Int J Parallel Prog (2017) 45:734–759
DOI 10.1007/s10766-016-0451-4

Parallelizing Convolutional Neural Networks for Action Event Recognition in Surveillance Videos

Qicong Wang[1] · **Jinhao Zhao**[1] · **Dingxi Gong**[1] · **Yehu Shen**[2] · **Maozhen Li**[3,4] · **Yunqi Lei**[1]

Received: 18 May 2016 / Accepted: 28 July 2016 / Published online: 6 August 2016

Abstract In order to deal with action recognition for large scale video data, this paper presents a MapReduce based parallel algorithm for SASTCNN, a sparse auto-combination spatio-temporal convolutional neural network. We design and implement a parallel matrix multiplication algorithm based on MapReduce. We use the MapReduce programming model to parallelize SASTCNN on a Hadoop platform. In order to take advantage of the computing power of multi-core CPU, the Map and Reduce processes of MapReduce are implemented using a multi-thread technique. A series of experiments on both WEIZMAN and KTH data sets are carried out. Compared with traditional serial algorithms, the feasibility, stability and correctness of the parallel SASTCNN are validated and a speedup in computation is obtained. Experimental results also show that the proposed method could provide more competitive results on the two data sets than other benchmark methods.

Keywords Action recognition · Convolutional neural network · Parallelization · MapReduce · Multicore

✉ Yunqi Lei
yqlei2001@163.com

1 Department of Computer Science, Xiamen University, Xiamen 361005, China

2 Department of System Integration and IC Design, Suzhou Institute of Nano-tech and Nano-bionics, Chinese Academy of Sciences, Suzhou, China

3 Department of Electronic and Computer Engineering, Brunel University, Uxbridge UB8 3PH, UK

4 School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang 212013, China

1 Introduction

Currently action recognition based on videos has a wide range of applications such as human–computer interaction, surveillance, intelligent transport system and space exploration [1–4]. The state-of-art methods for action recognition have adopted intensively the technique of machine learning [5–7]. The convolutional neural network (CNN) [8], which is a multi-layer neural network, is a biologically inspired type of deep learning model. Unlike traditional machine learning methods using complex handcrafted features, CNN is able to automatically learn discriminative features. This network model can directly apply to the original image and automatically extract the classification features for image classification eliminating the complexity and the blindness of the handcrafted features in traditional image classification. However, CNN can only be suitable for static images. To process time serial dynamic video images, we apply a spatio-temporal convolutional neural network model (STCNN) [9] to extract the image features on the spatial and the motion information on the temporal dimension from successive video frames, which can build a sparse representation. Sparse auto-encoder (SA) [10,11] is a kind of unsupervised deep learning model based on the sparse coding concept by imposing the sparse constraints to the training of each layer of the auto-encoder. In order to enhance STCNN, inspired by the sparse auto-encoder algorithm, we use a sparse auto-combination strategy to combine input feature maps to which a kind of sparsity constraint is imposed in the convolution stage. So the convolution layer is able to learn the optimal combination of feature maps as input. It can extract the essential action features from the video data. Compared to the methods of manually selected inputs, this method is more natural and is also beneficial to enhance the expressive ability of the convolution model.

With the rapid development of sensors, networks, electronic technologies, various kinds of video data are produced explosively. This has a very negative impact on the training speed of the convolutional neural network for a specific task. So, the parallel processing of large scale data from image sensors has become a key problem to be solved. These large scale data could not rely on traditional PC or supercomputer to store and process. To mine potential information from the huge amounts of data, some researchers have made many attempts in order to expand machine learning into the large scale applications on Hadoop [12–15]. Hadoop is an open source implementation of the MapReduce model, and is one of the most popular cloud computing platforms. Its requirements for the hardware are not high. Hadoop can be deployed on a number of common PC to form a powerful distributed cluster. In the background of Big Data, combination of distributed computing and machine learning will become an important development direction of machine learning.

This paper presents SASTCNN, a sparse auto-combination spatio-temporal convolutional neural network that can learn the advanced features of the video data automatically. It well suits large scale data mining. To improve the classification ability of SASTCNN and mine the essential features of the action from the large scale video data, we further parallelize SASTCNN on a Hadoop cluster named SASTCNN-MR meaning SASTCNN with MapReduce.

At present, from commercial servers to personal PCs, multi-core CPU architecture has been widely used in the design. However, multi-core programming is relatively complex. Most existing software techniques are still designed for single core platforms, which is unable to take advantage of the processing capacity [16] of multi-core CPU effectively. The Hadoop parallel programming platform based on MapReduce has very strongly computing ability by the distributed clusters, but it is also designed for the single-core CPU. Therefore, researches have proposed a number of improvement approaches for MapReduce parallel programming model. The Phoenix system [17] was implemented with MapReduce for multi-core computers. This work provides a good direction for researchers to explore the acceleration capability of multi-core CPU. The hash table with B+ tree presented in [18] is used to optimize the intermediate results of the Map process whose performance is better than that of Phoenix. An application programming interface based on Phoenix was exploited for the multi-core CPU environment [19], in which the Reduction objects managed by the programmer were defined. It can reduce the memory used in the large scale data applications. A MapReduce framework based on multi-core CPU was proposed to study the speed-up ratio of K-means, SVM, PCA and other machine learning algorithm [20]. A multi-core MapReduce framework was implemented using JAVA, called MR-J [21]. We can see that the processes of MapReduce are considered as a number of parallelizable working pieces and its execution mode is guided to speed up the working speed of MapReduce in the multi-core environment [22]. Currently, deep belief network and convolutional neural network are the two main large-scale deep learning architectures. Deep belief network had implemented on Hadoop. However, most of CNN models were accelerated on GPUs for training their deep networks, such as Torch7 [23], Caffe [24] and Theano [25]. Due to their fixed hardware, they lack the ability to develop the finer granularity parallelism. In order to utilize the computing ability of multi-core CPU effectively, we present a multi-core and multi-thread load balancing algorithm for SASTCNN-MR. We call this algorithm as SASTCNN-MRMC (SASTCNN-MR with MULTI-CORE).

The rest of the paper is organized as follows. We present a brief introduction of the framework of SASTCNN in Sect. 2. Section 3 presents the implementation of SASTCNN for action event recognition. We discuss the MapReduce implementation of matrix multiplication in Sect. 4. Section 5 describes MapReduce speed-up on multi-core CPU. The experimental results on WEIZMANN and KTH data set are reported in Sect. 6. We draw a conclusion in Sect. 7.

2 The Framework of SASTCNN

Supposing there has data $x = \{x_1, x_2, x_3, x_4, x_5\}$, for learning some compact features from x, we use an auto-encoder network including a hidden layer, as shown in Fig. 1. Layer 1 is the input layer, Layer 2 is the hidden layer, and Layer 3 is a reconstitution layer for the output layer. The training process is to minimize the error between the input layer and the output layer in the reconstitution layer. The essential features of the data can be extracted from the hidden layer, so the hidden layer can be regarded as another kind of representation of data.

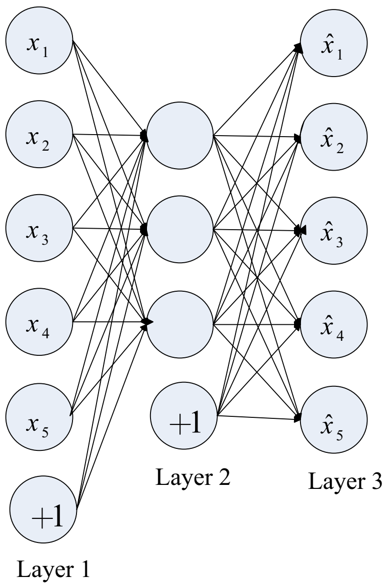

Fig. 1 The schematic diagram of the sparse auto-encoding convolutional neural network

Actually, the auto-encoder network aims to learn a function $h_{W,b}(x) \approx x$. This structure could mine the hidden features from the data by restricting the number of the neurons in the hidden layer. For instance, a 32×32 image can be processed using 1024 neurons. Training an auto-encoding network whose hidden layer has 50 neurons can realize the compact representation of the image. This is similar to the function of PCA [26], LLE [27], and other dimension reduction methods [28]. However, the number of neurons in the hidden layer is very small. Actually we can also find the inherent features from the data if the number of neurons in the hidden layer is large by imposing the sparse restrictions. Suppose the activation value of the jth neuron in the hidden layer is a_j, a network can become sparse by imposing the following constraint:

$$\rho_j = \frac{1}{m}\sum_{i=1}^{m} a_j x_i \tag{1}$$

m is the number of the neurons in the input layer. ρ_j is a sparse constraint parameter. ρ_j is not a variable and is supposed to be close to the sparse constant ρ (like 0.05). When solving the hidden layer, we can optimize ρ_j using the KL distance function.

$$\mathrm{KL}(\rho\|\rho_j) = \rho \log \frac{\rho}{\rho_j} + (1-\rho)\log\frac{1-\rho}{1-\rho_j} \tag{2}$$

In the convolution layer of STCNN, it simply specifies the input of each feature map by the spatio-temporal convolution kernel. However, the manual setting of the method restricts the automatic learning ability of the network in extracting the essential features from data. In order to further improve the feature learning ability of STCNN, we use a sparse auto-combination algorithm which can automatically learn the combination of the input feature maps as the input of the convolution layer.

