26)  endfor
(27) endfor
(28) output(outkey,"HTH");
```

In Algorithm 3, Steps (1)–(4) are some preparation for the calculation, including parsing the input *value*, normalizing the dependent variables and initiating the intermediate string *outvalue* and *depVarMatrix*, where *outvalue* is the output *value* string, and *depVarMatrix* is to save the additive terms of matrix $\mathbf{H}^T \times \mathbf{T}$. Steps (5)–(15) calculate each additive term of matrix $\mathbf{H}^T \times \mathbf{H}$ and matrix $\mathbf{H}^T \times \mathbf{T}$ emitted by one sample, and append all these terms to string *outvalue*, spaced with ",". Step (16) is to output the intermediate $\langle key, value \rangle$ pair, $\langle$"*HTH*", *outvalue*$\rangle$.

In Algorithm 4, Steps (1)–(2) are some preparation for the calculation, including parsing the input *value*, and initiating the intermediate matrix *sumMatrix* and string *depVarMatrix*, where *sumMatrix* is to save the elements of matrix $\mathbf{H}^T \times \mathbf{H}$, and *depVarMatrix* is to save the elements of matrix $\mathbf{H}^T \times \mathbf{T}$. Steps (3)–(14) calculate the elements of matrix $\mathbf{H}^T \times \mathbf{H}$ and matrix $\mathbf{H}^T \times \mathbf{T}$ by summing all the corresponding additive terms. Step (15) initiates the output *key* string, *outkey*. Steps (16)–(21) append all the elements of matrix $\mathbf{H}^T \times \mathbf{H}$ to string *outkey* spaced with ",". Steps (22)–(27) append all the elements of matrix $\mathbf{H}^T \times \mathbf{T}$ to string *outkey* spaced with ",". Step (28) is to output the final $\langle key, value \rangle$ pair, $\langle$*outkey*,"*HTH*"$\rangle$ to sequence files on HDFS.

5. Experiments

During to different application backgrounds including practical stock pricing, house pricing and ailerons controlling regression problems, we design experiments for regression accuracy, optimal number of reducers and parallel performance of PELM with respect to speedup, scaleup and sizeup, which are the three most important performance evaluation of parallel algorithm at present.

All the experiments run in a cluster of 10 computers on Linux Operation System, where each has 4×2.8 GHz cores and 4 GB memory. The MapReduce system is configured with Hadoop version 0.20.2 and Java version 1.5.0_14.

5.1. Experiments for regression accuracy

In order to illustrate regression accuracy of PELM, the dataset *stock* from UCI is used in the experiments, which is obtained from practical stock pricing problems, with nine independent variables, one dependent variable and totally 950 instances. The front 900 instances are used as training dataset, and the rest 50 instances are used as testing dataset. Besides, the number of hidden nodes is assigned 20, the transfer function of hidden nodes is "Sigmoid". In the above conditions, the regression accuracy of PELM on *stock* dataset is shown in Fig. 3.

In Fig. 3, "training data" represents the 900 samples used as training dataset, "testing data" represents the 50 samples used as testing dataset, "output of PELM" represents the regression result of PELM algorithm. The average root mean square error (RMSE) of training data is 3.387, and the average RMSE of testing data is 5.502. In the same conditions, the average RMSEs of training data and testing data of the sequence Matlab program of ELM are 3.471 and 5.562, respectively. Therefore, PELM can get a similar regression accuracy with the sequence ELM algorithm.

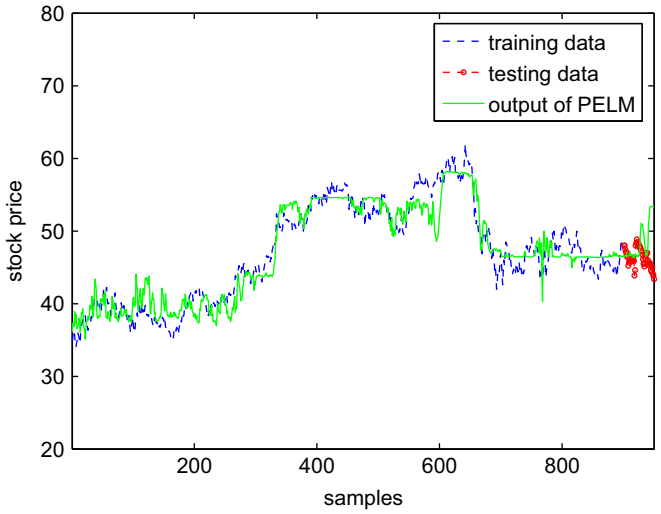

Fig. 3. Regression accuracy of PELM.

