A Bi-layered Parallel Training Architecture for Large-Scale Convolutional Neural Networks

Jianguo Chen, Kenli Li, *Senior Member, IEEE*, Kashif Bilal, *Member, IEEE*, Xu Zhou, Keqin Li, *Fellow, IEEE*, and Philip S. Yu, *Fellow, IEEE*

Abstract—Benefitting from large-scale training datasets and the complex training network, Convolutional Neural Networks (CNNs) are widely applied in various fields with high accuracy. However, the training process of CNNs is very time-consuming, where large amounts of training samples and iterative operations are required to obtain high-quality weight parameters. In this paper, we focus on the time-consuming training process of large-scale CNNs and propose a Bi-layered Parallel Training (BPT-CNN) architecture in distributed computing environments. BPT-CNN consists of two main components: (a) an outer-layer parallel training for multiple CNN subnetworks on separate data subsets, and (b) an inner-layer parallel training for each subnetwork. In the outer-layer parallelism, we address critical issues of distributed and parallel computing, including data communication, synchronization, and workload balance. A heterogeneous-aware Incremental Data Partitioning and Allocation (IDPA) strategy is proposed, where large-scale training datasets are partitioned and allocated to the computing nodes in batches according to their computing power. To minimize the synchronization waiting during the global weight update process, an Asynchronous Global Weight Update (AGWU) strategy is proposed. In the inner-layer parallelism, we further accelerate the training process for each CNN subnetwork on each computer, where computation steps of convolutional layer and the local weight training are parallelized based on task-parallelism. We introduce task decomposition and scheduling strategies with the objectives of thread-level load balancing and minimum waiting time for critical paths. Extensive experimental results indicate that the proposed BPT-CNN effectively improves the training performance of CNNs while maintaining the accuracy.

Index Terms—Big data, bi-layered parallel computing, convolutional neural networks, deep learning, distributed computing

1 Introduction

In recent years, Deep Learning (DL) techniques have achieved promising results in various domains [1], [2]. Convolutional Neural Network (CNN) algorithm is an important branch of DL. Benefitting from large-scale training datasets and the complex training network, CNN achieves high accuracy and is widely applied in various domains, such as image classification [3], speech recognition [4], and text processing [5]. However, the training process of CNN is very time-consuming, in which large amounts of training samples and iterative operations are required to obtain high-quality weight parameters [6], [7]. It is critical to accelerate the training process and improve the performance of CNN. Cloud computing, high-performance computing cluster, and supercomputing provides strong computing power for various applications [8], [9], [10]. Therefore, it is a critical issue that how to design an effective parallel CNN training model based on distributed computing clusters and address the challenges of data communication, synchronization, and workload balancing, while maintaining high performance and high accuracy.

Numerous enhancements were proposed to accelerate the CNN and DL algorithms by improving the execution environments [8], [11], [12], [13], [14], [15], [16]. The CPU/GPU based methods [8], [12], [13] can perform more arithmetic operations and are suitable for the training of modestly sized DL models. However, a known limitation of the these methods is that the training acceleration is small when the scale of training datasets or DL models exceeds the GPU memory capacity. To process large-scale datasets and DL models, some distributed architecture based solutions were proposed, such as DistBelief [14], Caffe [15], and Tensorflow [16]. Considerable improvements might be obtained by combining the advantages in both aspects and make use of the computing power of multiple machines in a distributed cluster and the high-performance CPUs or GPUs on each machine.

There exists multiple challenges in this regard. First, the entire CNN model contains multiple CNN subnetwork models that are trained in parallel on different machines, which requires synchronization and integration operations. Moreover, high-quality CNN models often require large-scale

- *J. Chen, K. Li, and X. Zhou are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410006, China, and also with the National Supercomputing Center in Changsha, Hunan, Changsha 410082, China. E-mail: cccjianguo@163.com, lkl@hnu.edu.cn, happypanda2006@126.com.*
- *K. Bilal is with COMSATS University Islamabad, Abbottabad 45550, Pakistan, and also with Qatar University, Doha 2713, Qatar. E-mail: kashifbilal@ciit.net.pk.*
- *K. Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha 410006, China, and with the National Supercomputing Center in Changsha, Hunan, Changsha 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561. E-mail: lik@newpaltz.edu.*
- *P.S. Yu is with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607, and also with the Institute for Data Science, Tsinghua University, Beijing 100084, China. E-mail: psyu@uic.edu.*

Manuscript received 7 Mar. 2018, revised 31 Aug. 2018, accepted 15 Oct. 2018, Date of publication 22 Oct. 2018; date of current version 10 Apr. 2019. (Corresponding authors: Kenli Li and Keqin Li.)
Recommended for acceptance by Z. Chen.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2877359

training dataset and a large number of iterations. Hence, an effective parallel mechanism should be carefully designed to minimize the data communication overhead between different iteration steps, tasks in different threads/CPUs/GPUs, and distributed computers. Furthermore, considering the heterogeneity of distributed computing clusters, computing nodes might be equipped with different CPU or GPU structures and have different training speed. How to partition the training dataset into these computers and how many parallel training tasks are started on each computer to maximize the computing power and workload balance of each computer.

In this paper, we aim to address the above challenges and fully utilize the parallel computing capacity of computing clusters and multi-core CPU to accelerate the training process of large-scale CNNs. We propose a Bi-layered Parallel Training-CNN (BPT-CNN) architecture in distributed computing environments. The contributions of this paper are summarized as follows.

- In the outer-layer parallelism, an Incremental Data Partitioning and Allocation (IDPA) strategy is proposed to maximize the workload balance and minimize data communication among computers.
- An Asynchronous Global Weight Updating (AGWU) strategy is proposed to integrate CNN subnetwork models from different computers and to address the synchronization waiting problem during the global weight update process.
- In the inner-layer parallelism, two time-consuming training steps of the CNN model are parallelized on each computer based on task-parallelism, including convolutional layer and local weight training process.
- To achieve thread-level load balancing and critical paths waiting time minimization, we introduce task decomposition and scheduling strategies for CNN training tasks with multi-threaded parallelism.

The rest of the paper is organized as follows. Section 2 reviews the related work. Section 3 presents the BPT-CNN architecture and the outer-layer parallelization process. Section 4 describes the inner-layer parallel training of BPT-CNN. Experimental results and evaluations are discussed in Section 5. Finally, Section 6 concludes the paper with a discussion of future work and research directions.

2 Related Work

Previous works have proposed various hardware designs for CNNs and other deep learning algorithms acceleration [8], [17]. FPGAs have been widely explored as hardware accelerators for CNNs because of their reconfigurability and energy efficiency [18]. In [11], a parallel solution of CNNs was designed on many-core architecture, in which the model is parallelized on a new platform of Intel Xeon Phi Coprocessor with OpenMP. Caffe [15] provides multimedia scientists and practitioners with a clean and modifiable framework for state-of-the-art deep learning algorithms. Chung et al. proposed a parallel Deep Neural Networks (DNNs) training approach for big data on the IBM Blue Gene/Q system, in which issues of regarding programming model and data-dependent imbalances were addressed [18]. In [19], a massively parallel coprocessor was designed as a meta-operator for CNNs, which consists of parallel 2D convolution primitives and programmable units.

To efficiently handle large-scale CNN and big data, outstanding distributed architecture based solutions were implemented in [14], [20]. Adam [21] is an efficient and scalable deep learning training system, optimizing and balancing workload computation and communication through entire system co-design. An energy-efficient reconfigurable accelerator was presented in [22] for deep CNN. To minimize the energy cost of data movement for any CNN shape, a processing row stationary dataflow was introduced to reconfigure the computation mapping of a given shape. In [14], Dean et al. introduced a distributed system (termed DistBelief) for training large neural networks on massive amounts of data. DistBelief uses two complementary types of parallelism: distributed parallel between multiple models and in each model, respectively. In addition, an asynchronous Stochastic Gradient Descent (SGD) procedure was employed to support a large number of model replicas.

Tensorflow [16] is a popular framework for large-scale machine learning on heterogeneous distributed systems. The computation model of Tensorflow is based on dataflow graphs with mutable state, where the graph nodes can be distributed and executed in parallel on different workers, multi-core CPUs, and general-purpose GPUs. In addition, Tensorflow uses a declarative programming paradigm, and developers can focus on the symbolic definition and computation logic instead of the implementation details. However, there are some shortcomings in Tensorflow: (a) TensorFlow attempts to occupy all available GPU memory in the initial phase, which makes the machines deploying the Tensorflow program infeasible to share with other applications, and (b) many high-level operations and interfaces in TensorFlow are nested and chaotically packaged, making it difficult to customize programming. Hence, we study the parallelism idea of the DistBelief and Tensorflow approaches, and implement a bi-layered parallel training architecture for large-scale CNNs by combining the advantages of both distributed computing and CPU/GPU parallel computing.