For the lth sub-sampling layer, if there are N_{in} input feature maps, to calculate each output feature map of the sub-sampling layer, each feature map has only two parameters, which are the convolution kernel W_{ij}^{ℓ} and the bias term b_j^{ℓ}. We introduce a sparse constraint parameter α_{ij}, which represents the weight or the contribution of the ith input feature map $X_j^{\ell-1}$ when determining the jth output feature map X_j^{ℓ}. Thus the jth output feature map X_j^{ℓ} can be expressed as the following formula:

$$X_j^{\ell} = f\left(\sum_{i=1}^{N_{\text{in}}} \alpha_{ij}\left(X_i^{\ell-1} * W_{ij}^{\ell}\right) + b_j^{\ell}\right) \tag{3}$$

And it must satisfy the following constraints:

$$\sum_i \alpha_{ij} = 1, \quad \text{and} \quad 0 \le \alpha_{ij} \le 1 \tag{4}$$

For the back-propagation process of the lth sub-sampling layer, first we need to determine the corresponding connection relation between the sub-sampling layer and the next convolution layer. Thus the residual $\delta^{\ell+1}$of the next layer can be conducted backward. We can use the gradient descent to calculate the residual δ_j^{ℓ} of the jth feature map. Suppose the derivative $f'\left(z_j^{\ell}\right)$ of the activation function f for the input z_j^{ℓ} of the lth layer. The calculation process is the following formula:

$$\delta_j^{\ell} = f'\left(z_j^{\ell}\right) \bullet conv2\left(\delta_j^{\ell+1}, rot180\left(W_j^{\ell+1}\right)\right) \tag{5}$$

In the above calculation process, we need to rotate the convolution kernel, so that the convolution function $conv2\,(\bullet)$ can be performed cross-correlation calculation.

In the sparse auto-encoder neural network, the sparse constraint is imposed on the output. However, here we impose the sparse constraint to the input. These two working modes are different, but the functions are the same. In the sparse auto-encoder neural network, it can extract the low-level features from the input data. When using the sparse constraints on the output side, a small number of neurons in the output layer are activated. In this paper, in order to achieve the compact representation of the input data, i.e. extracting the advanced features from data, we shall restrict only a few inputs to activate a neuron in the output layer, so that it can find the most compact representation of data. The framework of SASTCNN is shown in Fig. 2.

The major change of this framework is that when making the spatio-temporal convolution, all the input feature maps of the previous layer are taken as an input of each of the output feature maps. But the number of feature maps fed to the output feature map is extremely limited due to the existence of the sparse constraints.

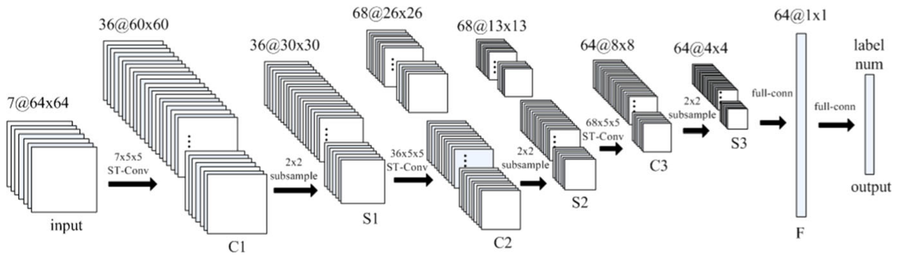

Fig. 2 The structure of SASTCNN

3 The SASTCNN for Action Event Recognition

In the above framework, to capture the motion information encoded in multiple contiguous video frames, taking the current frame as the center, we take 7 consecutive 64×64 size frames as the input of the SASTCNN. Suppose the input frames are all 64×64 size gray images. If the sizes are different, they must be normalization to 64×64 size through scaling method.

The C1 layer can get 36 feature maps from the seven consecutive input frames by using $7 \times 5 \times 5$ convolution kernel, which means using 36 different learnable convolution kernels to extract 36 different features. Although action event classification depends on lots of complicated characteristics, 36 feature maps extracted from the input frames are fully able to classify the simple action. Due to the size of the convolution kernel is $7 \times 5 \times 5$, on the time dimension, each feature map takes seven successive frames as the input, which amounts to fully connecting on the time dimension. While on the space dimension, the size of the convolution kernel is 5×5. It means that each neuron of each feature map of the C1 layer is connected with all seven 5×5 size image block. Then the output of the C1 layer is thirty-six 60×60 size feature maps.

The S1 layer is a sub-sampling layer. It aims to scale the obtained feature maps of the C1 layer, which can enhance the robustness of the SASTCNN for the scale changes and the slight deformation. The scaling factor of the sub-sampling layer cannot be set too big. Otherwise we could not be able to extract effective features from the original image data. We use uniform scaling for the thirty-six 60×60 size feature maps of the C1 layer, so the output of the S1 layer is thirty-six 30×30 feature maps.

The C2 layer is also a convolution layer, but it is different from the C1 layer. The C1 layer takes seven consecutive frames as input for spatio-temporal convolution, and whiles the C2 layer takes 36 feature maps as input for convolution which uses $3 \times 5 \times 5$ size convolution kernel. Then we can get two groups of feature maps. Each group contains 34 feature maps. Its working process is as follows. For the 36 input feature maps, we take every three adjacent feature maps as input for convolution, which can produce thirty-four ($36 - 3 + 1 = 34$) different combinations in total. The size of the convolution kernel is 5×5, so the size of each feature map is 26×26. We get two groups of feature maps by repeating this process twice.

The S2 layer is similar to the S1 layer which is a sub-sampling layer. The scaling factor is two, and we can obtain two groups of the feature maps, which has thirty-four 13×13 size feature maps.

The C3 layer is also a convolution layer. It takes the two groups of the feature maps from the S2 layer as input, using $3 \times 6 \times 6$ size convolution kernel for convolution, and it will get two sets of feature figures, each set has thirty-two 8×8 size feature maps. The previous is 5×5 size convolution kernel, but here is 6×6 size convolution kernel. The main reason is convenient to follow by a sub-sampling layer. If we use 5×5 size convolution kernel, the size of output feature map is 9×9. Because nine is not even, we are not able to use the scaling factor 2 for sub-sampling. Then 64 feature maps can be merged into a group directly from the obtained output of the C3 layer.

The S3 layer is similar to the S2 layer a sub-sampling layer, the scaling factor is two, and can get sixty-four 4×4 size feature maps.

The S3 layer is followed by a fully connection layer, called F layer. The fully connection means that each neuron of the S3 layer is connected to each neuron of the F layer, so that it degenerates into a general neural network. Actually, we can spread all the neurons of the S3 layer as a network layer which has $1024(64 \times 4 \times 4 = 1024)$ neurons, and then we carry out a fully connection to the F layer which has sixty-four neurons. Therefore the S3 layer and the F layer have 1024×64 connections totally.

The final output layer is a fully connection layer after the F layer. The number of its neurons is the number of action event recognition. The one of output neurons with maximum activation is seen as the classification results. For example, if the first output neuron has the maximum value, this network recognition results represent the input samples are belong to the first class.

4 MapReduce Implementation of Matrix Multiplication

The training process of SASTCNN is a continuous iteration process, in which each iteration is highly dependent on the previous results. It is not suitable for parallelization using MapReduce on the framework. However each iteration is a matrix multiplication actually. The Algorithm 1 is a global SASTCNN training framework based on MapReduce.