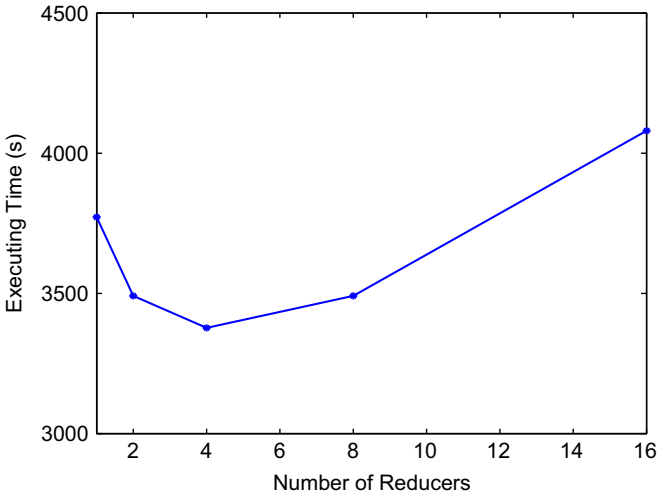

Fig. 4. Computing time under different number of reducers.

5.2. Experiments for optimal number of reducers

In Hadoop version 0.20.2, the number of mappers can be automatically determined by the cluster system, however, the number of reducers needs to be given by users. So before the parallel performance experiments, we can pick up the optimal number of reducers by assigning different number of reducers in the experiments.

We still use the dataset *stock* in the experiments. For the accuracy of experiments, the training dataset is copied to 2400-times of the original training dataset, the number of computer nodes is 2, and the computing times under different number of reducers are shown in Fig. 4.

As shown in Fig. 4, with the increase of reducers, the degree of parallelism increases, the total computing time decreases gradually and reaches its minimum when the number of reducers is 4. After that, the computing time increases with the increase of reducers because of the additional management time and the extra communication time.

As we all known, the total computing time of a Hadoop program is mainly composed of two parts, one part is computing time of each map and reduce phases, the other part is communication and networking between map and reduce phases. The more the number of reduces is, the shorter the computing time of each map and reduce phases trends. However, the longer will be the time of communication and networking between map and reduce phases. Thus there can be some trade off between the two parts.

Generally speaking, in a cluster of computers without job fails in the running time, the optimal number of reducers is 0.95 or 1.75 × number of nodes × mapred.tasktracker.tasks.maximum. In case of 0.95, all the reduces can be invoked at the same time when all the maps finish. In case of 1.75, a faster node may execute multiple reduces, thus each node could be loaded in more balance.

According to the configuration of our cluster, each node has 4 cores, so the mapred.tasktracker.tasks.maximum of each node is two. And there are two nodes used in the experiments, therefore, it can get the shortest computing time when the number of reduces is four on condition that there are no job fails in the running time.

5.3. Experiments for parallel performance of PELM

Firstly, the meanings of speedup, scaleup and sizeup are given as follows. The speedup of a larger system with m computers is measured as

$$Speedup(m) = \frac{computing\ time\ on\ 1\ computer}{computing\ time\ on\ m\ computers}$$

The scaleup is to measure the ability of an m-times larger system to perform an m-times larger job in the same computing time as the original time, which can be expressed by

$$Scaleup(m) = \frac{computing\ time\ for\ processing\ data\ on\ 1\ computer}{computing\ time\ for\ processing\ m \times data\ on\ m\ computers}$$

The sizeup is to measure how much longer it takes on a given system when the dataset size becomes m-times larger than the original dataset, which can be expressed by

$$Sizeup(m) = \frac{computing\ time\ for\ processing\ m \times data}{computing\ time\ for\ processing\ data}$$

Next, take dataset *stock* as example to evaluate the parallel performance of PELM. In order to evaluate the speedup of PELM, we run the parallel algorithm on the system with the cores number varying from 4 to 32, and the training dataset sizes are 1200-times, 2400-times, 4800-times and 9600-times of the original training dataset, respectively. The speedup performance of PELM is shown in Fig. 5.

