An Efficient Method for Training Deep Learning Networks Distributed

Chenxu WANG*†, Member, Yutong LU†††, Zhiguang CHEN††††, Nonmembers, and Junnan LI†, Member

SUMMARY Training deep learning (DL) is a computationally intensive process; as a result, training time can become so long that it impedes the development of DL. High performance computing clusters, especially supercomputers, are equipped with a large amount of computing resources, storage resources, and efficient interconnection ability, which can train DL networks better and faster. In this paper, we propose a method to train DL networks distributed with high efficiency. First, we propose a hierarchical synchronous Stochastic Gradient Descent (SGD) strategy, which can make full use of hardware resources and greatly increase computational efficiency. Second, we present a two-level parameter synchronization scheme which can reduce communication overhead by transmitting parameters of the first layer models in shared memory. Third, we optimize the parallel I/O by making each reader read data as continuously as possible to avoid the high overhead of discontinuous data reading. At last, we integrate the LARS algorithm into our system. The experimental results demonstrate that our approach has tremendous performance advantages relative to unoptimized methods. Compared with the native distributed strategy, our hierarchical synchronous SGD strategy (HSGD) can increase computing efficiency by about 20 times.

key words: deep learning, distributed training, hierarchical synchronous stochastic gradient descent, data-parallelism

1. Introduction

Deep learning (DL) has developed rapidly in recent years and has been widely used in the fields of automatic driving [1], natural language processing [2], biomedical technology [3], computer vision [4], etc. In order to obtain higher accuracy, researchers have developed deeper DL models and bigger datasets, such as VGG-Net [5] and ImageNet [6]; however, these approaches can make the training process take an impractically long time. Although the computing power of high-performance devices such as GPGPUs, KNLs, and FPGAs rapidly increasing, it is still difficult to train DL models quickly enough using a single computing card. There is thus an urgent need for a distributed implementation of the training process.

In order to speed up the training process, many scientists have expended great efforts to distribute the training process across many computing nodes. DistBelief [7] successfully scales the training of deep networks across tens of thousands of CPU cores, using data parallelism and an asynchronous stochastic gradient descent method with a parameter server. FireCaffe [8] distributes the deep neural network (DNN) training process across a cluster of GPUs. This approach uses reduction trees to efficiently communicate parameters, achieving near-linear acceleration of DNN training on GPU clusters. You [9] trains DNN models on KNL and GPU clusters. They redesign four efficient algorithms for HPC clusters, achieving 91.5% weak scaling efficiency on 64 nodes.

High performance computing cluster like supercomputers are a class of computers with the most powerful computing ability, plenty of storage resources, and efficient interconnection. For example, the TH-2 was the fastest supercomputer in the world from 2013 to 2015 according to the TOP500 lists. There are about 8000 computing nodes in TH-2, each of which is equipped with 24 cores and 64 GB memory; thus, TH-2 has 192000 cores and 1.34 PB memory in total. The superior performance of these high performance computing clusters makes them an excellent platform for large-scale DL training.

In this paper, we develop a novel method to efficiently train large-scale DL networks distributed. In order to improve the efficiency of the computing process, we propose a hierarchical synchronous structure. In the first level, we integrate multiple models into one node and interconnect these model sets across multiple nodes. The parameters are first synchronized within a node, then synchronized between nodes. Moreover, we design a two-step communication process based on a hierarchical structure. In the first step, the gradients are aggregated together through shared memory operations. In the second step, the results of the first step are communicated by performing ‘all-reduce’ operations among all nodes. Subsequently, we optimize the I/O and data layout of this hierarchical distributed architecture to improve the efficiency of reading data from disks through high concurrency and continuous reading. Moreover, we also introduce a large batch size training algorithm (LARS) to further improve scalability.

The remainder of this paper is organized as follows. The related works are briefly presented in Sect. 2. In Sect. 3, we introduce some necessary background knowledge. In Sect. 4, we present the design of our proposed method. In Sect. 5, we empirically evaluate our approach. Finally, we conclude this work in Sect. 6.
2. Related Works

The training of DL models is a highly computationally intensive process that consumes a huge amount of computing resources and an extremely large amount of time. Many existing works have aimed to speed up the training tasks by training large-scale DL models across multiple high-performance nodes. These works accelerate the training tasks in terms of three main aspects: namely, computing, communication, and I/O.

2.1 Computing Optimization

Many efforts about paralleling computing have been made to improve the computing speed of the training process, and the parallel methods mainly consist of model parallelism and data parallelism. Model parallelism divides a complete DL model into several parts and ensures that a node is responsible for each corresponding subsection. Data parallelism distributes a batch of data to multiple nodes, each of which contains a complete model to compute the sub-gradients belonging to it. A number of studies [7], [10]–[12] use model and data parallelism jointly to accelerate the training speed, while others [13]–[16] use only data parallelism to assign tasks across nodes. Compared to data parallelism, the model parallelism approach involves dividing the network and communicating the sub-results between nodes, leading to low computational efficiency and high communication overhead. As a result, data parallelism is the most widely used parallel method, as well as the one adopted in this paper.