Comparing with existing efforts, the proposed BPT-CNN architecture in this paper fully utilizes the parallel capacity of both the distributed cluster and multi-core CPU of individual machines. Benefitting from the proposed IDPA and AGWU strategies, we effectively improve the training performance of the CNN model and address the problems of data communication, synchronization, and workload balancing of distributed cluster. Moreover, according to task decomposition and scheduling strategies, BPT-CNN achieves the optimization objectives of thread-level load balancing and waiting time minimization of critical paths.

3 BPT-CNN Architecture for CNNs

3.1 Convolutional Neural Networks

CNN model is one of the most representative network structures of DL technologies and has become one of the hot topics in various fields of science. The common architecture of a CNN network includes two components: a feature extractor and a fully-connected classifier. In a convolutional layer, each batch of the input dataset is analyzed to obtain different abstract features. Given an input X with scale (D_x, H_x, W_x),

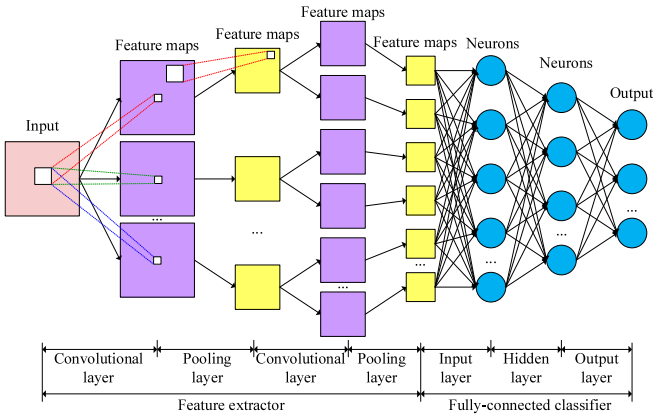


Fig. 1. Example of a CNN architecture.

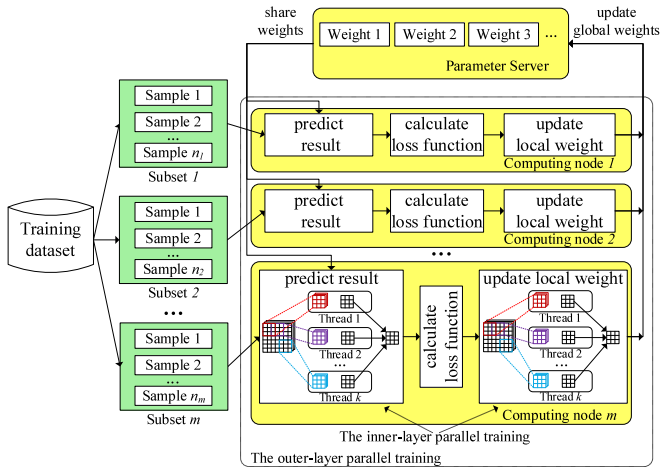


Fig. 2. Bi-layered parallel training architecture for CNNs.

where D_x, H_x, and W_x refer to the depth, height, and width of X. Assuming that a filter with scale (D_f, H_f, W_f) is used in the convolutional layer to extract a feature map A, then $a_{i,j}$ denotes the value of the jth column of the ith row of the current feature map, as calculated in Eq. (1):

$$a_{i,j} = f\left(\sum_{d=1}^{D_f}\sum_{m=1}^{H_f}\sum_{n=1}^{W_f}(w_{d,m,n} \times x_{d,i,j}) + w_b\right), \quad (1)$$

where D_f, H_f, and W_f are the depth, height, and width of the current filter, and $f()$ is an activation function, such as $sigmoid$, $tanh$, or $relu$ function [23].

Pooling layer is utilized on each feature map to reduce the feature dimensions. There exist various pooling methods, i.e., max pooling and mean pooling. Fully-connected layer is a classification layer of CNN, where all output features of convolutional or pooling layers are connected to all hidden neurons with weight parameters. An example of a CNN architecture is illustrated in Fig. 1.

Massive training datasets and iterative training process guarantee the high precision of CNN. However, they become the performance bottleneck when the training network structure is complex and the computing power is insufficient.

3.2 Bi-Layered Parallel Training Architecture

To accelerate the training process of CNNs, we propose a bi-layered parallel training architecture for large-scale CNNs. We describe the distributed computing environment and training process of the BPT-CNN architecture.

3.2.1 BPT-CNN Architecture

BPT-CNN architecture is composed of two main components: (a) an outer-layer parallel training for multiple CNN subnetworks on separate data subsets, and (b) an inner-layer parallel training for each subnetwork. The proposed BPT-CNN architecture is illustrated in Fig. 2.

(1) *Outer-layer parallel training*. A data-parallelism strategy is adopted in the outer-layer parallel training process, where a large-scale dataset is split into multiple subsets and allocated to different computing nodes to be trained in parallel. At the parameter server, the global weight parameters of the entire CNN network are updated depending on the local weights from each training branch. The updated global weight parameters are shared to each machine for the next iterative training.

(2) *Inner-layer parallel training*. The inner layer adopts a task-parallelism strategy to further accelerate the training process of each CNN subnetwork on each computer. Two time-consuming computation tasks are parallelized, including convolutional layer and the local weight training process. Computation tasks on these processes are decomposed depending upon their logical and data dependence, and are executed with multi-threaded parallelism.

3.2.2 Distributed Computing Cluster for BPT-CNN

We construct a distributed computing cluster for the proposed BPT-CNN architecture to efficiently handle massive training datasets and large-scale CNN models. The distributed cluster mainly consists of a main server, several computing nodes with mult-core CPU, and a parameter server, as shown in Fig. 3.

The main server is responsible for CNN training task management as well as data partition and allocation. It copies the CNN training network and allocates them to each computing node. During the parallel training, the main server monitors the training time costs on computing nodes and migrates datasets for the optimization objective of synchronization delay minimization.

On each computing node, samples in the subset are calculated by the corresponding CNN subnetwork, while the network weight parameters are trained as a local weight

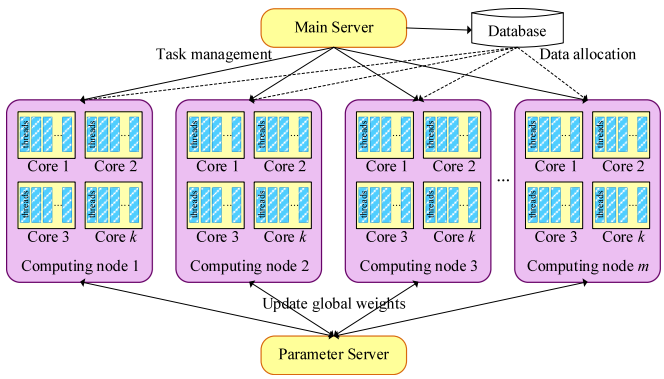


Fig. 3. Structure of the distributed computing cluster for BPT-CNN.

set. The training process on each computer is executed in parallel. In addition, each computer is equipped with a multi-core CPU platform. In an inner-layer parallel training, the training process of each CNN subnetwork is further parallelized using multi-threaded parallelism.

The parameter server collects the trained local weight parameters from each computing node and updates the global weight parameters. Then, the updated global weight set is re-allocated to each computing node for the next epoch of training.

3.3 Outer-Layer Parallel Training of BPT-CNN

In BPT-CNN's outer-layer parallel training architecture, we address critical issues of distributed and parallel computing, including data communication, synchronization, and workload balancing.

3.3.1 Incremental Data Partitioning and Allocation Strategy

Considering the heterogeneity of computing nodes and their different training speed, to maximize the workload balance of the distributed cluster and minimize the synchronization in global weight update process, we propose an Incremental Data Partitioning and Allocation strategy based on heterogeneous sensing. As there are no dependencies between training samples, they can be partitioned and allocated in batches instead of done at once, according to the computing power of the computing nodes. Assume that there are N training samples, m computing nodes in a distributed cluster, and K training iterations are required for the CNN model. Let A ($A < K$) be the number of batches of data partitioning, that is, the entire training dataset is incrementally partitioned and allocated in A times, and each time $\lfloor \frac{N}{A} \rfloor$ new samples are processed.