The perfect parallel system demonstrates linear speedup: a system with m-times the number of computers yields a speedup m. However, in practice, the linear speedup is difficult to achieve because the communication cost increases when the number of computers becomes large. As shown in Fig. 5, with the increase of the dataset size, the speedup of PELM algorithm tends to be approximately linear, especially for the large dataset. The larger the dataset size is, the higher speedup can achieve, which is consistent with the actual situation.

For the evaluation of scaleup performance of PELM, we have performed scalability experiments where the increase of the datasets size is in direct proportion to the increase of the number of computer cores in the system. Thus, the datasets size of 1200-times, 2400-times, 4800-times and 9600-times of original training dataset are used on 4, 8, 16 and 32 cores, respectively. The scaleup performance of PELM is shown in Fig. 6.

In an ideal parallel system, the scaleup could be constantly equal to 1, but in practice it is impossible. In general, the scaleup of a parallel algorithm has a decline trend gradually with the increase of the dataset size and the cores number. As shown in

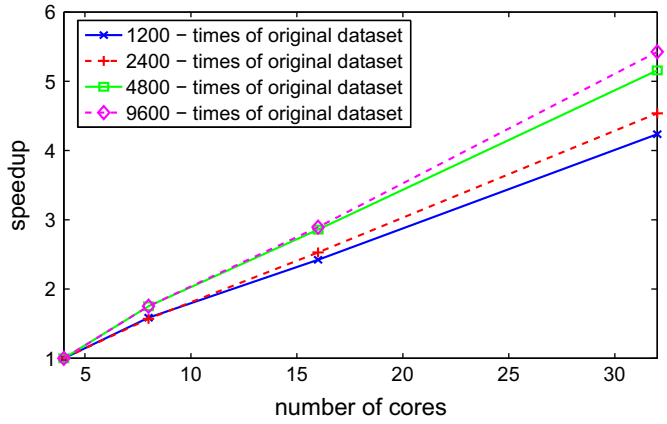

Fig. 5. Speedup of PELM.

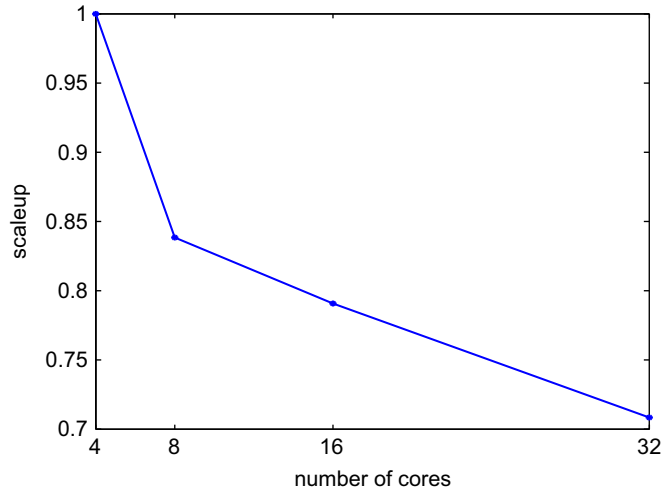

Fig. 6. Scaleup of PELM.