Some prior works have accelerated DL applications by speeding up the performance of the convolution kernel or GEMM based on architectural characteristics. Evangelos [17] introduces direct convolution kernels for Xeon and Xeon Phi systems; these kernels are implemented via a dynamic compilation approach, and achieve performance close to the theoretical peak. swDNN [18] optimizes convolution kernels on SW26010 processors; these are a kind of many-core processor that provide computing power for Sunway TaihuLight. The swDNN performs a systematic optimization, such as the organization of convolution loops, blocking techniques, register data communication schemes, and the reordering strategy for the two instructions pipelines. Chen [19] optimizes the memory usage in DNN training on GPUs and cuts down memory usage up to 59X, using multiple technologies to reduce both memory consumption and time overhead.

2.2 Communication Optimization

Some previous works have been designed to optimize the communication between nodes in order to accelerate the training of DL applications. A parameter server scheme is usually used to apply parameter synchronization. However, all nodes are designed to send gradients to the server node, which can cause a serious bandwidth contention issue. Performing reduction tree operations is thus both more efficient and scalable than the traditional parameter server approach. Several prior works [8], [20]–[23] all implement ‘allreduce’ operations, customized by cluster interconnect features, to optimize the transmission process.

Compressing gradients without compromising accuracy is also an effective way to reduce communication overhead. Frank Seide et al. [24] show that we can use 1-bit gradients to train DL models, thereby reducing data-exchange bandwidth for data-parallel SGD without incurring a loss of accuracy. AdaComp [25] compresses gradients in the distributed training of DL models using gradient sparsity and quantization. It automatically tunes the compression rate based on the localized selection of gradient residues and local activity. Deep compression [26] is used to compress neural networks without affecting accuracy by pruning the unimportant connections, quantizing the network using weight sharing, and applying Huffman coding. Moreover, Deep Gradient Compression [27] reduces the communication bandwidth required and improves the scalability of distributed training. It compresses the gradients by employing momentum correction, local gradient clipping, momentum factor masking and warm-up training, and thereby reduces the gradients by 270-600X.

2.3 I/O Optimization

The I/O subsystem is responsible for reading hard disk data into memory for calculation purposes, and may be highly inefficient on large-scale systems. LMDBIO [28] presents a scalable I/O plugin for DL frameworks designed to ease the interaction with the process scheduling system and the network-based parallel file system. In order to reduce the I/O overhead, S-Caffe [29] provides DL frameworks with parallel reading capabilities in order to take advantage of parallel file systems such as Lustre. swCaffe [20] improves the aggregated bandwidth of disk arrays by adjusting the data layout, enabling it to better fit the hardware architecture.

3. Background

In this subsection, we will introduce some background knowledge of DL first; Then some parallel training techniques are demonstrated; At last, three communication methods are described.

3.1 Deep Learning Networks

DL learns by mimicking the neural activities of human brains, which is an approach that has shown great promise in many practical domains. Convolutional neural networks (CNNs) are one of the most important network in the DL context. A CNN is a hierarchical structure containing a data processing layer, convolution layers, polling layers, fully connected layers, softmax layer, etc. Convolution operation
is the key to CNN: this is a process of multiple kernels which convolutes with multiple input feature maps for feature extraction. It is also a highly computationally demanding operation that usually accounts for the majority of computing time.

In this paper, we mainly use the stochastic gradient descent (SGD) algorithm to train DL networks, which contains a lot of iterations. Each iteration comprises two computationally intensive passes: the Forward pass and the Backward pass. During the Forward pass, part of the training data are fed into the DL network in order to calculate the error terms. During the Backward pass, the error terms are used to compute the gradients, and each connection weight is updated with the gradient. The training lasts for multiple epochs (during which the entire training dataset is processed) until the validation set error converges to a desired value.

3.2 Parallel Technology

There are two main parallel methods involved in deploying SGD across multiple nodes for distributed training purposes: namely, model parallelism and data parallelism. In the former method, the DL network is partitioned into pieces, each of which is assigned to an individual node. In the latter method, moreover, the dataset is partitioned equally across multiple nodes, each of which has a complete copy of the DL network. Data parallelism only sends gradients at the end of each iteration, while model parallelism should send both gradients and intermediate results to other nodes. Compared with data parallelism, model parallelism suffers from serious bandwidth problems; on the other hand, partitioning the DL network results in a smaller scale of computing, leading to a low hardware device usage. Thus, data parallelism is more suitable for the distributed training of DL networks, and we accordingly choose data parallelism as the parallel method of our design.