Initially, we take the first $\lfloor \frac{N}{A} \rfloor$ samples as the training dataset in the first batch. Before the training iteration, we use the constant characteristics of the computing nodes to represent their heterogeneity, i.e., the CPU/GPU frequency is measured. Let μ_j be the CPU/GPU frequency of computing node C_j, and the number of samples that will be partitioned and allocated to C_j is calculated as:

$$n_j^{(1)} = \begin{cases} \left\lfloor \lfloor \frac{N}{A} \rfloor \times \frac{\mu_j}{\sum_{j'=1}^{m} \mu_{j'}} \right\rfloor & 1 \le j \le (m-1), \\ \lfloor \frac{N}{A} \rfloor - \sum_{j'=1}^{m-1} n_{j'}^{(1)} & j = m. \end{cases} \tag{2}$$

After receiving the training samples, each computing node begins the first iteration of training. At the same time, we monitor the execution time of each computing node to complete the iteration and evaluate its actual computing power. Being of the opinion that there might be more applications from different employers executing on the compute nodes, although we can predict the computing power based on the computing node's CPU/GPU frequency, it is more accurate to evaluate its actual computing power by actual execution time. Therefore, after the first training iteration, we can partition the training dataset according to the actual computing power of the machines. Let T_j be the execution time of computing node C_j to train $n_j^{(1)}$ samples in the current iteration, then we can get the average execution time of C_j for a sample as $\overline{t_j} = \frac{T_j}{n_j^{(1)}}$. We collect the execution time of the computing nodes in the current iteration and predict the execution time required by all computing nodes in the next iteration. Note that $\lfloor \frac{N}{A} \rfloor$ new samples will be partitioned and allocated in the ath batch. Namely, there is a total of $\lfloor \frac{N}{A} \rfloor \times a$ samples on the computing nodes. The average execution time of the computing nodes in the ath training iteration is calculated as:

$$T_a = \frac{\lfloor \frac{N}{A} \rfloor \times a \times \overline{t}}{m}, \tag{3}$$

where $\overline{t} = \frac{1}{m} \sum_{j=1}^{m} \overline{t_j}$ is the average execution time for training a sample by any compute node. To minimize the synchronization latency among computing nodes during the global weight update process, we expect all nodes to complete each iteration as close as possible. Assume that n'_j samples on C_j after the ath batch partitioning and allocation, we can calculate the value of n'_j as:

$$n'_j = \frac{T_a}{\overline{t_j}}. \tag{4}$$

Accordingly, we can obtain the number of samples that C_j can accept in the ath batch allocation according to its actual computing power, as calculated as:

$$n_j^{(a)} = \begin{cases} n'_j - \sum_{a'=1}^{a-1} n_j^{(a')} & 1 \le j \le m-1, \\ \lfloor \frac{N}{A} \rfloor - \sum_{j'=1}^{m-1} n_j^{(a)} & j = m. \end{cases} \tag{5}$$

Repeat this process A times until the entire training dataset is partitioned and allocated to the heterogeneous computing cluster. By considering the heterogeneity of computing nodes, each computing node receives a corresponding number of training samples based on its actual computing power. The total number of samples allocated to each computing node C_j is denoted as n_j, and $n_j = \sum_{a=1}^{A} n_j^{(a)}$. The detail steps of the IDPA strategy are described in Algorithm 3.1.

Benefitting from the IDPA strategy, the training dataset is well partitioned and allocated to the computing nodes, allowing them to complete each iteration in same duration, achieving minimal synchronization delay and maximum workload balancing. Moreover, no data migration is required among compute nodes during the training process, thereby no unnecessary data communication overhead is incurred.

Recall that K iterations are required for the CNN model, that is, each sample has an average of K times to train the weight parameter set of the CNN network model. After the A iterations in the data partitioning process, each computing node continues to execute the remaining iterations on the n_j samples. Since these samples are incrementally allocated to the computing nodes, the actual training times of samples in the A iterations are $\frac{N}{A} \times \sum_{a=1}^{A} a = \frac{N(A+1)}{2}$ instead of $N \times A$. Therefore, we should recalculate the remaining iterations ΔK of the training process, as defined below:

$$\begin{aligned} \Delta K &= \left\lfloor \frac{(N \times K) - \frac{N(A+1)}{2}}{N} \right\rfloor \\ &= K - \frac{A}{2} - 1. \end{aligned} \tag{6}$$

The total number of training iterations of the CNN model is $K' = A + \Delta K = K + \frac{A}{2} - 1$. To simplify the expression, we denote K' as K in the remaining context.

Algorithm 3.1. Incremental Data Partitioning and Allocation Strategy

Input:

N: the number of samples in the training dataset;
m: the number of computing nodes in the distributed cluster;
A: the number of batches for data partitioning and allocation;
a: the current batch of data partitioning and allocation.

Output:

n_s: the number list of sample partitioned to the computing nodes.

1: **if** $a = 1$ **then**
2: **for** j from 1 to $m - 1$ **do**
3: $n_j^{(1)} \leftarrow \left\lfloor \frac{\mu_j}{\sum_{j'=1}^{m} \mu_{j'}} \times \lfloor \frac{N}{A} \rfloor \right\rfloor$; n_s.append($n_j^{(1)}$);
4: $n_m^{(1)} \leftarrow \lfloor \frac{N}{A} \rfloor - \sum_{j'=1}^{m-1} n_{j'}^{(1)}$; n_s.append($n_m^{(1)}$);
5: **else**
6: collect training duration T_j from each computing node C_j in the $(a-1)$th iteration;
7: calculate the average training duration $\overline{t}_j \leftarrow \frac{T_j}{n_j^{(1)}}$ and $\overline{t} \leftarrow \frac{1}{m} \sum_{j=1}^{m} \overline{t}_j$;
8: predict the training duration of the ath iteration $T_a \leftarrow \frac{\lfloor \frac{N}{A} \rfloor \times a \times \overline{t}}{m}$;
9: **for** j from 1 to $m - 1$ **do**
10: get the number of samples $n'_j \leftarrow \frac{T_a}{\overline{t}_j}$ for C_j in the ath iteration;
11: $n_j^{(a)} \leftarrow n'_j - \sum_{a'=1}^{a-1} n_j^{(a')}$; n_s.append($n_j^{(a)}$);
12: $n_m^{(a)} \leftarrow \lfloor \frac{N}{A} \rfloor - \sum_{j'=1}^{m-1} n_j^{(a)}$; n_s.append($n_m^{(a)}$);
13: **return** n_s.

3.3.2 Global Weights Updating Strategies

There are massive connections with different weight parameters among all layers in a CNN network. We define these weight parameters as a weight set. We need to collect the training results on each computing node to update the global weight set for the entire CNN network. In this section, we propose two global weight updating strategies for the CNN network. We respectively define the local weight of each CNN subnetwork and the global weight of the entire CNN network as follows.

Definition 1. *Local Weight Set. The weight parameters among all training layers of a CNN training network are denoted as a weight set. At each computing node, the weight set of a CNN subnetwork is defined as the local weight set of the corresponding subnetwork. The local weight set is trained based on the related data subset. In a distributed computing cluster, there is a local weight set on each computing node, which is updated after training a sample.*

Definition 2. *Global weight set. The weight set of the entire CNN network is defined as the global weight set. We provide a parameter server for calculating the global weight set by combining parts or all of the local weight sets. The global weight set is aggregated by each local weight set and shared to all computing nodes for the next epoch of training.*

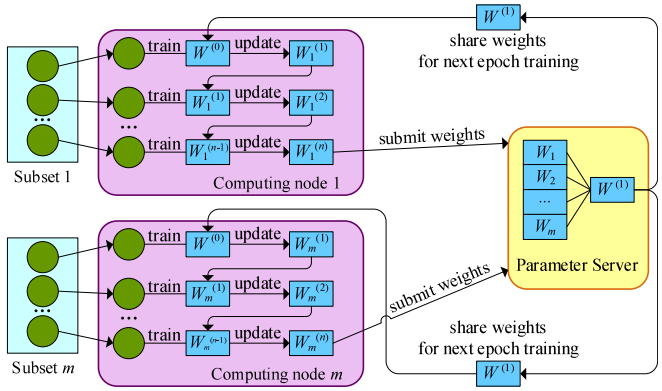


Fig. 4. Synchronous global weight updating strategy. Initially, all computers use the same global weight set $W^{(0)}$ to train the first sample and get the corresponding outputs. At each computer C_j, W^0 is updated to $W_j^{(1)}$ based on the first sample's output. Then, $W_j^{(1)}$ is used to train the second sample and obtain $W_j^{(2)}$. Repeat this step, until all samples on C_j are trained, defining as an epoch of local iteration training.