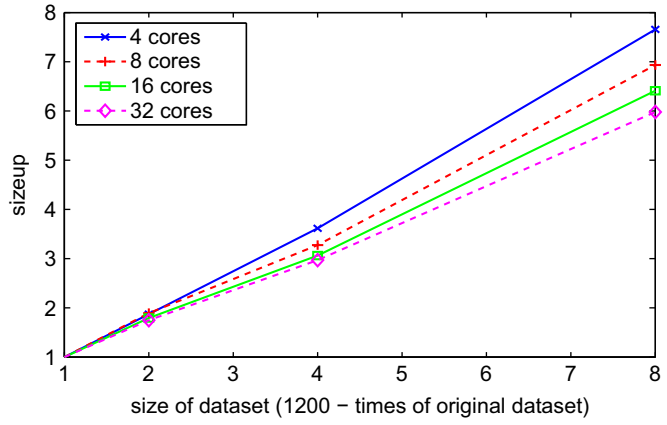

Fig. 7. Sizeup of PELM.

Fig. 6, the scaleup performance of PELM algorithm reduces slowly when the dataset becomes larger, that is, the PELM algorithm has a good scalability.

To demonstrate the sizeup performance of PELM, a series of experiments have been run on the cores number fixed system. For each experiment, the number of cores is fixed, and the sizes of datasets are 1200-times, 2400-times, 4800-times and 9600-times of original training dataset, respectively. Fig. 7 shows the good sizeup performance of PELM algorithm.

Table 1
Specification of benchmark datasets.

Name	#Training Data	#Testing Data	#Attributes	
			Independent	Dependent
cal_housing	20,000	460	8	1
delta_ailerons	5000	2129	5	1
delta_elevators	5000	4517	6	1
house_8L	20,000	2784	8	1

Table 2
Running time of PELM on benchmark datasets.

No. of cores	cal_housing run time(s)				delta_ailerons run time(s)			
	10^6	2×10^6	4×10^6	8×10^6	10^6	2×10^6	4×10^6	8×10^6
4	1900	3538	6849	13,372	1357	2560	4983	9845
8	1264	2362	4246	7978	947	1618	3099	5996
16	726	1294	2357	4419	558	924	1727	3270
32	520	858	1481	2541	387	643	1119	1999
	delta_elevators run time(s)				house_8L run time(s)			
4	1629	2920	5786	11,302	1982	3444	6901	13,313
8	1097	1860	3558	6691	1303	2104	4071	7826
16	643	1049	1982	3641	717	1203	2286	4026
32	444	724	1305	2173	495	771	1421	2483

10^6, 2×10^6, 4×10^6 and 8×10^6 in the table represent the sample number of each copied train dataset, not the times of original train dataset.

Several other datasets from UCI are also used to test the parallel performance of PELM, which have been shown in Table 1. They are all collected from practical house pricing and ailerons controlling regression problems. To form large training datasets, the original training datasets are replicated to 10^6, 2×10^6, 4×10^6 and 8×10^6 samples, respectively. The running time of each dataset in Table 2 demonstrates that the proposed PELM algorithm has a stable parallel performance in many actual problems.

6. Conclusions

ELM algorithm has recently attracted a significant amount of research attention, and it has been used to solve many regression problems. However, the enlarging data in applications make the regression by ELM a challenging task. In this paper, by analyzing the mechanism of ELM, a parallel ELM algorithm is presented and implemented based on MapReduce. Experiments demonstrate that the proposed PELM algorithm not only can process large scale dataset, but also has a good speedup, scaleup and sizeup performance. In the future work, we will conduct the experiments and improve the parallel algorithm to make better use of computing resources.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Nos. 60933004, 60975039, 61175052, 61035003, 61072085), National High-tech R&D Program of China (863 Program) (No. 2012AA011003).

References

[1] G.-B. Huang, D.H. Wang, Y. Lan, Extreme learning machines: a survey, Int J. Mach. Learn. Cybernet. 2 (2) (2011) 107–122.
[2] G.-B. Huang, H. Zhou, X. Ding, R. Zhang, Extreme Learning Machine for Regression and Multi-Class Classification, IEEE Trans. Syst. Man Cybernet.: Part B, accepted for publication.

[3] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: a new learning scheme of feedforward neural networks, in: Proceedings of the International Joint Conference on Neural Networks (IJCNN2004), vol. 2, 25–29 July 2004, pp. 985–990.
[4] G.-B. Huang, Q.-Y. Zhu, C.-K. Siew, Extreme learning machine: theory and applications, Neurocomputing 70 (2006) 489–501.