Algorithm 1: SASTCNN-MR training Algorithm

```
1:train SASTCNN(samples)
2:init a global SASTCNN
3:for sample in samples
4:    output=feed-forward-MapReduce(SASTCNN,    sample)
5:    error=Calculate-error(output,    label)
6:    Error-backpropagation-MapReduce(SASTCNN,    error)
7:end for
8:save SASTCNN
```

Each sample is used to update it. When updating, we employ MapReduce parallelization to the forward propagation and the error back propagation. To accelerate the training of SATCNN using MapReduce, we propose a matrix parallel computing method based on MapReduce. Suppose there are two matrices $\boldsymbol{A}_{m\times t}$ and $\boldsymbol{B}_{t\times n}$. In order to calculate $\boldsymbol{C}_{m\times n} = \boldsymbol{AB}$, the traditional algorithm is as follows:

Algorithm 2: Traditional Matrix Multiplication Algorithm

```
1:Input: $\boldsymbol{A}_{m\times t}$, $\boldsymbol{B}_{t\times n}$
2:Output:$\boldsymbol{C}_{m\times n}$
3:  $\boldsymbol{C}_{m\times n}$:=0;
4:  for i:m
5:    for j=1:n
6:      for k=1:t
7:          $C_{i,j}$= $C_{i,j}$ + $A_{i,k}B_{k,j}$;
8:      end for
9:    end for
10: end for
```

The time complexity of the above algorithm is $\mathrm{T} = O(mnk)$. It is equivalent to the amount of cubic level calculation. This is a very time consuming operation. Carefully observing the above algorithm, $C_{i,j}$ is only related with the ith row of $\boldsymbol{A}$ and the jth column of $\boldsymbol{B}$. Therefore, when calculating different $C_{i,j}$, we can send them to different machines for parallel computing. As a result, we design a parallel matrix multiplication algorithm based on MapReduce.

Given matrices $\boldsymbol{A}_{m\times t}$ and $\boldsymbol{B}_{t\times n}$, we solve $\boldsymbol{C}_{m\times n} = \boldsymbol{AB}$. Considering the acceptable input of MapReduce process is a file which consists of some KVP (Key Value Pair), we need to take some constraints on the format of $\boldsymbol{A}$ and $\boldsymbol{B}$. Either $\boldsymbol{A}$ or $\boldsymbol{B}$ meet the following conditions: for the matrix $\boldsymbol{M}$, it always saves one element each line in a distributed file, and for the element $M_{i,j,}$ it is stored as 'M_i_j_$M_{i,j}$', in which M is the name of the matrix, i and j represent the index subscript of the element. For example, if the value of M_{13} is 7, it is saved as 'M_1_3_7'. According to the above assumption, we design the Map process and the Reduce process as shown in Algorithm 3.

Algorithm 3: MapReduce Implementation of Matrix Multiplication

```
1:Map(Object key, Text value, Context context):
2:// key: row number
3:// value: M_$i$_$j$_$M_{i,j}$
4:if M=='A'   //value is an element of A
5:   for $c$=1:$n$
6:      map_key. set($i$_$c$);
7:      map_value. set(A_$j$_$A_{i,j}$);
8:      context. write(map_key, map_value);
9:   end for
10:else      //value is an element of B
11:   for $r$=1:$m$
12:      map_key. set($r$_$j$);
13:      map_value. set(B_$i$_$B_{i,j}$);
14:      context. write(map_key, map_value);
15:   end for
16:end if
17:Reduce(Object key, Iterator<values>, Context)
18://key: $r$_$c$
19://values: A_$j$_$A_{i,j}$ or B_$i$_$B_{i,j}$
20:sum:=0;
21:$\boldsymbol{A}_{\text{row}}$:=0;
22:$\boldsymbol{B}_{\text{col}}$:=0;
23:for value in values
24:   if value==A_$j$_$A_{i,j}$
25:      $A_{\text{row}}[j]$. set($A_{i,j}$);
26:   else
27:      $B_{\text{col}}[i]$. set($B_{i,j}$);
28:   end if
29:end for
30:for $k$=1:$t$
31:   sum+=$A_{\text{row}}[k]$*$B_{\text{col}}[k]$;
32:end for
33:context. write(C_$r$_$c$_sum);
```

Now we analyze the Algorithm 2. During the Map process, it just copies the elements of $\boldsymbol{A}$ and $\boldsymbol{B}$, and then they are submitted to the Reduce process for calculating. For each element $A_{i,j}$ of $\boldsymbol{A}$, it must be multiplied with $B_{j,c} (c = 1, 2, \ldots, n)$. $A_{i,j} {}^* B_{j,c}$

will accumulate to $C_{i,c}$. Thus (i, c) is used as the keyword to identify. In the end it will produce m^*t^*n elements. For each element $B_{i,j}$ of $\boldsymbol{B}$, it must be multiplied with $A_{r,i}(r = 1, 2, \ldots, m)$. $A_{r,i}{}^*B_{i,j}$ will accumulate to $C_{r,j}$. (r, j) which is used as the keyword to identify. Finally it will produce t^*n^*m elements. During the shuffling phase, the $<$ key, value $>$ which is the same as keyword (r, c) is submitted to Reduce for processing. After shuffling, the keyword (r, c) has $2t$ elements which are just the rth row of $\boldsymbol{A}$ and the cth column of $\boldsymbol{B}$. During the Reduce phase, the $2t$ elements are divided into two sets, in which one set is the rth row of $\boldsymbol{A}$, denoted by $\boldsymbol{A}_{\text{row}}$, and the other is the cth column of $\boldsymbol{B}$, denoted by $\boldsymbol{B}_{\text{col}}$. $C_{r,c}$ is the dot product by those two vectors.

On Hadoop, the Map function and the Reduce function can be considered as the smallest parallel execution units. Supposing the above program is running in a distributed cluster composed of p computers, we will analyze the time complexity of the algorithm. During the Map phase, the amount of data of matrix A that need to be processed is mt, i.e. there are mt Map functions that need to be executed in total. Those Map functions can be executed by the p computers in parallel. The time complexity of each Map function is $\mathrm{O}(n)$, so the ideal time complexity for processing matrix $\boldsymbol{A}$ is $\mathrm{O}(\max(mt/p, 1)n)$. The meaning of $\max(mt/p, 1)$ is that the mt Map functions can be at most distributed to mt computers when p is infinite, i.e. each computer processes one Map function, so it spends the time of processing one Map function at least. Then the ideal time complexity is $\mathrm{O}(n)$ when the cluster is infinite. Similarly, the time complexity of processing matrix $\boldsymbol{B}$ is $\mathrm{O}(\max(tn/p, 1)m)$. Therefore, when p is small, the total time complexity of the Map phase can be considered as $\mathrm{O}(mtn/p)$. When p is very large, the total time complexity of the Map phase is $\mathrm{O}(m + n)$.

During the Reduce phase, it needs to execute mn Reduce functions. The time complexity of each Reduce function is $\mathrm{O}(t)$. Therefore, the total time complexity of the Reduce phase is $\mathrm{O}(\max(mn/p, 1)t)$. When p is small, the time complexity of the whole MapReduce process is $\mathrm{O}(mtn/p)$. When p is very large, the time complexity of the whole MapReduce process is $\mathrm{O}(m + n + t)$. The above analysis does not consider the time cost of network transmission and the cost of file reading and writing during the MapReduce process. Due to the different circumstance, they may be large differences during the experiment process. Considering the time cost of network transmission and file reading and writing, supposing the time cost of each Map function and each Reduce function is e, the time complexity of the Map phase is $\mathrm{O}(\max(mt/p, 1)(n + e) + \max(nt/p, 1)(m + e))$, and the time complexity of the Reduce phase is $\mathrm{O}(\max(mn/p, 1)(t + e))$. Then the total time complexity is $\mathrm{O}(\max(mt/p, 1)(n + e) + \max(nt/p, 1)(m + e)) + \mathrm{O}(\max(mn/p, 1)(t + e))$.

In SASTCNN, considering that it has sparse constraints to the network, most matrices composed of parameters are also sparse. We can further improve the above matrix multiplication. At first, when storing the matrix, we only store nonzero elements. Then the Reduce function of Algorithm 3 can be modified as shown in Algorithm 4.