3.3 Communication Method

Currently, distributed training of DL networks is usually conducted via synchronous stochastic optimization, asynchronous stochastic optimization, and hybrid asynchronous stochastic optimization. In the synchronous method, the parameters do not update until all nodes have finished their calculating of gradients, which introduces idle time that is wasted on waiting for straggling workers. By contrast, the asynchronous method updates the parameters all at once after a node computes its gradients; however, this comes at the cost of staleness caused by asynchrony. Compared to this hybrid asynchronous stochastic optimization, our method adopts a two-level structure too. However, the two-level structure of our method both use synchronous approach, and we place the workers in the same subsets of the first level in the same computing nodes to make full use of the computing and storage capabilities of high-performance computing hardware.

4. Proposed Method

In this section, we mainly improve the efficiency of distributed training DL networks from four aspects. First, we design a hierarchical synchronous SGD to make full use of the high-performance hardware resources; Second, we use a two-level synchronization scheme to save communication time through shared memory copy; Third, we improve the
I/O read efficiency by the means of high concurrency and continuous reading; Fourth, we integrate the LARS algorithm into our system to further improve the training efficiency. At the last, we introduce the distributed deep learning framework developed based on our proposed method.

### 4.1 Hierarchical Synchronous SGD

In this paper, we adopt the data parallelism strategy, and some features of high performance computing cluster can be used to optimize this strategy. Computing clusters are usually composed of multiple high-performance devices, which have very strong computing power along with many computing cores, large storage space, and high memory bandwidth. The scale of DL networks and the amount of data reading each time are much far less than the memory of these high-performance computing devices. For example, the memory of the GTX Titan is 12 GB, and the scale of VGG16 is 518M. Even considering the memory occupied by intermediate variables, a Titan can still hold multiple VGG16 models in a Titan simultaneously. On the other hand, existing DL training frameworks place only one training model in each computing node, leading to low computational efficiency. DeepBench [31] is an open-source benchmarking tool that measures the performance of the basic operations involved in training deep neural networks. From its results, we can conclude that the true computing speed is much worse than the peak speed of the corresponding hardware device. Consequently, we can integrate multiple training process into a single node, and thereby form a hierarchical synchronous SGD to train DL networks in a distributed manner.

As shown in Fig. 3, the hierarchical synchronous SGD (HSGD) is still essentially a synchronous SGD algorithm. Unlike the traditional synchronous SGD method, HSGD divides the synchronization process into two parts. In the first part, HSGD integrates multiple DL training workers into one computing node based on the node’s computational resources. Each training worker is a complete training process, including initialization, forward pass, backward pass, communication, and update. After all workers in a node complete the backward pass, the gradients are synchronized across the workers in this node. Notably, the synchronization process of this part takes place in shared memory, which saves time for communication. In the second part, HSGD interconnects multiple computing nodes of this kind, which comprise multiple training workers. After the communication within the node is synchronized, HSGD synchronizes the gradient between nodes. In this part, we use the reduction trees to make full use of the interconnection potential of clusters.

There are three main benefits of using HSGD. The first one is that it improves the computational efficiency of hardware devices. The HSGD integrates multiple training processes into a single computing node, meaning that each node can handle more computing tasks, while the amount of data processed in one iteration also increases. This will help improve the computational efficiency of high-performance devices. The second is that it reduces bandwidth contention. HSGD migrates the process of parameter synchronization within a node to shared memory, which can reduce the number of nodes required for data synchronization. Thus, HSGD can reduce bandwidth contention compared to traditional synchronous SGD. HSGD can also alleviate the negative effects of stragglers. The duration of the synchronization method depends on the slowest worker. The computing power of each worker is part of the overall computing power of the hardware device, meaning that the difference in computing power between workers is smaller than the difference in computing power between nodes. On the other hand, the amount of data allocated to each worker in HSGD is much less than the traditional synchronization SGD. As a result, the negative effects of stragglers in HSGD are smaller than the case for traditional synchronization SGD.

#### 4.2 Two-Level Synchronization Scheme

There are two kinds of interconnections between workers in HSGD: the interconnection of workers in the same computing nodes, and the interconnection of workers between nodes. In the first type of interconnection, the synchronization of parameters is only operations on the shared memory, and does not require incurring bandwidth resource costs. In the second interconnection, we should spend bandwidth resources to communicate parameters across different nodes. As a result, we divide the parameter synchronization of HSGD into two steps: first, we summarize the results of the workers in the same computing node; then, we select a root worker in each computing node to communicate the gradients across different nodes.

The gradient synchronization process within a node is shown from line 8 to line 11 of Algorithm 1. We need to calculate the average of the gradients that are calculated by

![Fig. 3](image-url) The schematic diagram of hierarchical synchronous SGD.
**Algorithm 1** HSGD algorithm with two-level parameter synchronization

**Input:** dataset $X$, batchsize $B$, the number of computing nodes $N$, the number of worker in a computing node $K$, parameters $W$

1. for iteration $t = 0, 1, \ldots, \max_{\text{iter}}$ do
2. launch $K$ DL network workers
3. for each computing node $n$ do
4. for each worker $k$ in a computing node do
5. read $\nabla f(x_k; W)$
6. calculate $\nabla f(x_k; W)^T$
7. end for
8. end for
9. reduction task division in computing nodes
10. complete parameter synchronization within a computing node
11. root worker in Computing nodes all-reduce $G^t$
12. complete parameter synchronization between computing nodes
13. $W^{t+1} \leftarrow W^t - \eta W$
14. end for
15. end for

![Fig. 4](image-url) The process of reduction inside a computing node.