(1) *Synchronous global weight updating strategy.* We propose a Synchronous Global Weight Updating (SGWU) strategy for BPT-CNN, where each computing node trains all the samples of the current subset and updates the local weight set for an iteration. The local weight sets trained by all computing nodes in the current iteration are gathered at the parameter server, where a new version of the global weight set is generated. The workflow of the SGWU strategy is illustrated in Fig. 4.

Considering that different local weight sets are trained by the corresponding subsets on different computers, having different contributions for the global weight set. We verify the accuracy of each CNN subnetwork after completing an epoch of local iteration training and use it as the contribution of the local weight set. After all computers finish an epoch of local iteration training, the latest local weight set trained on each computer is aggregated to the parameter server to update the global weight set as a new version $W^{(1)}$. The global weight set $W^{(i)}$ for the (i)th epoch of iteration training is defined in Eq. (7):

$$W^{(i)} = \sum_{j=1}^{m} W_j^{(i-1)} \times \frac{Q_j^{(i-1)}}{\sum_{k=1}^{m} Q_k^{(i-1)}}, \tag{7}$$

where $W_j^{(i-1)}$ and $Q_j^{(i-1)}$ are the local weight set and the corresponding accuracy of the CNN subnetwork on computer C_j, which is obtained in the $(i-1)$th epoch of local iteration training.

In a distributed computing cluster, due to the different available computing capabilities, computers need different time costs to execute each training iteration. Let $t_j^{(i)}$ be the execution duration for the (i)th training iteration on computer C_j. The waiting time for synchronization of the entire computing cluster is defined in Eq. (8):

$$\mathbb{T}_{SGWU} = \sum_{i=1}^{K} \sum_{j=1}^{m} \left(\max_{j'=1}^{m} (t_{j'}^{(i)}) - t_j^{(i)} \right), \tag{8}$$

where K is the number of iteration training and m is the number of computing nodes.

(2) *Asynchronous global weight updating strategy.* To address the synchronization problem of SGWU, we propose

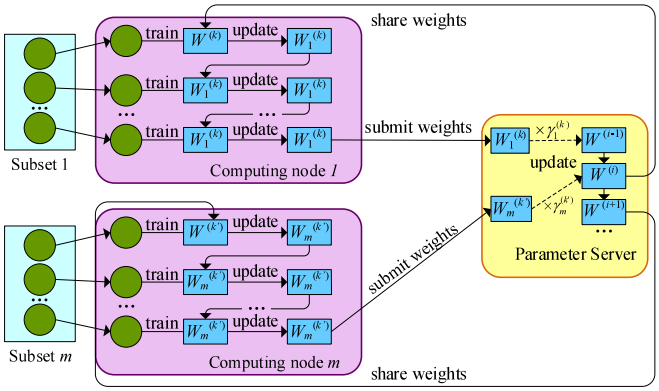


Fig. 5. Asynchronous global weight updating strategy. Each computer C_j uses the current global weight set $W^{(k)}$ to train local samples and update the local weight set $W_j^{(k)}$. Then, $W_j^{(k)}$ is submitted to the parameter server. Note that the global weight set may have been updated from $W^{(k)}$ to $W^{(i)}$ by other computing nodes. Now, based on the local weight set $W_j^{(k)}$, the global weight set $W^{(i)}$ is immediately updated to $W^{(i+1)}$. Repeat this step, until all computing nodes complete the training iterations.

an Asynchronous Global Weight Updating strategy. In AGWU, once a computing node completes a training iteration on the local samples, the updated local weight set is submitted to the parameter server to immediately generate a new version of the global weight set, without waiting for other computing nodes. Compared with SGWU, AGWU can effectively solve the synchronization waiting problem without increasing the communication overhead. The workflow of the AGWU strategy is shown in Fig. 5.

Algorithm 3.2. Asynchronous Global Weight Updating Strategy

Input:
$W_j^{(k)}$: the local weight set from C_j trained based on $W^{(k)}$;
$Q_j^{(k)}$: the accuracy of the CNN subnetwork model on C_j in the (k)th local iteration training.

Output:
$W^{(i)}$: the new version of the global weight set.

1: **for** j' from 1 to m **do**
2: find the version of the global weight set $W^{(k')}$ used for $W_{j'}^{(k')}$;
3: calculate time attenuation factor $\gamma_j^{(k)} = e^{\frac{k}{i-1}} / \sum_{\forall W_{j'}^{k'}, j' \neq j} e^{\frac{k'}{i-1}}$;
4: calculate the update component $\Delta W_j^{k \to i} \leftarrow \gamma_j^{(k)} \times Q_j^{(k)} \times (W_j^{(k)} - W^{(k)})$;
5: update to the global weight set $W^{(i)} \leftarrow W^{(i-1)} + \Delta W_j^{k \to i}$;
6: **return** $W^{(i)}$.

Considering the heterogeneity of computing nodes, according to the IDPA strategy, each computing node may contain different scales of training subset. In addition, due to the different training speeds, each computing node may also submit its local weight set to the parameter server at different time points, and get different versions of the global weight set. For example, for a computing node C_j with n_j samples, we assume that C_j train the local weight set $W_j^{(k)}$ based on based on the version W^k of the global weight set in the current iteration. During the current training iteration of C_j, the global weight set has been updated from $W^{(k)}$ to $W^{(i)}$ by other computing nodes. In this case, the low speed computers train the local weight set based on the old version of the global weight set, while the high speed computers based on the newer version. Denote $(W_j^{(k)} - W^{(k)})$ as the increment between the submitted local weight set $W_j^{(k)}$ and its base version global weight set $W^{(k)}$. Assume that there is another local weight set $W_{j'}^{(i-1)}$ on $C_{j'}$ and it is trained based on the version $W^{(i-1)}$ of the global weight set, where $k < i-1$. It is easy to know that $W_j^{(k)}$ has less impact than $W_j^{(k)}$ in the process of updating $W^{(i)}$. Hence, we can conclude that the local weight sets using the old version of the global weight set have less impact on the global weight updating than those using the new version of the global weight set. Therefore, we adopt a time attenuation factor γ to measure the impact of each local weight set to the current global weight update process. Denote $\gamma_j^{(k)}$ as the time attenuation factor of the local weight set $W_j^{(k)}$ submitted from C_j, as calculated in Eq. (9):

$$\gamma_j^{(k)} = \frac{e^{\frac{k}{i-1}}}{\sum_{\forall W_{j'}^{k'}, j' \neq j} e^{\frac{k'}{i-1}}}, \tag{9}$$

where $(i-1)$ is the latest version of the global weight set, and k is the version of the global weight set that used to train $W_j^{(k)}$.

Since there is no dependence among the training subsets on different computing nodes, the global weight update process does not require the training results from all computing nodes at the same time. Once a local weight set is submitted, the current global weight set $W^{(i-1)}$ is immediately updated to a new version $W^{(i)}$, without waiting for other computing nodes. The ith version of the global weight set is updated as:

$$\begin{aligned} W^{(i)} &= W^{(i-1)} + \Delta W_j^{k \to i} \\ &= W^{(i-1)} + \gamma_j^{(k)} \times Q_j^{(k)} \times (W_j^{(k)} - W^{(k)}), \end{aligned} \tag{10}$$

where $\Delta W_j^{k \to i}$ is the update component from $W_j^{(k)}$, and $Q_j^{(k)}$ is the accuracy of the CNN subnetwork on computer C_j, which is evaluated by the output of the current local iteration training.

After obtaining the updated global weight set $W^{(i)}$, the parameter server shares $W^{(i)}$ to W_j for the next iteration training. Subsequently submitted local weight sets from other computing nodes will update the global weight set based on the latest version. The steps of the AGWU strategy of BPT-CNN is described in Algorithm 3.2.

In comparison with the SGWU strategy, in the AGWU strategy, each computing node independently participates in the global weight update process, so there is no synchronization waiting problem in AGWU. Furthermore, from the perspective of the entire training process, the update of the global weight set depends on the training results of all compute nodes. According to Eqs. (7) and (10), the global weight set is updated based on the local weight set and the corresponding accuracy of the trained mode in both of SGWU and AGWU strategies.