Algorithm 4: Reduce Function of Sparse Matrix Multiplication Algorithm

```
1:Reduce(Object key, Iterator<values>, Context)
2://key: $r\_c$
3://values: A_$j$_$A_{i,j}$ or B_$i$_$B_{i,j}$
4:sum:=0;
5:$\boldsymbol{A}_{\text{row}}$:=0;
6:$\boldsymbol{B}_{\text{col}}$:=0;
7:for value in values
8:    if value==A_$j$_$A_{i,j}$
9:        $A_{\text{row}}[j]$. set($A_{i,j}$);
10:   else
11:       $B_{\text{col}}[i]$. set($B_{i,j}$);
12:   end if
13:end for
14:for $k$=1:$t$
15:   if $A_{\text{row}}[k]>0$ and $B_{\text{col}}[k]>0$
16:       sum+=$A_{\text{row}}[k]*B_{\text{col}}[k]$;
17:   end if
18:end for
19:if sum>0
20:   context. write(C_$r$_$c$_sum);
21:end if
```

Because the Map phase just copies the elements, there is already a sparse matrix stored in the file. It is not necessary to modify the Map function. During the Reduce phase, it needs to do multiplication only when the corresponding position elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ are both nonzero, and it writes the results only when the corresponding position element of $\boldsymbol{C}$ is nonzero. When the matrix is sparse, the above algorithm can further improve the efficiency of the algorithm.

5 MapReduce Speed-up on Multi-core CPU

5.1 MapReduce Implementation with Multi-thread

Hadoop is a highly successful MapReduce open source parallel computing platform. However, it does not consider a computing environment using multi-core CPUs. Hadoop is a master-slave structure, i.e. there is a Master host working as a controlling machine, and multiple Slave hosts working as the working machines. The Master is responsible for distributing, tracking, and summarizing the results. The Slaves are responsible for task executions. Whenever there is a job submitted to the Master, the Master assigns the tasks to different Slave hosts. Each Slave host produces a MapTask process and a ReduceTask process which deal with the Map task and the Reduce task allocated from the Master respectively. Multiple jobs can be in parallel execution at the same time. Each job produces one MapTask and one ReduceTask on different com-

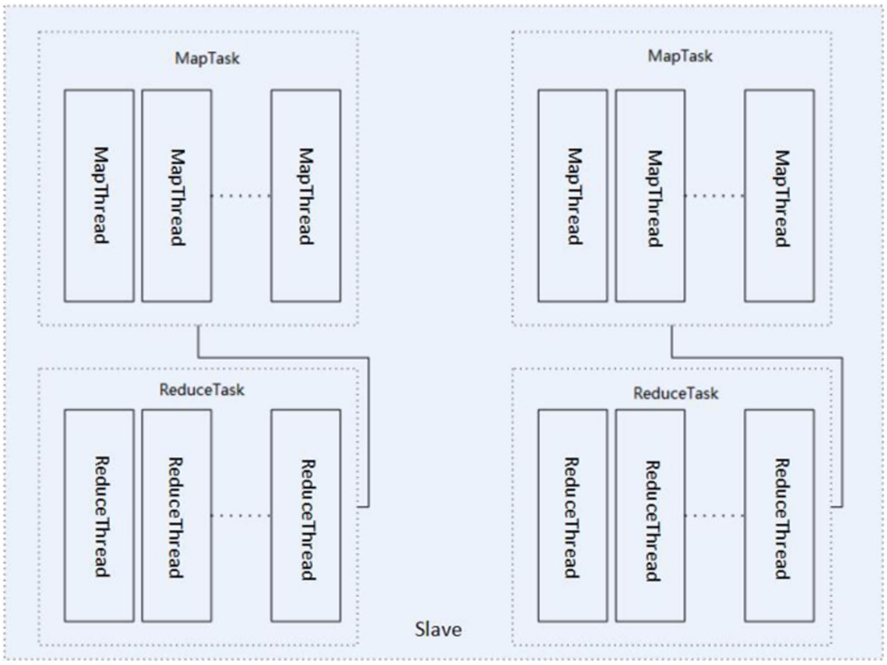

Fig. 3 Running MapReduce on a multi-core computer

puters, which the finer granularity of parallel is realized, so Hadoop can be competent for analyzing a large scale data set using multiple tasks. When Hadoop executes one job, it only works in parallel on multiple computers, but it cannot take advantage of multi-core of CPU of a single computer.

To realize the finer granularity of parallel for each job and improve the execution efficiency of a single task on Hadoop, we implemented the Map function and the Reduce function using multiple threads, which makes full use of the computing ability of a multi-core computer. The Map function and the Reduce function are shown in Fig. 3.

The Map process and the Reduce process of each job are all executed by multiple threads. The Map process on each Slave runs multiple times. At each time one input is processed. The input may be a row of the file, or may be a special format specified by users. The execution mode of the Reduce function and the Map function are similar, therefore we only analyze the Map function to illustrate the differences when applying multi-threading technique. Without multi-threading, the execution of the Map function is serial. Its execution mode on the multi-core CPU is shown in Fig. 4.

The multiple execution of the Map function can be viewed as a sequential execution of multiple Map tasks. Logically, they are independent of each other. They are all bound to one process, so that they could not be divided into different CPU cores for parallel execution. Their execution on the multi-core CPU is not symmetrical. One core is always very busy, and meanwhile the other cores are in the idle state. As Fig. 4 shows, the *n* Map tasks wait for execution on the core CPU0. The *n* Map tasks together can be

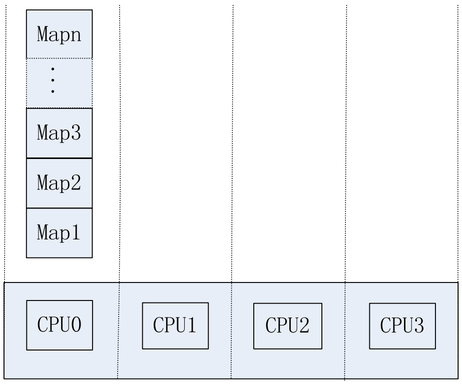

Fig. 4 Running the Map process on a single core

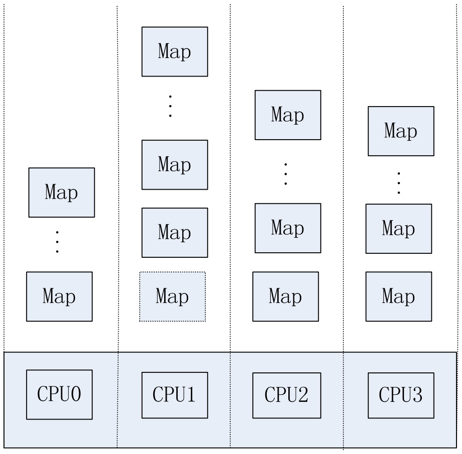

Fig. 5 Running the Map process on multiple cores

considered as a large task. This is the reason why CPU0 is always busy while CPU1, CPU2, and CPU3 are in the idle state. If we can utilize the four cores to execute these tasks simultaneously, the execution efficiency can be improved significantly.

The execution process of the Map function using multiple threads can be viewed as multiple independent Map tasks which have no order difference. Their execution mode on the four-core CPU is shown in Fig. 5.

The Map function adopting multiple threads can generate multiple Map tasks to different CPU cores for running at the same time, so they can be executed in parallel on the multi-core CPU to improve the execution efficiency.

One time execution of the Map function is taken as a Map task, and each Map task can be seen as a thread. The produced threads may be a lot. If we take a row of the file as the input of a Map task, for a large file, thousands of threads can be produced. Every produced thread needs to consume a certain amount of system resources. Although these resources may be very small, when the number of the threads is very large, these resources could also substantial. Moreover, the generation and the destruction of the thread also needs some time.

Every Map task producing a thread is not cost-effective. For this reason, we further use a thread pool, which can save the system expense, to control the threads. The

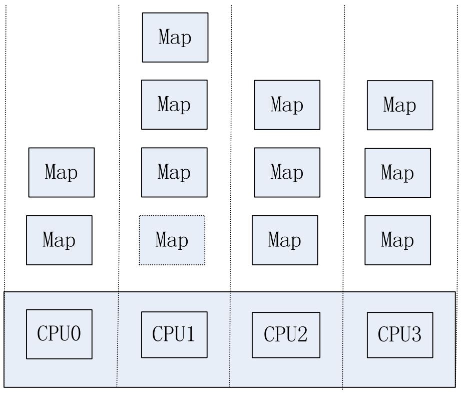

Fig. 6 Running the Map process on a four-core CPU using a thread pool

thread pool produces a certain number of threads called by the Map tasks. When a Map task appears, it can take a thread from the thread pool to execute. After the Map task is finished, the thread pool recycles the thread. The number of threads can be kept to a certain number, so that it does not consume too many system resources. Suppose that the execution mode of Fig. 5 is replaced with a thread pool using 12 threads, at a certain time the execution process is shown in Fig. 6.