The training of DL uses three steps to iteratively processes the data: 1) reading data from the I/O subsystem; 2) forward propagation for identifying deviation errors; 3) backward propagation for correcting deviations. This process is conducted over a large number of iterations in order to ensure higher accuracy. As the parallel scale and the computing power of the hardware increases, the time required for the second and third steps is greatly reduced, making data reading an important factor in the overall system. As a result, an efficient parallel I/O subsystem is critical to distributed DL training systems.

High-performance computing cluster storage systems are mostly friendly to distributed parallel data reading. Take the TH-2 storage system we use as an example. It exploits a hybrid two-layer storage architecture, in which the upper layer is composed of I/O nodes and the bottom layer is composed of storage servers. The local storage devices of I/O nodes are SSDs, and the storage devices of storage servers are SSDs for metadata servers and traditional magnetic storage devices for object storage servers. They are managed by a file system named H2FS, which is based on Lustre file system. The local storage of SSD devices in I/O node provide high concurrency performance and disk arrays in shared storage provide enough storage space and sustaining performance.

Accordingly, based on parallel data reading friendly storage system, we redesign the parallel I/O subsystem. Each computing node has multiple readers and a prefetch queue. All readers read a partial batch of data to the prefetch queue, after which the workers in this computing node feed the data in the prefetch queue to their networks for training purposes. The number of readers can be set according to the I/O bandwidth and the calculation speed of the hardware. The prefetch process is independent of the computing, and does not require waiting for the calculations to be completed before proceeding to the next prefetch.

In our proposed method, we design multiple data readers to provide data input for multiple workers; different data layouts may result in different read efficiency. We have provided two schemes to assign data to different workers. The specific details of these schemes are presented in Fig. 5. In the first scheme, we iteratively assign data in order to different readers. In the second one, we divide the data into
multiple contiguous parts and assign each part to a reader. We refer to the first scheme as Gap and the second one as Continuous; while both methods can achieve the same result, their implementation efficiency may be different. Accordingly, we have conducted several experiments to help us choose the better one.

We test the read performance of the two methods on these four datasets and record the four networks’ total reading time of 1000 iterations with batchsize of 1k and 2k to get the average time for each data reading. The results are shown in Fig. 6. From the results, we can clearly figure out that the Continuous method performs better than the Gap method in any situation. This is because the overhead of continuous disk access is much lower than the overhead of large-scale jumping access in TH-2 storage system. In Continuous method, readers read data as continuously as possible, while the Gap method reads data at intervals. Compared with the Continuous method, there are more large-scale jumping access, leading to worse disk access performance. Thus, we choose Continuous as the input method of our system.

4.4 Improving Efficiency

The scale of DL training’s batchsize is small in the traditional SGD-like algorithm. As the scale of deep learning increases, the data allocated to each computing node becomes very small, which will result in low computational efficiency. As increased data volume helps to increase computational efficiency, we can therefore improve the training efficiency by increasing the batchsize. However, when the batchsize is large, the convergence effect of the models will decline. To solve this problem, we introduce the LARS algorithm proposed in [15], [16].

The LARS algorithm is an accelerated convergence method that is used to solve the large batchsize problem in synchronous distributed training. There are many layers in DL models, and each layer has its own weights \( w \) and gradients \( \nabla w \). The standard SGD algorithm uses the same learning rate \( \eta \), the update rule for which is

\[
\eta = l \times \gamma \times \frac{||w||_2}{||\nabla w||_2} \quad (1)
\]

However, the ratio between \( ||w||_2 \) and \( ||\nabla w||_2 \) varies significantly for different layers, which may result in different learning rates for different layers. The learning rates of LARS are presented in Eq. (1), where \( l \) is the scaling factor and \( \gamma \) is the base input learning rate. In practice, we add momentum and weight decay to SGD. The learning rates of LARS thus change into Eq. (2), where \( \beta \) is the weight decay.

\[
\eta = l \times \gamma \times \frac{||w||_2}{||\nabla w||_2 + \beta ||w||_2} \quad (2)
\]

\[
\eta = l \times \gamma \times \frac{||w||_2}{||\nabla w + \beta w||_2} \quad (3)
\]

The LARS algorithm uses different learning rates for different layers based on the norm of weights \( ||w||_2 \) and the norm of gradients \( ||\nabla w||_2 \). The learning rates of LARS are presented in Eq. (1), where \( l \) is the scaling factor and \( \gamma \) is the base input learning rate. In practice, we add momentum and weight decay to SGD. The learning rates of LARS thus change into Eq. (2), where \( \beta \) is the weight decay. During the testing process, we also identify some problems and make some appropriate corrections to the LARS algorithm.