(3) *Data communication of global weight updating*. In BPT-CNN, data communication only incurs between each computing node and the parameter server for the global weight updating and sharing. In both of SGWU and AGWU strategies, the global weight set is updated for every epoch of

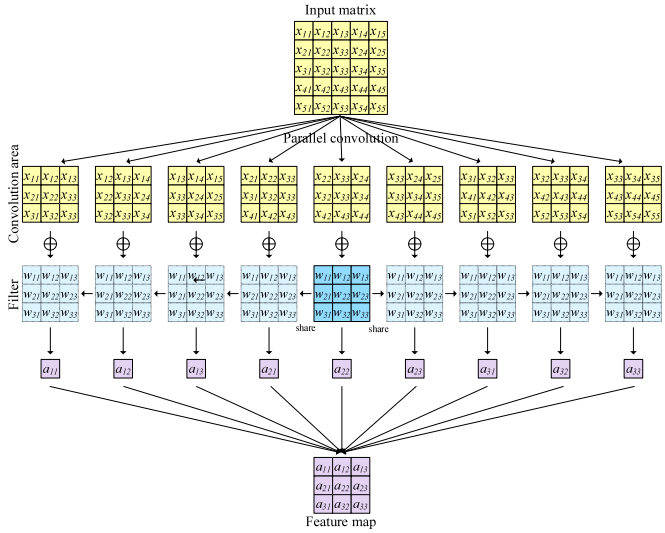


Fig. 6. Parallel convolutional operation based on task-parallelism.

iteration training. Therefore, both strategies produce the same data communication overhead. Denote K as the number of CNN iteration training, data communication $\mathbb{C}_{SGWU}$ in SGWU and $\mathbb{C}_{SGWU}$ in AGWU between the parameter server and all computing nodes is calculated in Eq. (12):

$$\mathbb{C}_{AGWU} = \mathbb{C}_{SGWU} = 2c_w \times m \times K, \tag{11}$$

where c_w is a unit communication cost for transmitting the global weight set between the parameter server and a computing node. For each update of the global weight set, there exist 2 iterations of data communication: (1) submitting the local weight set from a computing node to the parameter server, and (2) sharing the updated global weight set from the latter to the former.

4 Inner-Layer Parallel Training of BPT-CNN

In the inner-layer parallel training of BPT-CNN, we further parallelize the training process for each CNN subnetwork on each computing node. Two time-consuming training steps are parallelized based on task-parallelism, including convolutional layer and the weight training process. In addition, we propose task decomposition and scheduling solutions to realize thread-level load balancing and critical paths waiting time minimization.

4.1 Parallel Computing Models of CNN Training Process

4.1.1 Parallelization of Convolutional Layer

We use the data partitioning method of the input matrix in CNN and extract all convolution areas from the input matrix. Then, by sharing the filter matrix, all convolution areas are convoluted in parallel with the shared filter matrix.

Given an input matrix X with the shape of $(D_x \times H_x \times W_x)$, where D_x, H_x, and W_x are the depth, height, and width of X. Providing a filter parameter matrix F with the shape of $(D_f \times H_f \times W_f)$, a feature map A is generated via convolutional multiplication on X and F. Based on the scales of X and F, the shape of A is calculated as:

$$\begin{aligned} D_a &= D_x = D_f, \\ H_a &= \frac{H_x - H_f + 2P}{S} + 1, \\ W_a &= \frac{W_x - W_f + 2P}{S} + 1, \end{aligned} \tag{12}$$

where D_a, H_a, and W_a are the depth, height, and width of A, respectively. Based on the scales of X, F, and A, we calculate the times K_C of convolutional operations in the current convolutional layer, which will be executed in parallel. K_C is calculated in Eq. (13):

$$K_C = \left(\frac{H_x - H_f + 2P}{S} + 1\right) \times \left(\frac{W_x - W_f + 2P}{S} + 1\right), \tag{13}$$

where S is the stride of the convolutional operation and P is the number of the zero padding, which means appending P laps elements around X with the value of 0.

To execute these K_C operations in parallel, we need to identify the convolution areas of the input matrix X for each task. A convolution area $X[r_{begin} : r_{end},\ c_{begin} : c_{end}]$ of X includes the begin and end rows and columns. In each convolutional operation task, an element-by-element multiplication is executed on $X[r_{begin} : r_{end},\ c_{begin} : c_{end}]$ and F to generate the corresponding element $a_{i,j}$ of A. For each element $a_{i,j}$ in A, location indexes of the convolution area in X is calculated in Eq. (14):

$$\begin{aligned} r_{begin} &= i \times S, \quad r_{end} = r_{begin} + H_f, \\ c_{begin} &= j \times S, \quad c_{end} = c_{begin} + W_f. \end{aligned} \tag{14}$$

After obtaining location indexes of each convolution area, we extract the contents of different convolution areas and perform the related convolutional operations in parallel, without waiting for the end of the previous convolutional operations. These parallel convolutional operations on different areas access the input and filter matrices repeatedly and simultaneously from the same memory without updating the contents. Without data dependence among these tasks, different tasks can access different convolution areas in X simultaneously. An example of the parallel convolutional operation of each CNN subnetwork in BPT-CNN is illustrated in Fig. 6 and the steps of this process are described in Algorithm 4.1.

Algorithm 4.1. Parallel Convolutional Operation of BPT-CNN

Input:
X: The input training dataset;
F: the filter parameter matrix.
Output:
PT_{Conv}: the parallel subtasks of the current convolutional layer.
1: calculate the size (D_a, H_a, W_a) of feature map A in Eq. (12);
2: calculate convolution operation times K_C in Eq. (13);
3: **for** each k in K_C **do**
4: get convolution area $X[r_{begin} : r_{end},\ c_{begin} : c_{end}]$;
5: generate subtask $T_k \leftarrow Conv(X[r_{begin} : r_{end},\ c_{begin} : c_{end}], F, a_{i,j})$;
6: append to the parallel task list $PT_{Conv} \leftarrow T_k$;
7: **return** PT_{Conv}.

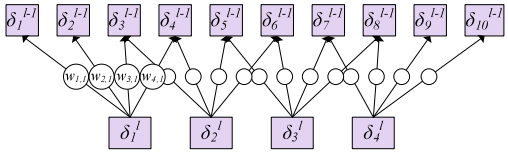


Fig. 7. Example of the loss function computation.

As defined in Eq. (13), the maximum parallelism degree of a convolutional layer is equal to the number of elements of the output feature map, which is computed according to the scale of X and F. The total execution duration $\mathbb{T}_{Conv}$ of a convolutional layer is calculated in Eq. (15):

$$\mathbb{T}_{Conv} = \max_{i=1}^{|A|} \mathbb{T}_i, \tag{15}$$

where $|A| = H_A \times W_A$ is the number of elements in A and $\mathbb{T}_i$ is the execution duration of the ith operation task.

4.1.2 Parallelization of Local Weight Training Process

To distinguish the weight set of the entire CNN network and that of each CNN subnetwork, we respectively define the global weight set and local weight sets in Section 3.3.2. In this section, training process of the local weight set of each CNN subnetwork is parallelized on each computer.

After obtaining the outputs of a CNN subnetwork, the error (loss function) of each layer is evaluated from the output layer to the first convolutional layer using the Back Propagation (BP) method. The Stochastic Gradient Descent process [24], [25] is involved in updating the weight parameters among all layers of the current CNN subnetwork. In the output layer, the square error of all neurons is taken as the objective function of weight training, as defined in Eq. (16):

$$E_x = \sum_{i \in L_{output}} (y'_i - y_i)^2, \tag{16}$$

where E_x denotes the loss function of the input x, and y'_i and y_i are the label and the output of the neuron a_i in the output layer, respectively. The error δ_i of a_i is the inverse of the partial derivative of the error of the input of a_i, as calculated in Eq. (17):

$$\delta_i = -\frac{\partial E_x}{\partial net_i} = -\frac{\partial E_x}{\partial \sum_j w_{ji} x_{ji}}, \tag{17}$$

where x_{ji} is the input of the neuron a_i that connected with a_j, that is, x_{ji} is the output of a_j. w_{ji} is the weight of the connection between neurons a_j and a_i.

Let δ^l be the set of errors of neurons in the lth layer L_l. Based on δ^l, the error set δ^{l-1} of neurons in L_{l-1} is calculated in Eq. (18):

$$\delta^{l-1} = \sum_{i=1}^{N} \delta_i^l \times W_i^{l-1} \oplus f'(net^{l-1}), \tag{18}$$

where W^{l-1} is the weight set of L_{l-1} and net^{l-1} is the weighted input of L_{l-1}, as defined as:

$$\begin{aligned} net^{l-1} &= conv(W^{l-1}, a^{l-2}) + w_b, \\ a_{i,j}^{l-2} &= f^{l-2}(net_{i,j}^{l-2}), \end{aligned} \tag{19}$$

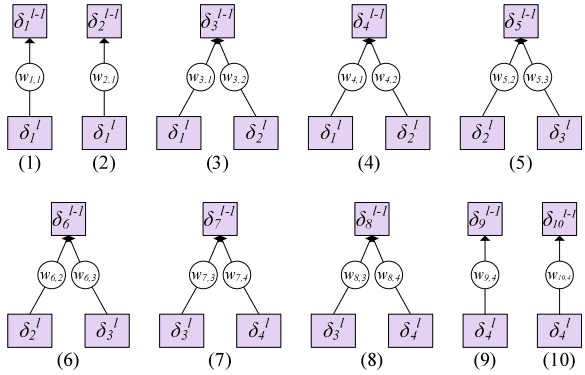


Fig. 8. Example of the loss function parallel computation.

where a^{l-2} is the output matrix of L_{l-2}, consisting of each element $a_{i,j}^{l-2}$. An example of the calculation process of loss function between layers L_l and L_{l-1} is shown in Fig. 7.