5.2 The Load Balancing Algorithm of Multi-core CPU

The more the number of threads is, the higher the overhead of thread management, and the greater the complexity of the assigned tasks. But the computation capacity of CPU can be improved. So, after the Map function and the Reduce function of Hadoop are implemented using multiple threads, load balancing can be used to make the executions of both the Map function and the Reduce function more effective. We can see the difference from Figs. 4 and 5. We have not mentioned the allocation of the multiple Map tasks to each CPU core. Thus for a thread pool with 12 threads, Figure 6 are only the ones of the possible executive modes. Actually, the multiple Map tasks may entirely execute on CPU0, which is the same as the serial of single core CPU. We need to carry out load balancing for the multi-core CPU, so that we can utilize all the CPU cores more effectively.

When there is a new task, it is always assigned to the least busy CPU core. If a CPU core finishes all the tasks, the busiest CPU core is found out. Some tasks are taken from the busiest CPU core to the least busy CPU core, so that the loads of the two CPU cores can be balanced.

We analyze the execution process of the Map function on multi-core CPU as shown in Fig. 6, which supposes that the whole process is executed under the ideal states. If the operation speed of all the CPU cores is the same, at the next moment after task allocation, the load of the four CPU cores is shown in Fig. 7a. Meanwhile, keeping the number of threads invariant, it will produce four Map tasks. If the four tasks are produced at the same time, the allocation process of the four Map tasks as follows: firstly find the least-busy CPU core, and assign all the four Map tasks to CPU0. This

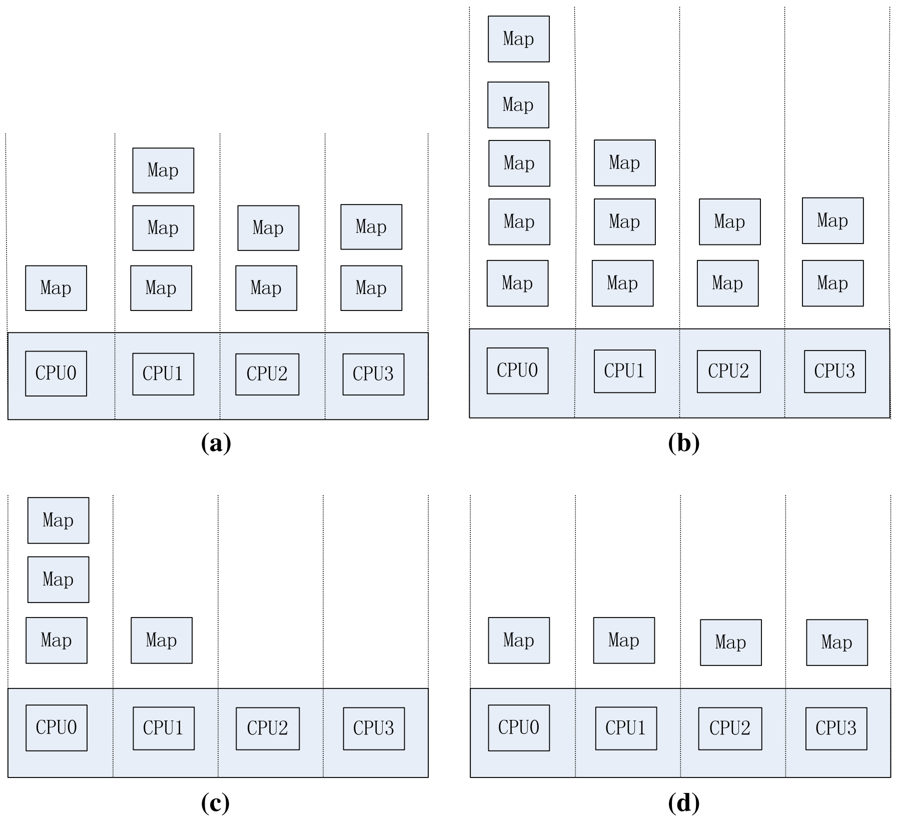

Fig. 7 Load balancing on multi-core CPU

is because they tend to be related when the tasks are produced at the same time. They are allocated to a CPU core, so that they are more convenient in communication. The load of CPU after allocation is shown in Fig. 7b. If there is no new task after that and CPU2, CPU3 have finished the tasks, the load is shown in Fig. 7c. At this moment, because CPU2 and CPU3 are free, and CPU0 and CPU1 still have tasks, task transferring will occur. A task from the busiest CPU0 will be transferred to CPU2 and CPU3 respectively. The final load balancing is shown in Fig. 7d. Ideally, its speed-up ratio is p compared with the single core CPU, in which p is the number of CPU cores.

6 Experimental Results

6.1 Experimental Environment

The development platform is a Hadoop cluster composed of 3 PCs connected by a local area network. The specific hardware and software environment of the cluster is shown in Table 1.

Table 1 The hardware and software configurations

Computer name	Operation system	CPU	Memory	Hard disk
Hadoop cluster				
Master	Ubuntu 12. 10 64 bit Server	Intel Core i7 3770k	32 GB	2 TB
Slave1	Ubuntu 12. 10 64 bit Server	Intel Core i7 3770k	32 GB	2 TB
Slave2	Ubuntu 12. 10 64 bit Server	Intel Core i7 3770k	32 GB	2 TB

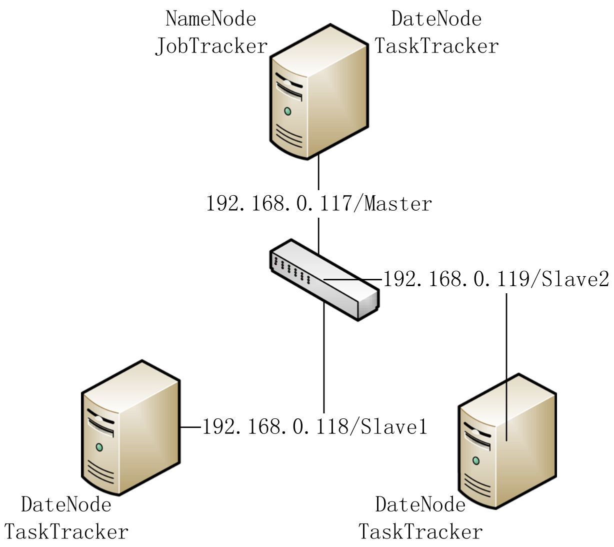

Fig. 8 The topological structure of the Hadoop cluster

All the experiments are conducted using Ubuntu12.04 operating system, and all the CPUs are Intel Core i7 3770k with 4 cores. The topological structure of the Hadoop cluster is shown in Fig. 8.

In a traditional Hadoop cluster, the Master node usually works as NameNode, but here the Master host works not only as NameNode, but also as DataNode. This is because the number of the computers in this cluster is small, and this can also be beneficial to fully utilize the hard disk of all the machines.

6.2 Action Event Recognition

For action event recognition, the WEIZMANN data set is the commonly used to test the performance of algorithms. It is collected by 9 subjects and contains 10 types

Table 2 Comparison of obtained results with other methods

	Recognition rate (%)	Frames
Blank et al. [29]	99.6	10
Niebles and Fei-Fei [31]	55.0	12
Jhuang et al. [30]	93.8	9
Our work 1-frames	87.0	1
Our work 3-frames	90.7	3
Our work 7-frames	94.9	7

of action events. In order to illustrate the superiority of the proposed approach, we compare SASTCNN with the related works on the WEIZMANN data set. Table 2 is a comparison of obtained results on WEIZMANN data set with the methods for the segmented video clips.

It must be noted that they cannot be compared directly due to two different strategies. Each video clip is taken as a unit and assigned an event label in our method and literature [29]. The methods of literatures [30,31] use a time window to calculate the feature, but a label is only assigned to the frame which is the middle one of the window. BLANK [29] assigns a label to each video clip which consists of 10 frames, and the features are calculated from these ten frames. JHUANG [30] extracts the features from each nine adjacent frames (Centre on the current frame, taking four frames forward and backward respectively). Although the recognition rate of our method is not the highest in the best case, when it uses one-frame video, we can still get a very high recognition rate. It shows that our method has good stability. In the different applications, our method is more flexible, because it can adjust its own parameters according to different needs.