First, we find that some layers have zero value of \( ||w||_2 \) and gradients \( ||\nabla w||_2 \); as a result, the learning rates of these layers are infinite. In order to improve the stability of training, we set the learning rates of these layers to 1. Moreover, we find that the training effect will be improved after adjusting the learning rates of the LARS algorithm to be in line with Eq. (3). We also conduct some experiments to verify the LARS algorithm, the results of which demonstrate that the LARS algorithm enables the training of DL models with large batchsize without harming the accuracy. The results can be found in Sect. 5.4.

4.5 Developed Framework

In this subsection, we describe the distributed DL framework used in our experiments. We develop the proposed framework based on Caffe and optimize it by the methods described above. The framework can be roughly grouped
into I/O subsystem, Computing subsystem, and Communication subsystem. In I/O subsystem, we set up multiple auxiliary threads for each node to concurrently read datasets continuously, and the number of auxiliary threads can be set according to hardware performance and system requirements. In computing subsystem, we use OpenMP to start multiple computing threads on a single node, and every thread is a complete Caffe forward and backward calculation process. In addition, we also modify the Solver in Caffe to deploy the LARS algorithm in our framework. In Communication subsystem, each node selects a thread as the communication thread, and the communication threads communicate between all nodes through MPI after collecting the gradients of all computing threads in the same node.

5. Experiments

In this section, we present and discuss several experiments and their results to show the capability of our proposed method. First, the experimental configurations are introduced. Second, we test the performance of parallel data reading. Third, we verify the accuracy of our distributed method. Fourth, we evaluate the single-node performance of our proposed system in terms of different models in a computing node. Then we analyse the overall performance of our system and compare it with traditional SGD. Finally, we deploy a distributed framework using HSGD and compare it with other popular distributed frameworks.

5.1 Experimental Configuration

We use four datasets for our experiments. The first dataset is Mnist, which is trained using LeNet5 [34]. Mnist is a handwriting recognition dataset containing a training set of 60,000 examples and a test set of 10,000 examples. The second and third datasets are Cifar10 and Cifar100 [35], which are both made up of 50,000 training images and 10,000 test images; however, there are 10 classes in Cifar10 and 100 classes in Cifar100. Cifar100 is trained using AlexNet [4], and Cifar10 is trained using Resnet110 [36]. The fourth dataset is ImageNet [6], which is trained using GoogleNet [37]. The ImageNet dataset contains 1000 categories, and there are 1.2 million images in the training set and 50,000 images in the validation set.

The experimental evaluations for this paper are all performed on Tianhe-2 at the National Supercomputer Center in Guangzhou. The Tianhe-2 supercomputer consists of 8,000 computing nodes connected via TH Express-2 with Fat-tree topology. There are 64 GB memory and two Bridge-EX 2.2 GHz Xeon E5-2692v2 processors (12 cores) in a node and 192,000 computing cores in total. The operating system is Linux 2.6.32-279-TH2 #91 SMP, and the compiler is mpicxx for MPICH version 3.1.3 and icpc version 14.02 (gcc version 4.4.7 compatibility). All experiments are run three times on average.

5.2 Parallel Data Reading

In this subsection, we evaluate parallel data reading performance. It can be observed through experiments that when the amount of data and the number of readers are fixed, the number of nodes has little effect on the performance of an individual node. Accordingly, we test the performance on Mnist, Cifar10, Cifar100, and ImageNet using multiple data readers per node; the number of data readers ranges from 4 to 20. We set the batchsize to 2K, 3K, and 4K, and the time taken to perform reading 1000 times is counted. In the case of large batchsizes, if the memory is insufficient, we only run the data reading part and ignore the training part. As a result, we can test the I/O performance of ImageNet with a batchsize up to 4K; in the overall performance section, however, we can only test the batchsize up to 600. The results are shown in Fig. 7.

For Cifar10, Cifar100, and ImageNet, increasing the number of data readers will speed up the data reading process. As the number of data readers increases, data read concurrency also increases, and the data reading by each reader therefore decreases; accordingly, the efficiency of data reading will increase as the number of data readers increases.

For the Mnist dataset, when the batchsize is 3K and 4K, the time for data reading will decrease with the increase of the number of data readers; when the batchsize is 2K, however, an abnormal situation occurs. When the number of data readers is less than 12, increasing the number of data readers can improve the data reading efficiency. However, once the number of data readers exceeds 12, the data reading efficiency reduces to a small extent. When the number of data readers exceeds 16, the reading time of 2K is even bigger than that of 3K. The reason for this is as follows. We use the Continuous strategy for parallel data reading. As the number of data readers increases, the amount of large-scale jumping disk access also increases. For example, when the number of data readers is 20, the amount of large-scale jumping access is five times that occurring when there are four data readers. The large-scale access overhead is bigger than the continuous access. After increasing the number of data readers, the increase in large-scale jumping will gradually offset the benefits of increasing data readers.