We parallelize the process for the loss function calculation, where the errors of neurons in the same layer are computed in parallel. In the convolutional layer, each neuron in the output layer (a feature map) is connected to a part of neurons in the input layer (an input matrix). In such a case, the error calculation of neurons in the previous layer L_{l-1} depends on the results of a part of neurons in the next layer L_l. Hence, we parallelize this process depending on neurons in L_{l-1}. An example of the loss function calculation parallelization is shown in Fig. 8.

After obtaining the error set of neurons in L_l, we calculate the error of each neuron in L_{l-1}. Let $\delta_{i,j}^{l-1}$ be the error component of neuron a_j in L_l for a_i in L_{l-1}, as defined as:

$$\delta_{i,j}^{l-1} = \frac{\partial E_d}{\partial net_{i,j}^{l-1}} = \frac{\partial E_d}{\partial a_{i,j}^{l-1}} \frac{\partial a_{i,j}^{l-1}}{\partial net_{i,j}^{l-1}}, \tag{20}$$

where H_f and W_f are the height and width of the filter parameter matrix F between L_{l-1} and L_l. Based on the error set of neurons, the weight parameters of F are computed subsequently. The gradient of each weight $w_{i,j}$ is calculated in parallel, as defined in Eq. (21):

$$\frac{\partial E_d}{\partial w_{i,j}} = \sum_{h=1}^{H_f} \sum_{m=1}^{W_f} \delta_{h,m}^l \times a_{i+h,j+m}^{l-1}. \tag{21}$$

The gradient of the bias weight w_b is computed in Eq. (22):

$$\frac{\partial E_d}{\partial w_b} = \sum_{a_i \in L_{l-1}} \sum_{a_j \in L_l} \delta_{i,j}^l. \tag{22}$$

Based on the gradient values, each weight $w_{i,j}$ is updated in Eq. (23):

$$w_{i,j} = w_{i,j} - \eta \frac{\partial E_d}{\partial w_{i,j}}, \tag{23}$$

where η is the learning rate of the CNN network.

4.2 Implementation of Inner-Layer Parallel Training

We implement the inner-layer parallel training of BPT-CNN on computing nodes equipped with multi-core CPUs. Based on the parallel models proposed in the previous section, computing tasks of these training phases are decomposed

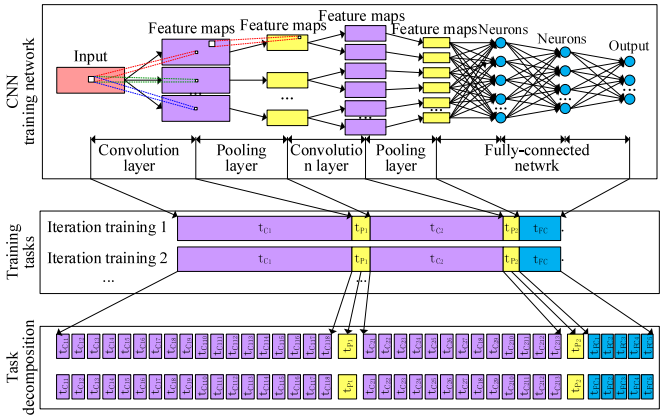


Fig. 9. Task decomposition for a CNN subnetwork.

into several subtasks. The workflow of task decomposition for a CNN subnetwork is illustrated in Fig. 9.

(1) Task priority marking.

According to the logical and data dependence of the decomposed subtasks, a task Directed Acyclic Graph (DAG) is created. With the thread-level load balancing and completion time minimization as the optimization goal, the priorities of tasks in the task DAG are marked. We set a maximum value for the entrance task of the task DAG graph. Then, the priorities of tasks in each level are set according to the tasks' level. Specifically, upstream tasks' priorities are higher than that of downstream tasks, while tasks at the same level have the same priority.

(2) Task scheduling and execution.

Based on the priorities of tasks, we allocate these tasks to threads on the multi-core CPU platform using the priority task scheduling algorithm [26]. Based on the task priorities, tasks of the entire CNN training network are allocated to different threads on the different CPU cores. An example of the task scheduling of the CNN training network with multi-threaded parallelism is illustrated in Fig. 10.

5 Experiments

5.1 Experimental Settings

All of the experiments are conducted on a distributed computing cluster built with 30 high-performance computing nodes, and each of them is equipped with Intel Xeon

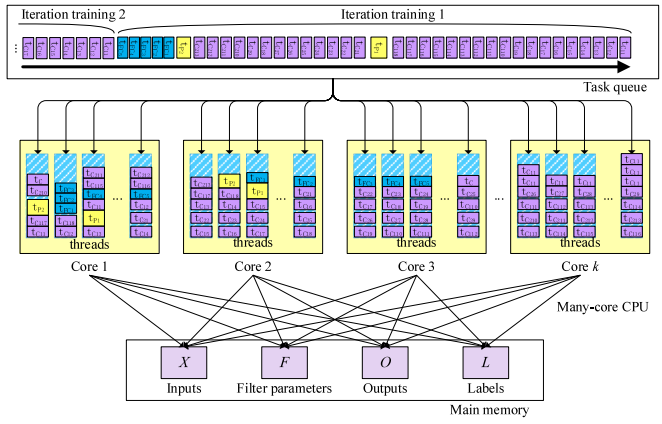
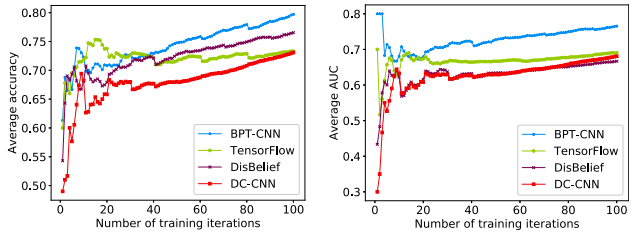


Fig. 10. Task scheduling of a CNN subnetwork on multi-core CPU platform.



(a) Accuracy (b) AUC

Fig. 11. Accuracy evaluation of the comparison algorithms.

Nehalem EX CPU and 48 GB main memory, respectively. Each Nehalem-EX processor features up to 8 cores inside a single chip supporting 16 threads and 24 MB of cache. Comparison experiments are conducted to evaluate the proposed BPT-CNN by comparing with Tensorflow CNN [16], DisBelief [14], and DC-CNN [23] algorithms, in terms of accuracy and performance evaluation. Large-scale public image datasets from ImageNet [3] with 14,197,122 samples are used in the experiments.

Algorithm 4.2. Parallel Task Scheduling of BPT-CNN

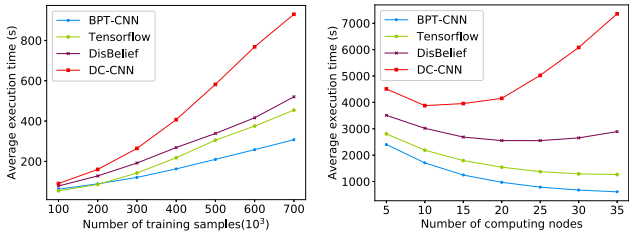
Input:
PTs: The list of parallel tasks PTs for the current CNN network;
Ths: the available threads on the current computing nodes.
Output:
$A(PTs)$: a schedule of tasks in PTs which maximizes thread-level load balancing and minimizes the waiting time of critical paths.

order PTs with priority level $\leftarrow$ orderByPriority(PTs);
while $PTs \neq NULL$ **do**
  take the top element from the task list $t_i \leftarrow PTs$.top();
  get the logical or data depended tasks ts_i of t_i;
  for t_j in ts_i **do**
    if t_j.state $\neq$ complete **do**
      t_i waits and break;
  find thread Th_k from THs with minimal workload;
  call Assignment $A(PTs) \leftarrow (t_i, Th_k)$;
  remove t_i from PTs;
return $A(PTs)$.