6.3 Performance of MapReduce Matrix Multiplication

In order to validate the correctness of SASTCNN-MR after using Hadoop MapReduce, both WEIZMANN and KTH data sets were used in the experiments. We compared the results of parallel experiment with the results of serial experiment. For the WEIZMANN data set, we employed a leave-one-out cross validation method that takes one sample as the test set and the remaining ones as the training set. We repeated the experiment 81 times. In the end, we took an average of all the results. As shown in Fig. 9, the action classification ability of SASTCNN-MR is almost the same as the action classification ability of SASTCNN.

This illustrates the portability of the method in this paper, and confirms the feasibility and the correctness of SASTCNN-MR parallelization. For the KTH data set, we use the same settings as described previously. All the experiment evaluations of the KTH data set execute under the 5 fold cross-validation method, which is to say that all the samples are randomly divided into 5 groups of an equal size. We randomly select 1 data set as the testing set and the rest of them are the train-

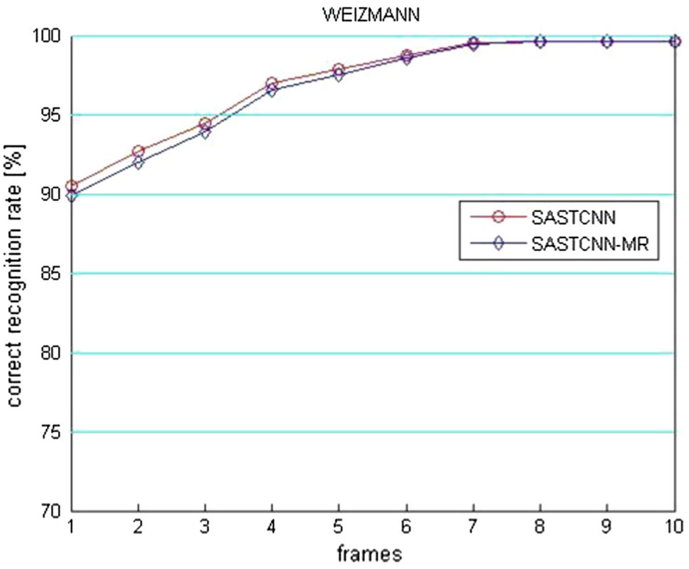

Fig. 9 The accuracy of SASTCNN-MR on the WEIZMANN dataset

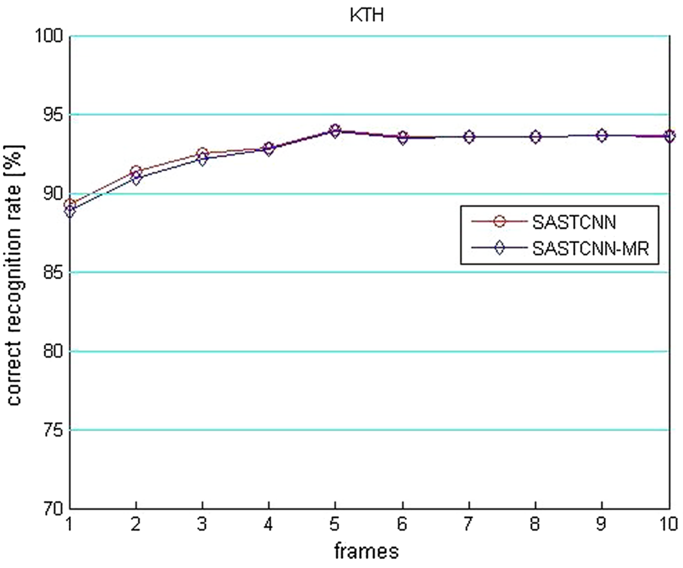

Fig. 10 The accuracy of SASTCNN-MR on the KTH dataset

ing sets. This procedure was executed 5 times and the results were reported as the average of the 5 experiments. As shown in Fig. 10, the comparison experiments on the KTH data set also confirm the correctness of the SASTCNN-MR algorithm.

Table 3 A comparison of the computational costs

	SASTCNN (ms)	SASTCNN-MR (ms)	Speedup
WEIZMANN			
Training	97,854	79,485	1.23
Testing	15,652	10,422	1.50
KTH			
Training	793,412	643,776	1.23
Testing	117,167	77,478	1.51

In order to analyze the speed-up ratio after SASTCNN-MR parallelization, we compared the training time and testing time between SASTCNN and SASTCNN-MR respectively. Here the training and testing time of SASTCNN refer to the running time of SASTCNN-MR with a single machine equipped with Hadoop, so the obtained speed-up ratio is the relative speed-up ratio. For the WEIZMANN data set, we extracted a video segment which contains 81 samples. Furthermore, we copied each of them 100 times and get 8100 samples as a result. We train the data samples 10 times and averaged the training time results. We selected 900 samples randomly for testing and repeated it 10 times.

Using the same experimental method on the KTH data set, we generated 60,000 samples in total. We selected 10,000 samples randomly as the testing set. The reason for separating the training process and the testing process was that there are essential differences between the training process and the testing process, i.e. the training process can only accelerate the matrix operation, but the testing process is the full acceleration for each testing sample. Results are shown in Table 3 from which we can see that the speed-up ratio is not very high.

Using the Hadoop cluster composed of 3 computers, the theoretical speed-up ratio should be 3, but actually the obtained speed-up ratios is far less than 3. The reason is probably that the network transmission delay and the reading and writing files occupied most of the time. It can be seen that the speed-up ratio of the training process is too small, this is because the training process mainly rely on the matrix operation to accelerate, and the matrix size is not very large, the network transmission delay overwhelm the acceleration effect by the matrix parallel algorithm. The speed-up ratio of the testing process is higher than that of the training process due to the reason that the testing process takes the Map process and the Reduce process as the granularity of parallelism, the acceleration effect is higher than the matrix parallel operation.

6.4 Load Balancing on Multi-core CPU

In this section we analyze the impact of multi-core CPU on the execution time of SASTCNN-MR. When we create initial tasks in the system, these tasks are put in a

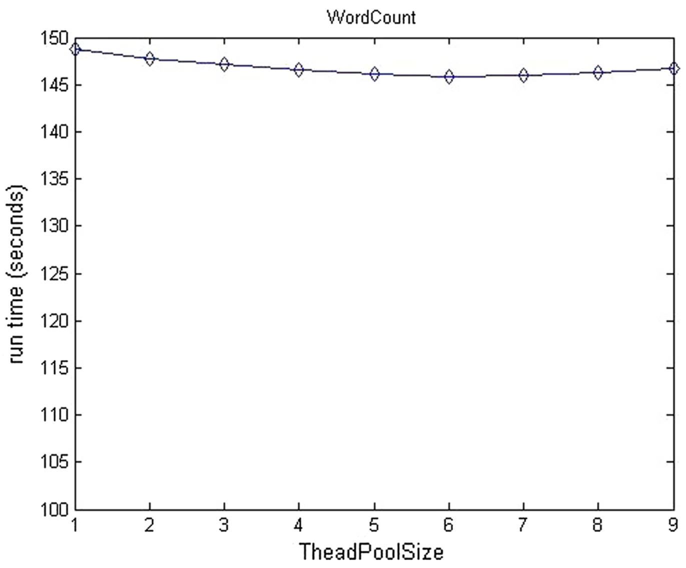

Fig. 11 Running the Word Count application using the thread pool

given run queue. Usually, we don't know when a task is short and when it needs a long run. Therefore, the initial task assignment to CPUs may be not ideal. In order to maintain the balance of task load between CPUs, the tasks can be redistributed. For sequential execution tasks, they can be divided into a number of nodes, each node is a thread. The channels must be placed between the nodes to form a pipeline. This can also greatly enhance the utilization rate of CPU. The experimental environment in this section is still the same. As mentioned before, the threads of MapReduce are controlled by the thread pool. To determine the possible optimal size of the thread pool, we conducted the following experiment. First we created a text file composed of English words. The size of the file is 101.4 MB, and we implemented a Word Count application using multi-threading technique. We recorded the average running time with different thread pool sizes. The results are illustrated in Fig. 11.