![Fig. 7  Performance of parallel data reading.](image-url)
On the other hand, Mnist is a tiny dataset, and the outcome of increasing the number of data readers is relatively small compared to the impact for large datasets. As a result, after the number of data readers exceeds 12, the data reading efficiency begins to decline; moreover, when the number of data readers is 16, the reading time of 2K is even bigger than that of 3K. Above all, our parallel read strategy improves I/O read efficiency.

5.3 Communication Benefits

In our proposed HSGD architecture, we put multiple training processes in a computing node. The gradient synchronization process is divided into two parts: 1) the gradient synchronization between the training threads in one node via memory copy, 2) the gradient synchronization between different nodes through network interconnection. This two-level synchronization process can significantly reduce the number of processes involved in communication and thus reduce communication overhead. In this subsection, we will verify the communication benefits of our system. We run our verification experiments on Tianhe-2, which uses proprietary interconnect, called TH Express-2 network for high-bandwidth and low-latency interprocessor communications. TH Express-2 network leverages the features of high-radix router and network interface chips, and implements an optimized message passing interface (MPI) layer, which can support efficient execution of massively parallel message-passing programs from user space with minimal software overhead. This high-radix router chip switches data among 16 symmetric network ports, and each port has eight lanes with the bi-directional bandwidth of 160 Gbps. Some novel techniques are also used in this chip such as dynamic buffer allocation, adaptive routing based on hierarchical look-up table, intelligent network management, low-latency scrambler, improved rolling CRC, etc.

We calculated the total communication time required for 1000 iterations using the four datasets and models. The total solver number is 128, and the number of models per node is 1, 2, 4, 8, and 16, thus the required communication processes are 128, 64, 32, 16, and 8. When the number of model in a node is 1, the system is naive data parallelism. The communication time of the four models is very different. For ease of presentation, we norm the communication time to 0–100, and the result is shown in Fig. 8. We can find that the bigger the number of solvers per node, the less the communication time is required. As the total number of solvers is constant, when there are more models in a node, there are fewer processes requiring network communication. Therefore, the contention for the network bandwidth becomes smaller, and the communication performance becomes better.

5.4 Accuracy of Our Method

We trained the AlexNet model on the Cifar10 and Cifar100 datasets; the results are shown in Fig. 9. When the batchsize is 100, we use SGD with momentum, while when the batchsize is 1K and 2k, we use our method. For AlexNet on Cifar10, when the batchsizes are 100, 1k, and 2k, the corresponding initial learning rates are 0.01, 0.08, and 0.16. The learning rate policy is fixed. From the results, we find that the accuracy of our method is comparable to the accuracy of SGD with momentum. The maximum accuracy of SGD momentum is 0.784, the maximum accuracy of 1K is 0.781, and the maximum accuracy of 2K is 0.783. For AlexNet on Cifar100, when the batchsize is 100, 1K, and 2K, the corresponding initial learning rates are 0.01, 0.08, and 0.16. The learning rates policy is multistep, such that we multiply the learning rate by 0.1 at 80 and 120 epochs. The results prove that the accuracy of our method and SGD momentum is almost the same: the accuracy is 0.461, 0.457, and 0.458 respectively.

We further verify the accuracy of AlexNet, VGG16, and ResNet-50 on ImageNet, and the results are shown in Table 1. We train AlexNet and VGG16 for 100 epochs and train ResNet-50 for 90 epochs. The batchsize of SGD is

![Fig. 8](image) Comparison of the communication time for 128 workers under different configurations of HSGD.

![Fig. 9](image) Accuracy of AlexNet on Cifar10 and Cifar100.

<table>
<thead>
<tr>
<th>Models</th>
<th>SGD Accuracy</th>
<th>Our Accuracy</th>
<th>Epoch</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>0.583</td>
<td>0.581</td>
<td>100</td>
</tr>
<tr>
<td>VGG16</td>
<td>0.719</td>
<td>0.716</td>
<td>100</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>0.764</td>
<td>0.761</td>
<td>90</td>
</tr>
</tbody>
</table>
small, and the batchsize of our method is large. AlexNet’s accuracy with SGD momentum is 0.002 higher than ours, while the accuracy of VGG16 and ResNet-50 is 0.003 higher than ours. From these results, we can conclude that the difference between our method and SGD with momentum is very small. As a result, our method can use large batchsize to train DL models without negatively affecting the accuracy.

5.5 Single-Node Performance

In this subsection, we verify the single-node performance of this method using LeNet5, AlexNet, ResNet110, and GoogleNet, which are trained using Mnist, Cifar100, Cifar10, and ImageNet respectively. Since the computing node that we use has 24 cores, we set the maximum number of models to 24. We measure the performance of this system using the number of images processed per second. The results are shown in Fig. 10.