5.2 Accuracy Evaluation

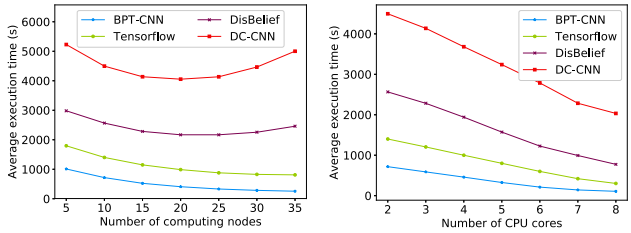
We evaluate the accuracy of BPT-CNN by comparing with Tensorflow, DisBeilef, and DC-CNN. For each algorithm, five-fold experiments on the ImageNet dataset with 100 epoch iterations are conducted and the average values of accuracy and the Area Under the Curve (AUC) are compared. The experimental results of accuracy and AUC of the comparison algorithms are presented in Fig. 11.

As shown in Fig. 11a and 11b, BPT-CNN achieves the similar accuracy with compared algorithms, as well as higher AUC values in most of the cases. The average value of accuracy of BPT-CNN is equal to 0.744, while that of Tensorflow, DisBelief, and DC-CNN is 0.721, 0.722, and 0.639, respectively. Because of the parallel training and global weight updating, BPT-CNN narrows the impact of local overfitting and obtains more stable and robust global network weights. As the epoch of iteration training increases, both of accuracy and AUC of BPT-CNN steadily increases. AUC of BPT-CNN is greater than that of Tensorflow by 5.91 percent, on average,



(a) Different data sizes (b) Different cluster scales

Fig. 12. Total execution time of the comparison algorithms.



(a) Different data sizes (b) Different cluster scales

Fig. 13. Total execution time of the comparison algorithms.

9.56 percent higher than that of DisBelief, and 10.09 percent higher than that of DC-CNN. Therefore, compared with Tensorflow, DisBelief, and DC-CNN, BPT-CNN does not reduce the accuracy of CNNs. Moreover, benefitting from the global weight updating strategy, BPT-CNN achieves more robustness than compared algorithms.

5.3 Performance Evaluation

5.3.1 Execution Time of Comparison Algorithms

The execution time of these algorithms is compared using 100 training iterations in various configurations: different data sizes and computing cluster scales. The comparison of the average execution time of each algorithm in each case is shown in Fig. 12.

As can be seen in Fig. 12a and 12b, the proposed BPT-CNN algorithm achieves higher performance than the compared algorithms in most of the cases. Benefitting from the data-parallelism strategy, when the data size increases, the volume of each partitioned subset on each computer is slightly increased, leading to a slight increase in the average workload of each computer. For example, when the number of training samples increases from 100,000 to 700,000, the execution time of BPT-CNN rises from 62.77 s to 307.35 s, while that of Tensorflow increases from 54.38 s to 454.23 s, and that of DC-CNN sharply increases from 91.21s to 929.74 s. In addition, taking advantage of the IDPA strategy, the proposed BPT-CNN algorithm owns scalability over the compared algorithms. When the scale of the computing cluster expended, the execution time of BPT-CNN and Tensorflow is significantly reduced. Experimental results indicate that BPT-CNN achieves high performance and scalability in distributed computing clusters.

5.3.2 Execution Time Comparison for Fixed Accuracy

Considering the different training architectures of various comparison algorithms, we discuss how these algorithm trade off performance and accuracy with resource consumption. We discuss the training iterations required for each algorithm to achieve different accuracy, and then measure the execution time each algorithm takes under different computing resources. The comparison results are shown in Table 1 and Fig. 13.

TABLE 1
Training Iterations Required by Comparison Algorithm for Different Accuracy

Accuracy	BPT-CNN	Tensorflow	DisBelief	DC-CNN
0.650	7	7	9	12
0.700	18	15	22	28
0.750	42	64	85	147
0.800	97	187	211	-

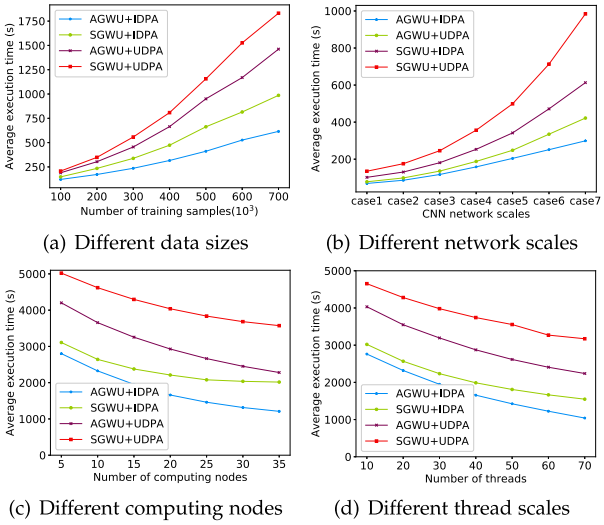
From Table 1, all algorithms use similar iterations to achieve an accuracy of 0.650. However, to achieve higher accuracy, BPT-CNN requires fewer iterations than Tensorflow, DisBelief, and DC-CNN. For example, BPT-CNN requires 42 iterations to achieve an accuracy of 0.750, while Tensorflow uses 64 and DisBelief uses 85, and DC-CNN requires up to 147. In addition, to achieve an accuracy of 0.750, we compare the actual execution times of each algorithm under different numbers of computing nodes and CPU cores, as shown in Fig. 13a and 13b. When the scale of the computing cluster and CPU cores expended, the execution time of BPT-CNN and Tensorflow is significantly reduced. In contrast, the execution time of DisBelief and DC-CNN algorithms is increased when the cluster scale reaches a certain amount (e.g., 25-35), which is caused by the more data communication among the increasing machines. Experimental results indicate that BPT-CNN achieves higher accuracy and performance than other algorithms using the same computing resource. Moreover, when the scale of computing nodes and CPU cores increases, the performance benefits of BPT-CNN is more noticeable.

5.3.3 Execution Time of BPT-CNN with Different Strategies

To evaluate the effectiveness of the IDPA strategy, we perform the same work using the Uniform Data Partitioning and Allocation (UDPA) strategy. The average execution time of BPT-CNN with different strategies is presented in Fig. 14.

In Fig. 14a, 7 different scales of CNN network are constructed in the experiments, as described in Table 2. Here "layers(Conv)" and "filters(Conv)" denote the number of the convolutional layer and that of filters at each layer, respectively. "layers(FC)" and "neurons(FC)" denote the number of layers in the fully-connected layers and number of neurons in each layer, respectively.

By comparing strategies AGWU and SGWU, the execution time of BPT-CNN using AGWU is obviously lower than SGWU in most cases. In AGWU, because of the asynchronous update of the global weight set, each computing node uses the minimum time to wait for the global weight update and trains almost continuously. In addition, by comparing data partitioning strategies IDPA and UDPA, benefitting from the incrementa data partitioning, the workload of computing nodes stays well balanced, which further shortens the waiting time among different machines. As shown in each case in Fig. 14, the execution time of BPT-CNN with IDPA is significantly lower than that with UDPA strategy. Hence, BPT-



(a) Different data sizes (b) Different network scales

(c) Different computing nodes (d) Different thread scales

Fig. 14. Execution time of BPT-CNN with different strategies.

CNN using AGWU+IDPA strategies exhibits the most efficient performance against other cases. Moreover, with the increase of data size or CNN network scale, the execution time of BPT-CNN using AGWU+IDPA maintains a slow rise.

5.4 Data Communication and Workload Balancing

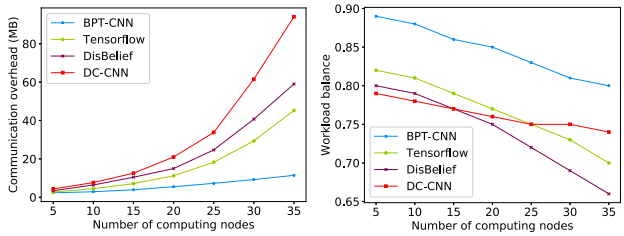
We evaluate the proposed BPT-CNN architecture in the view of data communication overhead and workload balancing by comparing with Tensorflow, DisBeilef, and DC-CNN algorithms. 600,000 training samples are used in the experiments, and the number of computing nodes increases from 5 to 35 in each case. Experiment results of data communication and workload balancing are shown in Fig. 15.

It is clear from Fig. 15a and 15b that, in most cases, BPT-CNN owns significant workload balancing and lower data communication costs than other algorithms. Due to the use of the IDPA strategy in BPT-CNN, there is only communication overhead between the computing nodes for transmitting local/global weight parameters, and no training sample migration is required. Hence, as the number of computing nodes increases from 5 to 35, the communication overhead of BPT-CNN slowly increases from 2.35 MB to 11.44 MB. In contrast, due to dynamic resource scheduling, Tensorflow generates communication overhead from 2.73 MB for 5 computers to 45.23 MB between 35 computers. Moreover, to achieve workload balancing, DisBelief and DC-CNN use data migration operations during training, which results in heavy communication overhead between computers.