We can see that the size of the thread pool has little effect on the Word Count application. This is because the Word Count application is an IO intensive. For confirmation purpose, we further designed the following experiment. We produced a dataset whose size is 56.6 MB. In this dataset, each line consists of 100 integers. Then we implement the following MapReduce program as shown in Algorithm 5 to sum over each row.

Algorithm 5: SUM MapReduce Algorithm

```
1:Map(Object key,    Text value,    Context context)
2://key: row number
3://value: 100 int number
4:int sum=0;
5:String[] numbers=value. split();
6:for number in numbers
7:    sum=sum+Integer. parseInt(number);
8:end for
9:Context. write(key, sum);
```

Algorithm 5 only has the Map process, and it does not have the Reduce process. It is handled by the default Reduce function in Hadoop, the default Reduce function copies the Map function's output directly without any processing. The above sum program needs to sum over each line of 100 integers, thus it is also a CPU-intensive operation. In the experiment, according to the size of the thread pool, we recorded the running time of each test of the sum program, and we took the average time of 10 running results which are shown in Fig. 12.

We can see that the sum program is closely related to the thread pool size. The running time decreases quickly as the size of the thread pool increases. When the number of the threads reaches 5, the running time reaches the lowest. After that, as the thread pool size increases, the running time is relatively stable. The CPU used in the experiment is a four-core CPU, and the best number of threads in the thread pool is just around 4. In the following experiment, the size of the thread pool was set to 8 conservatively. From Fig. 12 we can see that the number of threads should not be too small, otherwise it will greatly increase the running time, and the influence of increasing the number of threads is too small.

It should be noted that the above two experiments were both conducted on a single machine. Using the Top command to see the CPU usage rate, we obtained a comparison chart of CPU usage at a certain time as shown in Fig. 13.

We can see that the CPU usage rate of the Word Count application is too low, while the CPU usage rate of the sum program is higher. This again confirms that the Word Count application is IO Intensive, whereas the sum program is CPU Intensive.

Because Word Count application and action event recognition in this paper both take the Map function and the Reduce function as the smallest parallel execution units, we consider their empirical sizes of the thread pool should be similar. Through the above experiments, we set the size of the thread pool to 8. In this section we analyze

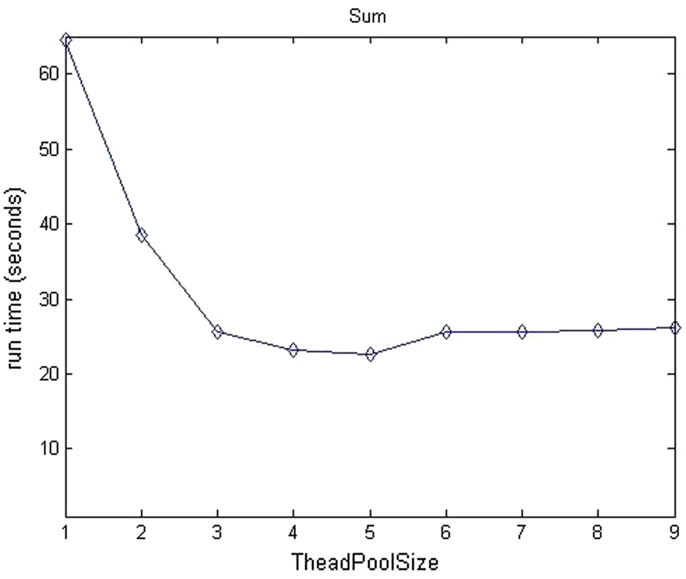

Fig. 12 Running the sum program using the thread pool

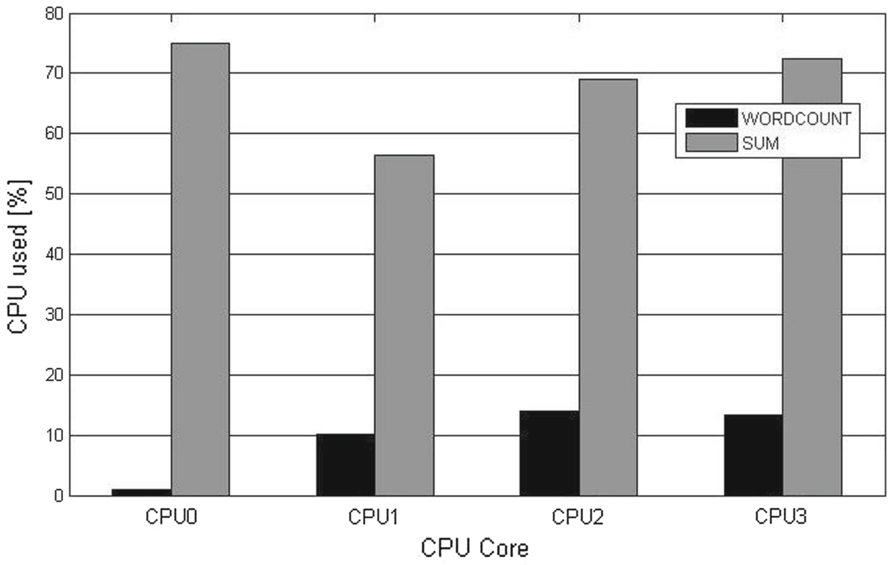

Fig. 13 A comparison of the CPU usage between the Word Count application and the sum program

the improvement of execution efficiency using multi-core CPU. In order to analyze the speed-up ratio of SASTCNN-MR, we compare the training time and the testing time of SASTCNN-MR and SASTCNN-MRMC (SASTCNN-MR with MULTI-CORE). We still use the above experiment procedure for speed-up ratio analysis. Table 4 is

Table 4 A comparison of the running times between SASTCNN-MR and SASTCNN-MRMC on a single machine

Environment	Stand-alone		
	SASTCNN-MR (ms)	SASTCNN-MRMC (ms)	Speedup
WEIZMANN			
Training	97,854	43,685	2.24
Testing	15,652	5252	2.98
KTH			
Training	793,412	352,628	2.25
Testing	117,167	38,797	3.02

a comparison of the running times between SASTCNN-MR and SASTCNN-MRMC on a single machine.

We can see that after using multi-core CPU, the computational efficiency is improved more than twice for both the training and testing processes. The experiments were tested on a four-core CPU, the theoretical speed-up ratio is 4, but due to the scheduling among multiple CPU cores, IO delay and other factors, it is impossible to reach the theoretical speed-up ratio. In this work, the obtained speed-up ratio is around 3 which is quite high due to the small granularity during the testing process.

The obtained speed-up ratio in the training process is still inferior to the speed-up ratio in the testing process. This is because there are time-consuming serial execution parts during the training process. It is consistent with the above experimental result. Comparing the results with Table 3, we can find that the acceleration rate with multi-core CPU is better than that using a Hadoop cluster. This may be caused by the following reasons. Transmitting data among different machines requires more time than between different CPUs. There are only 3 computers in the cluster whose parallel computing ability is limited. However, the CPU has 4 cores. As a result the computational power should be higher compared with the Hadoop cluster. Figure 14 illustrates the host CPU utilization percentages for both SASTCNN-MR and SASTCNN-MRMC on a single machine.

The CPU utilization percentage can also reflect the difference of execution efficiency. From Fig. 14 we can conclude that when running SASTCNN-MR, there is only one CPU core having a high CPU usage rate, but the other CPU cores are relatively free. While running SASTCNN-MRMC, all the four CPU cores' usage rates are high, which means that each CPU core is in a busy state. The SASTCNN-MRMC can improve the execution efficiency rapidly by fully utilizing all the computational power of the CPU. Table 5 compares the running times between SASTCNN-MR and SASTCNN-MRMC on the Hadoop cluster.