Figure 10(a) shows the results of LeNet5 trained using Mnist. Because our method can train DL models with large batchsize, we test the performance of batchsizes of 100, 1000, and 2000. When batchsize is 100, the performance first increases and then decreases as the number of threads increases. When batchsize is 1000 and 2000, the performance increases as the number of models increases; this increase in performance is similar to a linear increase. The reason for this phenomenon is as follows. Training LeNet5 using Mnist is a tiny job. When batchsize is 100, the computational task of each computing core is not fully utilized. As the number of threads increases, the computational task of each computing core is reduced, while the core computational efficiency is also reduced; this results in the performance first rising and then falling, when batchsize is 100. If batchsize is 1000 and 2000, each computing core can be provided with enough computing tasks to ensure the efficiency of these cores. Thus, their performance will improve as the number of threads increases.

Figure 10(b) and (c) present the results of AlexNet and ResNet10 trained using Cifar100 and Cifar10. Their performance behaviors are similar, as their performance continues to improve as the number of threads increases. This is because the calculations of AlexNet and ResNet110 are much larger than the calculations of LeNet5, while the small batchsize can meet the computational efficiency of the computing cores. Notably, when batchsize is 100, the performance is slightly better than when batchsize is 1000 and 2000. This is likely to be because, when batchsize is big, the cache miss rate of memory will become larger, leading to the gains from high computing efficiency are offset.

Figure 10(d) shows the results of GoogleNet trained using ImageNet. The DL model and dataset are very large. Due to the node memory limitations, we chose to test the performance of batchsizes of 100, 300, and 600. When batchsize is 300 and 600, each computing core achieves the highest computational efficiency; thus, their speeds are similar and grow linearly with an increasing number of models. When batchsize is 100, the performance growth is no longer linear; this is because the data batchsize is divided equally among threads. As the number of models increases, the amount of data allocated to each thread decreases, and the computational efficiency of the computing cores decreases accordingly; thus, the rate of increase in performance decreases. In addition, the performance of batchsize 100 is a little lower than other cases, which is mainly because the computing efficiency is low when batchsize is 100, and the gain from high computational efficiency cannot be totally offset by high cache misses. Above all, this method can greatly promote the training speed of DL models within a single node.

5.6 Overall Performance

In this subsection, we verify the overall performance of this method using LeNet5, AlexNet, ResNet110, and GoogleNet. They are trained using Mnist, Cifar100, Cifar10, and ImageNet, and their batchsize is 1000, 1000, 1000, and 600 respectively. In order to verify the performance advantages of HSGD over native SGD, we compare the performance of up to 128 models to train these DL networks. The number of models differs from the number of computing nodes. For example, when there are 64 models, there are 64 nodes in native SGD, 16 nodes in HSGD with 4 models in a node, and 8 nodes in HSGD with 8 models in a node. We measure the overall performance of this system using the number of images processed per second and the throughput per node. The results are shown in Fig. 11.

The left side of Fig. 11 shows the throughput (Image/s) of SGD and HSGD in different networks and datasets. The blue line denotes the results of native SGD, while the red line represents the results of HSGD with four models in a computing node and the green line shows the results of HSGD with eight models in a computing node. As the
number of models increases, the throughput grows larger and larger, and the growth trend is similar to that of linear growth. The reason for this behavior is as follows. The training system adopts data parallelism. As the number of models in this distributed training system increases, the training data assigned to each model becomes smaller. As a result, the degree of concurrency for training DL networks increases, with the result that the throughput becomes larger and larger. In addition, the TH-2’s communication efficiency is very high, so the growth trend is similar to linear growth.

In the left of Fig. 11, we can also see that when the number of models is the same, the throughput of native SGD is higher than HSGD (four models a node), and the throughput of HSGD (four modes a node) is higher than that of HSGD (eight models a node). For example, when the value of the x-axis is fixed, the blue line is higher than the red line, and the red line is higher than the green line. When we adopt HSGD, there are multiple models in a node. Compared to native SGD (one model a node), HSGD’s cache miss rate, memory contention and register contention are all higher than native SGD when the number of models is the same; furthermore, the more models in a node, the more unfavourable contention, which leads to a slight decline in performance.

The right of Fig. 11 shows the throughput per node, which is obtained through dividing throughput by the number of nodes. We can clearly see that HSGD can achieve higher throughput using the same number of nodes. Although HSGD will increase resource contention, it can overcome these adverse effects by increasing the hardware computing density. For example, when there are eight models in a node, the throughput per node of HSGD is significantly larger than that of native SGD. Notably, due to the effect of resource contention, the speedup is a little smaller than 8. Figure 12 shows the performance changes with the number of nodes. HSGD can achieve the same computing performance as SGD with fewer nodes. Above all, we can conclude that HSGD can accelerate the DL training process and has high computational efficiency.

5.7 Breakdown Analysis

In this subsection, we analyze the impact of various optimization strategies on the overall performance. First of all, we give the running time of I/O, communication and computing in the whole running process in Table 2. The data is obtained from GoogleNet training on ImageNet using HSGD (4 models per node), and other settings are the same with Sect. 5.6. Then we quantitatively analyze the impact of I/O and communication optimization on the overall performance, and the results are shown in Fig. 13.