[5] G.-B. Huang, L. Chen, C.-K. Siew, Universal approximation using incremental constructive feedforward networks with random hidden nodes, IEEE Trans. Neural Networks 17 (4) (2006) 879–892.
[6] G.-B. Huang, L. Chen, Enhanced random search based incremental extreme learning machine, Neurocomputing 71 (2008) 3460–3468.
[7] G.-B. Huang, L. Chen, Convex incremental extreme learning machine, Neurocomputing 70 (2007) 3056–3062.
[8] G.-B. Huang, C.-K. Siew, Extreme learning machine: RBF network case, in: Proceedings of the Eighth International Conference on Control, Automation, Robotics and Vision, Kunming, China, 6–9 December 2004.
[9] G.-B. Huang, C.-K. Siew, Extreme learning machine with randomly assigned RBF Kernels, Int. J. Inf. Technol. 11 (1) (2005) 16–24.
[10] S. Ghemawat, H. Gobioff, S.-T. Leung, The google file system, in: Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, pp. 29–43.
[11] D. Borthakur, The Hadoop Distributed File System: Architecture and Design, 2007.
[12] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: Sixth Symposium on Operating System Design and Implementation, 2004.
[13] T. White, Hadoop: The Definitive Guide, O'Reilly Media, Inc, 2009.
[14] Hadoop Official Website 〈http://hadoop.apache.org〉.
[15] Q. He, Q. Tan, X.-D. Ma, Z.-Z. Shi, The high-activity parallel implementation of data preprocessing based on mapreduce, in: Proceedings of the International Conference on Rough Set and Knowledge Technology, vol. 6401, 2010, pp. 646–654.
[16] Q. He, F.-Z. Zhuang, J.-C. Li, Z.-Z. Shi, Parallel implementation of classification algorithms based on MapReduce, in: Proceedings of the International Conference on Rough Set and Knowledge Technology, vol. 6401, 2010, pp. 655–662.
[17] Q. He, C.-Y. Du, Q. Wang, F.-Z. Zhuang, Z.-Z. Shi, A parallel incremental extreme SVM classifier, Neurocomputing 74 (2010) 2532–2540.
[18] D. Serre, Matrices: Theory and Applications, Springer-Verlag, Inc., 2002.
[19] C.-R. Rao, S.-K. Mitra, Generalized Inverse of Matrices and its Applications, John Wiley & Sons, Inc., 1971.

Qing He is a professor in the Institute of Computing Technology, Chinese Academy of Science (CAS), and he is a professor at the Graduate University of Chinese (GUCAS). He received the BS degree from Hebei Normal University, Shijiazhang, PR China, in 1985, and the MS degree from Zhengzhou University, Zhengzhou, PR China, in 1987, both in mathematics. He received the PhD degree in 2000 from Beijing Normal University in fuzzy mathematics and artificial intelligence, Beijing, PR China. Since 1987 to 1997, he has been with Hebei University of Science and Technology. He is currently a doctoral tutor at the Institute of Computing and Technology, CAS. His interests include data mining, machine learning, classification, fuzzy clustering.

Tianfeng Shang is a PhD candidate student in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include machine learning, data mining, distributed classification and clustering.

Fuzhen Zhuang is an assistant professor in the Institute of Computing Technology, Chinese Academy of Sciences. His research interests include machine learning, data mining, distributed classification and clustering, natural language processing. He has published several papers in some prestigious refereed journals and conference proceedings, such as IEEE Transactions on Knowledge and Data Engineering, ACM CIKM, SIAM SDM and IEEE ICDM.

Zhongzhi Shi is a professor in the Institute of Computing Technology, CAS, leading the Research Group of Intelligent Science. His research interests include intelligence science, multi-agent systems, semantic Web, machine learning and neural computing. He has won a 2nd-Grade National Award at Science and Technology Progress of China in 2002, two 2nd-Grade Awards at Science and Technology Progress of the Chinese Academy of Sciences in 1998 and 2001, respectively. He is a senior member of IEEE, member of AAAI and ACM, Chair for the WG 12.2 of IFIP. He serves as a Vice President for Chinese Association of Artificial Intelligence.