Generally speaking, the speed-up ratio on the cluster is inferior to that on a single machine. The reason is that when running on the cluster, there is a large part of time used to transmit data between the cluster nodes, the IO operations are not parallelizable, thus the total speed-up ratio is decreased. Furthermore, the obtained speed-up ratio

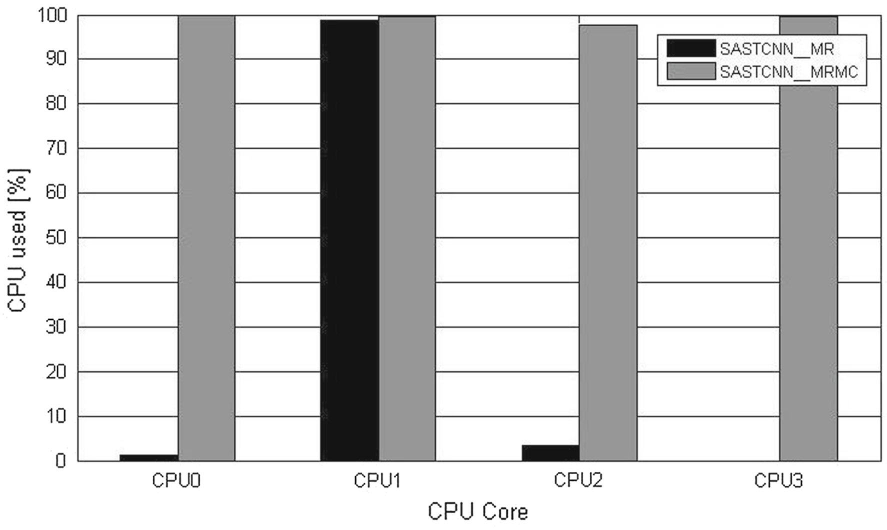

Fig. 14 The CPU usage rates of SASTCNN-MR and SASTCNN-MRMC on a single machine

Table 5 A comparison of the running times between SASTCNN-MR and SASTCNN-MRMC on the Hadoop cluster

Environment	Cluster		
	SASTCNN-MR (ms)	SASTCNN-MRMC (ms)	Speedup
WEIZMANN			
Training	79,485	42,965	1.85
Testing	10,422	4556	2.29
KTH			
Training	643,776	342,434	1.88
Testing	77,478	33,947	2.28

in the training process is inferior to the speed-up ratio in the testing process. This is consistent with the previous experimental results.

7 Conclusion

In this paper we have presented SASTCNN-MR using the Hadoop MapReduce for parallelization of matrix multiplication. SASTCNN-MR was initially implemented on a Hadoop cluster and subsequently on a multi-core computer. The performance of SASTCNN-MR was evaluated on both platforms and experimental results were analyzed from the aspects of accuracy and computation efficiency.

It is an effective attempt to use MapReduce to carry out the parallelization of SASTCNN. However, the work is still far from enough. If we can use the massive amounts of data to study, we may be able to further reveal the work of human visual system. In the aspect of parallelization, the parallel ability of GPU can be used to further improve the computing speed.

Acknowledgements This work was partially supported by NSFC under contract No. 61501451, by the scholarship from China Scholarship Council (CSC) under the Grant CSC No. 201606315022, and by the XMU-NU Joint Strategic Partnership Fund.

References

1. Wang, L., Suter, D.: Recognizing human activities from silhouettes: motion subspace and factorial discriminative graphical model. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–8 (2007)
2. Turaga, P., Chellappa, R., Subrahmanian, V., Udrea, O.: Machine recognition of human activities: a survey. IEEE Trans. Circuits Syst. Video Technol. **18**(11), 1473–1488 (2008)
3. Weinland, D., Ronfard, R., Boyer, E.: Free viewpoint action recognition using motion history volumes. Comput. Vis. Image Underst. **104**(2–3), 249–257 (2006)
4. Weinland, D., Boyer, E., Ronfard, R.: Action recognition from arbitrary views using 3D exemplars. In: Proceedings of IEEE International Conference on Computer Vision (2007)
5. Atmosukarto, I., Ahuja, N., Ghanem, B.: Action recognition using discriminative structured trajectory groups. In: Proceedings of IEEE Winter Conference on Applications of Computer Vision, pp. 899–906 (2015)
6. Yan, Y., Ricci, E., Subramanian, R., Liu, G., Sebe, N.: Multitask linear discriminant analysis for view invariant action recognition. IEEE Trans. Image Process. **23**(12), 5599–5611 (2014)
7. Bulbul, M., Jiang, Y., Ma, J.: DMMs-based multiple features fusion for human action recognition. Int. J. Multimed. Data Eng. Manag. **6**(4), 23–39 (2015)
8. Lawrence, S., Giles, C., Tsoi, A., Back, A.: Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Netw. **8**(1), 98–113 (1997)
9. Moez, B., Franck, M., Christian, W., Christophe, G., Atilla, B.: Spatio-temporal convolutional sparse auto-encoder for sequence classification. In: Proceedings of the 23rd British Machine Vision Conference 2012 (2012)
10. Ranzato, M., Huang, F.J., Boureau, Y., Lecun, L.: Unsupervised learning of invariant feature hierarchies with applications to object recognition. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–8 (2007)
11. Ranzato, M., LeCun, Y.: A sparse and locally shift invariant feature extractor applied to document images. In: Proceedings of International Conference on Document Analysis and Recognition, pp. 1213–1217 (2007)
12. Hsieh, L., Wu, G., Hsu, Y., Hsu, W.: Online image search result grouping with MapReduce-based image clustering and graph construction for large-scale photos. J. Vis. Commun. Image Represent. **25**(2), 384–395 (2014)
13. Han, J., Liu, Y., Sun, X.: A scalable random forest algorithm based on MapReduce. In: Proceedings of 2013 4th IEEE International Conference on Software Engineering and Service Science (ICSESS), pp. 849–852 (2013)
14. Chen, T., Zhang, X., Jin, S., Kim, O.: Efficient classification using parallel and scalable compressed model and its application on intrusion detection. Expert Syst. Appl. **41**(13), 5972–5983 (2014)
15. Tewari, N.C., Koduvely, H.M., Guha, S., Yadav, A.: MapReduce implementation of variational Bayesian probabilistic matrix factorization algorithm. In: Proceedings of IEEE International Conference on Big Data, pp. 145–152 (2013)
16. Nobakht, B., Boer, F.: Multi-core programming. Technol. Found. **104**(4), 430–445 (2006)
17. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for multi-core and multiprocessor systems. In: Proceedings of IEEE International Symposium on High Performance Computer Architecture, pp. 13–24 (2007)
18. Mao, Y., Morris, R., Kaashoek, M.F.: Optimizing MapReduce for multicore architecture. MIT-CSAIL-TR-2010-020 (2010)

19. Jiang, W., Ravi, V.T., Agrawal, G.: A Map-Reduce system with an alternate API for multi-core environments. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 84–93 (2010)
20. Chu, C., Kim, S., Lin, Y., Yu, Y., Bradski, R.: MapReduce for machine learning on multicore. Adv. Neural Inf. Process. Syst. **19**, 281–288 (2006)
21. Kovoor, G., Singer, J., Luján, M.: Building a Java MapReduce framework for multi-core architectures. In: Proceedings of Multiprog, pp. 87–98 (2010)
22. Chen, R., Chen, H., Zang, B.: Tiled-MapReduce: optimizing resource usages of data-parallel applications on multicore with tiling. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, pp. 523–534. ACM (2010)
23. Collobert, R., Kavukcuoglu, K., Farabet, C.: Torch7: A matlab-like environment for machine learning. In: BigLearn, NIPS Workshop (2011)
24. Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., Darrell, T.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the ACM International Conference on Multimedia, pp. 675–678. ACM (2014)
25. Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., Bengio, Y.: Theano: New Features and Speed Improvements. arXiv:1211.5590
26. Yang, J., Zhang, D., Frangi, A.F., Yang, J.: Two-dimensional PCA: a new approach to appearance-based face representation and recognition. IEEE Trans. Pattern Anal. Mach. Intell. **26**(1), 131–137 (2004)
27. Sheng, Z., Shan, Z.: Face recognition by LLE dimensionality reduction. In: Proceedings of International Conference on Intelligent Computation Technology and Automation, vol. 1, pp. 121–123 (2011)
28. Lanaaya, H., Martin, A., Aboutajdine, D., Khenchaf, A.: A new dimensionality reduction method for seabed characterization: supervised curvilinear component analysis. In: Oceans-Europe, vol. 1, pp. 339–344 (2005)
29. Blank, M., Gorelick, L., Shechtman, E., Irani, M., Basri, R.: Actions as space-time shapes. In: Proceedings of Tenth IEEE International Conference on Computer Vision, pp. 1395–1402 (2005)
30. Jhuang, H., Serre, T., Wolf, L., Poggio, T.: A biologically inspired system for action recognition. In: Proceedings of IEEE 11th International Conference on Computer Vision, pp. 1–8. IEEE (2007)
31. Niebles, J.C., Fei-Fei, L.: A hierarchical model of shape and appearance for human action classification. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–8. IEEE (2007)