In Table 2, we count the total time of I/O, communication and computing during 1000 iterations of GoogleNet, and obtain the average time of the three phases for each iteration. When the number of workers increases by eight times, the computing time decreases by nearly eight times, and the I/O time decreases by approximately two times. In addition,
although the number of nodes that need to communicate has increased by eight times, our two-level communication strategy can alleviate the communication pressure well, and the communication time has only slightly increased. Compared to computing time, I/O and communication time are much smaller. Therefore, the main acceleration benefit of HSGD comes from the improvement of computing parallelism and computing density. When the number of workers becomes larger, the proportion of communication time in the total running time becomes larger and larger. This is why the communication is a key factor affecting data parallelism DL training.

Figure 13 shows the impact of communication and I/O optimization on the overall performance of HSGD. The red line (HSGD - I/O) represents the performance of HSGD with communication optimization and no I/O optimization. The green line (HSGD - Comm) represents the performance of HSGD with I/O optimization and no communication optimization. The purple line represents neither of the two optimization methods. From the figure, the performance of red line is better than green line. We can conclude that communication optimization has a greater impact on overall performance than I/O optimization, which points out the optimization direction for our future research.

5.8 Comparison with Distributed DL Frameworks

Nowadays, there are many DL frameworks that support distributed training DL networks, such as Pytorch, Tensorflow, etc., all of which use native SGD. We develop a DL framework based on HSGD, and deploy it on Tianhe-2. In this subsection, we compare our framework with these distributed DL frameworks. Moreover, Caffe is a widely used DL framework, but it cannot be used for distributed training. Accordingly, in order to include it in the comparison, we combine Caffe and MPI to support distributed training. The comparison results between our framework and these three distributed frameworks are shown in Fig. 14. Since the other frameworks do not integrate the LARS algorithm, the batchsize is not very large. As a result, we just make a comparison in a small scale cluster.

The blue line represents the results of our framework, the red line denotes the results of Caffe + MPI, the green line indicates the results of Pytorch, and the purple line represents the results of Tensorflow. For Mnist/LeNet5, Cifar100/AlexNet, and ImageNet/GoogleNet, our framework has significant performance advantages over the other three distributed frameworks, and this performance advantage becomes larger as the number of nodes increases. For Cifar10/ResNet110, our framework performs better than Caffe + MPI and Tensorflow too. However, when the number of nodes is smaller than four, Pytorch achieves better performance than our framework, although the performance difference between our framework and Pytorch is very small in this situation. This is because our framework is developed based on Caffe, and the linear algebra libraries and GEMM algorithms used are different, which results in a small performance difference on different datasets and different networks. Once the number of nodes is greater than four, the performance of our framework grows significantly higher than Pytorch’s. In general, our framework performs much better than these three distributed frameworks based on native SGD.

6. Conclusion

We propose a novel method to efficiently train DL models on supercomputers. First, a hierarchical synchronous Stochastic Gradient Descent strategy is proposed to improve the computing efficiency. We then present a two-level parameter synchronization scheme to speed up the communication across different nodes. We further optimize the data I/O to improve data reading efficiency. In addition, the LARS algorithm is merged into our approach to increase the amount of data processed per iteration in order to improve the computational efficiency of the hardware. Furthermore, we perform ample experiments to verify the performance of our method. The proposed method can greatly improve the performance of training DL models distributed. In the future,
we intend to optimize the parameter communication process so as to reduce parameter communication without loss of accuracy.

References

ages,” University of Toronto, May 2012.


Chenxu Wang received his B.S. degree in Information Institute from Northeastern University, Shenyang, China, in 2013. And he received his M.S. degree in School of Computer Science from National University of Defense Technology, Changsha, China, in 2015. He is currently a Ph.D. student in School of Computer Science at National University of Defense Technology, Changsha, China. His research interests include machine learning, deep learning, high-performance computing and parallel algorithm.

Yutong Lu received her Ph.D. degree in School of Computer Science from National University of Defense Technology, Changsha China, in 2009. She is currently an professor of School of Data and Computer Science, Sun Yat-sen University. She is now the ISC Fellow and the director of National Supercomputer Center in Guangzhou. Her research interests include parallel system management, high speed communication, distributed file systems and advanced programming environments with MPI.

Zhiguang Chen received his Ph.D. degree in School of Computer Science from National University of Defense Technology, Changsha China, in 2013. He is currently a researcher of School of Data and Computer Science, Sun Yat-sen University. His research interests include high performance computing and large-scale storage system.

Junnan Li received his B.S. and M.S. degrees both in Computer Science and Technology from National University of Defense Technology (NUDT), Changsha, China, in 2013 and 2016 separately. He is currently pursuing the Ph.D. degree in NUDT from 2016. His research interests include packet processing on FPGA, network function acceleration, software-defined networking.