We compare the workload balance of each algorithm under different scales of the computing cluster, as shown in Fig. 15b. Our BPT-CNN architecture considers the heterogeneity of compute nodes and allocates corresponding workloads based on the actual computing power of each compute node. Hence,

TABLE 2
Different Scales of CNN Network Used in the Experiments

Scales	case1	case2	case3	case4	case5	case6	case7
layers(Conv)	2	4	6	8	8	10	10
filters(Conv)	4	4	8	8	10	10	12
layers(FC)	3	3	5	5	7	7	7
neurons(FC)	500	1000	1500	1500	2000	2000	2000



(a) Impact of data size (b) Impact of cluster scale

Fig. 15. Comparison on data communication and workload balancing.

as the scale of the cluster increases, BPT-CNN achieves a stable workload balance, keeping between 0.89 and 0.80. In contrast, without heterogeneity-aware data allocation, the workload of other comparison algorithms is not as balanced as BPT-CNN. The unbalanced workload further leads to long waiting time for synchronization and more execution time for the entire CNN network. Experimental results demonstrate that BPT-CNN significantly improves the workload balance of the distributed computing cluster with acceptable communication overhead.

6 Conclusions

This paper presented a bi-layered parallel training architecture to accelerate the training process of large-scale CNNs. In the outer-layer parallel training, the performance of the entire CNN network is significantly improved based on data-parallelism optimization, where the issues of data communication, workload balance, and synchronization, are well addressed. In the inner-layer parallelism, the training process of each CNN subnetwork is further accelerated using task-parallelism optimization. Extensive experimental results on large-scale datasets indicate that the proposed BPT-CNN effectively improves the training performance of CNNs in distributed computing clusters with minimum data communication and synchronization waiting.

Acknowledgments

This research is partially funded by the National Key R&D Program of China (Grant No. 2016YFB0200201), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the International Postdoctoral Exchange Fellowship Program (Grant No. 2018024), and the China Postdoctoral Science Foundation funded project (Grant No. 2018T110829). This work is also supported in part by NSF through grants IIS-1526499, IIS-1763325, CNS-1626432, and NSFC 61672313.

References

[1] A. Coates, B. Huval, T. Wang, D. J. Wu, and A. Y. Ng, "Deep learning with cots hpc systems," in *Proc. 30th Int. Conf. Int. Conf. Mach. Learn. - Vol. 28*, 2013, pp. 1337–1345.

[2] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proc. 17th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2011, pp. 69–77.

[3] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and F. Li, "Imagenet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[4] G. Heigold, V. Vanhoucke, A. Senior, P. Nguyen, M. Ranzato, M. Devin, and J. Dean, "Multilingual acoustic models using distributed deep neural networks," in *Proc. IEEE Int. Conf. Acoust. Speech Signal Process.*, 2013, pp. 8619–8623.

[5] L. Cavigelli, D. Gschwend, C. Mayer, S. Willi, B. Muheim, and L. Benini, "Origami: A convolutional network accelerator," in *Proc. 25th Edition Great Lakes Symp. VLSI*, 2015, pp. 199–204.
[6] L. Song, Y. Wang, Y. Han, X. Zhao, B. Liu, and X. Li, "C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization," in *Proc. 53nd ACM/EDAC/IEEE Des. Autom. Conf.*, 2016, Art. no. 123.
[7] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Proc. 24th Int. Conf. Neural Inf. Process. Syst.*, 2011, pp. 693–701.
[8] S. Fan, J. Fei, and L. Shen, "Accelerating deep learning with a parallel mechanism using cpu+mic," *Int. J. Parallel Program.*, vol. 46, no. 4, pp. 660–673, Aug. 2018.
[9] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang, "Training large scale deep neural networks on the intel xeon phi many-core coprocessor," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1622–1630.
[10] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing, "More effective distributed ml via a stale synchronous parallel parameter server," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst. - Vol. 1*, 2013, pp. 1–9.
[11] J. Liu, H. Wang, D. Wang, Y. Gao, and Z. Li, "Parallelizing convolutional neural networks on intel many integrated core architecture," in *Proc. Int. Conf. Archit. Comput. Syst.*, 2015, pp. 71–82.
[12] A. A. Huqqani, E. Schikuta, S. Ye, and P. Chen, "Multicore and gpu parallelization of neural networks for face recognition," *Procedia Comput. Sci.*, vol. 18, pp. 349–358, 2013.
[13] D. Strigl, K. Kofler, and S. Podlipnig, "Performance and scalability of gpu-based convolutional neural networks," in *Proc. Euromicro Conf. Parallel Distrib. Netw.-Based Process.*, 2010, pp. 317–324.
[14] J. Dean, G. Corrado, and R. Monga, et al., "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst. - Vol. 1*, 2012, pp. 1223–1231.
[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, 2014, pp. 675–678.
[16] M. Abadi, A. Agarwal, and P. Barham, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, no. 5, pp. 725–730, 2016.
[17] M. Mohammadi, A. Krishna, N. S., and S. K. Nandy, "A hardware architecture for radial basis function neural network classifier," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 3, pp. 481–495, Mar. 2018.
[18] I.-H. Chung, T. N. Sainath, B. Ramabhadran, and M. P., et al., "Parallel deep neural network training for big data on blue gene/q," *IEEE Tran. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1703–1714, Jun. 2017.
[19] M. Sankaradas, V. Jakkula, and S. Cadambi, "A massively parallel coprocessor for convolutional neural networks," in *Proc. 20th IEEE Int. Conf. Appl.-Specific Syst. Architectures Processors*, 2009, pp. 53–60.
[20] J. Bilski and J. Smolag, "Parallel architectures for learning the rtrn and elman dynamic neural networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2561–2570, Sep. 2015.
[21] T. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project adam: Building an efficient and scalable deep learning training system," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 571–582.
[22] Y. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-state Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.
[23] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *Proc. 37th Annu. Int. Symp. Comput. Archit.*, 2010, pp. 247–257.
[24] Q. V. Le, J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, and A. Y. Ng, "On optimization methods for deep learning," in *Proc. 28th Int. Conf. Int. Conf. Mach. Learn.*, 2011, pp. 265–272.
[25] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Proc. 23rd Int. Conf. Neural Inf. Process. Syst. - Vol. 2*, 2010, pp. 2595–2603.
[26] L. Zhang, K. Li, C. Li, and K. Li, "Bi-objective workflow scheduling of the energy consumption and reliability in heterogeneous computing systems," *Inf. Sci.*, vol. 379, pp. 241–256, 2017.

Jianguo Chen received the PhD degree from the College of Computer Science and Electronic Engineering, Hunan University, China. He was a visiting PhD student at the University of Illinois at Chicago from 2017 to 2018. He is currently a postdoctoral with the University of Toronto and Hunan University. His major research interests include parallel computing, cloud computing, machine learning, data mining, bioinformatics and big data.

Kenli Li received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. He is currently a full professor of computer science and technology with Hunan University and director of National Supercomputing Center in Changsha. His major research interests include parallel computing, high-performance computing, grid and cloud computing. He is an outstanding member of CCF and a senior member of the IEEE.

Kashif Bilal received the PhD degree from North Dakota State University. He is currently a postdoctoral researcher with Qatar University, Qatar. His research interests include cloud computing, energy efficient high speed networks, and robustness. He is awarded CoE Student Researcher of the year 2014 based on his research contributions during his doctoral studies at North Dakota State University. He is a member of the IEEE.

Xu Zhou received the PhD degree from the Department of Information Science and Engineering, Hunan University, in 2016. She is currently a postdoctoral with the the Department of Information Science and Engineering, Hunan University, Changsha, China. Her research interests include parallel computing and data management.

Keqin Li is a SUNY distinguished professor of computer science with the State University of New York. His current research interests include parallel computing and high-performance computing, distributed computing, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, and cyber-physical systems. He has published more than 590 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is a fellow of the IEEE.

Philip S. Yu received the BS degree in electrical engineering from National Taiwan University, the MS and PhD degrees in electrical engineering from Stanford University, and the MBA. degree from New York University. He is a distinguished professor in computer science with the University of Illinois at Chicago and also holds the Wexler chair in information technology. His research interest is on big data, including data mining, data stream, database and privacy. He has published more than 1,100 papers in refereed journals and conferences. He received or has applied for more than 300 US patents. He is a fellow of the ACM and the IEEE.