



LBB: load-balanced batching for efficient distributed learning on heterogeneous GPU cluster

Feixiang Yao¹ · Zhonghao Zhang¹ · Zeyu Ji¹ · Bin Liu^{1,2,3} · Haoyuan Gao¹

Accepted: 26 December 2023 / Published online: 9 February 2024

© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2024

Abstract

As the cost of deep learning training increases, using heterogeneous GPU clusters is a reasonable way to scale cluster resources to support distributed deep learning (DDL) tasks. However, the commonly used synchronous stochastic gradient descent (SSGD) algorithm based on the bulk synchronous parallel (BSP) model suffers from stragglers in heterogeneous clusters, resulting in a significant reduction in training efficiency. To overcome this challenge, we propose load-balanced batching (LBB) to eliminate stragglers in DDL workloads. LBB first formulates the load balancing problem and builds performance models for all workers in DDL workloads, which is achieved by analyzing the relationship between DDL iteration time and each worker's local batch size. Then the LBB balances all workers' workloads by coordinating local batch sizes. In particular, the LBB greatly mitigates static stragglers and severe dynamic stragglers by solving the load balancing problem and eliminates stragglers by batch size fine-tuning during training. LBB is implemented in PyTorch, and extensive experiments are performed on a heterogeneous server equipped with four GPUs with three different models. The experimental results verify the effectiveness of LBB on standard benchmarks, demonstrating that LBB can significantly reduce training time by 64.57%, 59%, and 5.4% compared to SSGD, local SGD, and FlexRR, respectively, without sacrificing accuracy.

Keywords Distributed deep learning · Synchronous SGD · Heterogeneous GPU cluster · Straggler · Batch size

1 Introduction

Recently, deep neural networks (DNNs) have made tremendous progress in many application domains [1, 2], but achieving optimal performance requires training increasingly complex and large models on massive datasets, which takes an enormous amount of computation time [3–5]. As a result, distributed training

has become more mainstream, with examples ranging from AlexNet [6] to Megatron-LM [7] and GPT3 [8]. These models require increasingly large clusters with effective distributed training methods to reduce the training time.

SGD (stochastic gradient descent) is a widely employed gradient descent algorithm that leverages a single-sample stochastic gradient instead of the full gradient in each iteration, thereby reducing computational overhead. The prevailing distributed training approach is data-parallel SSGD (synchronous stochastic gradient descent), an extension of SGD. In this method, each local working node on a device computes gradients using its own small batch of data, subsequently adding these gradients to the global model. The results are aggregated in a synchronized manner. Numerous studies and experiments have shown that SSGD with a fixed and average local batch size performs well on homogeneous clusters in terms of both time and performance [9].

However, due to the rapid upgrade in hardware, clusters often consist of various types of accelerator. To fully exploit the available computing resources, it is common to use heterogeneous clusters for training DDL tasks [10–12]. The performance of each worker may be degraded due to performance interference from co-located applications, or the cluster may contain servers with very different resource configurations. SSGD ignores these differences, treating all workers with the same batch size and requiring them to wait for all workers to complete gradient aggregation before the next iteration. However, this approach may give rise to the *straggler problem*, where some workers take longer to compute local gradients than others, causing the remaining workers to wait until the slowest worker has completed its computation and communication. In distributed deep learning (DDL), the straggler problem caused by resource heterogeneity is becoming an increasingly important issue that researchers are trying to solve [12–14]. Figure 2 shows a distributed training system with three workers: worker₁, worker₂, and worker₃, ranked from strongest to weakest. If the workers are given equal amounts of training data (i.e., the same local mini-batch size), worker₁ will complete its computation first and become idle, waiting for the other workers. This will cause worker₂ to become idle as well, waiting for worker₃ to complete its computation before synchronizing the gradient. This straggler effect can diminish cluster utilization and cause substantial idle time, resulting in inefficient use and wastage of computational resources.

Combining previous studies with our experiments, stragglers can be divided into two categories: static and dynamic. The former category refers to workers whose performance consistently lags behind their peers due to differences in hardware capabilities. In contrast, the latter category of stragglers emerges over time and is the result of resource contention among multiple tasks sharing a cluster. Although previous studies suggest that dynamic stragglers are the primary problem [14], recent experiments [12, 15] confirm that the performance degradation caused by static stragglers is more pronounced in heterogeneous environments. Several algorithms have been proposed over

the years to solve the straggler problem. Relaxing synchronization conditions is a common approach, but it can slow down convergence or even prevent it from converging [14, 16]. Although many new DDL-oriented load balancing techniques have been proposed [17–19], they often face significant communication overhead or are not timely enough. These studies have improved the system's ability to tolerate stragglers, but they are typically limited to homogeneous environments and are rarely conducted in heterogeneous environments.

The aim of this research is to examine the effect of stragglers on the training efficiency of DDL and propose solutions to improve it. First, it is feasible to balance the load of DDL by assigning different local mini-batch sizes to different workers based on earlier studies [20] and gradient accumulation [21]. Then, we propose a novel approach called LBB to solve the straggler problem more rapidly and precisely based on the above deduction. LBB is a load balancing method specifically designed for DDL, particularly suitable for batch processing tasks like SGD. It optimizes the batch size for each worker before training to balance computation time and significantly mitigate static stragglers. Unlike other existing methods that balance in every epoch [22], LBB dynamically rebalances the load of all workers more rapidly among iterations. To verify the training efficiency of LBB, extensive experiments are conducted on a heterogeneous server equipped with four GPUs with three different models. And four highly representative DNN models are trained on CIFAR10 and CIFAR100 datasets [23]. Experimental results demonstrate that LBB can adapt to most DNN models trained on different datasets. Furthermore, the results showed that LBB maintains the high statistical efficiency of synchronous training while achieving high throughput similar to asynchronous training.

The main contributions of this paper are as follows:

- A novel LBB method is proposed to eliminate both static stragglers and dynamic stragglers by assigning local batch sizes based on their performance, which can improve the utilization of heterogeneous clusters.
- The load balancing problem in distributed deep learning is mathematically formulated by analyzing the iteration process and modeling the performance of GPUs. This provides the foundation for LBB coordination algorithm to balance the load by coordinating local batch sizes.
- The LBB coordination algorithm is presented in this paper to eliminate stragglers. It first mitigates static stragglers by assigning appropriate batch sizes to workers before training. And then, it can rapidly predict optimal batch size for dynamic stragglers in model training, for which the load balancing is well solved.

The rest of this paper is organized as follows. Section 2 provides background information and a review of the relevant literature on the problem of stragglers. Section 3 presents the details of LBB, including how the LBB works and how it addresses the straggler problem. Section 4 describes the experimental design and analysis of the results obtained from the experiments. Finally, Sect. 5 of the paper presents the concluding remarks and future directions for this research.

2 Background and related work

2.1 Background

2.1.1 Distributed training

When training a DNN model in a distributed environment, the process typically involves three steps that are iterated repeatedly. These steps are: firstly, computing the loss through a forward pass; secondly, computing the gradients through a backward pass; and thirdly, updating the parameters through an optimizer step. The concept of data parallelism is widely applicable in this framework. Essentially, multiple copies of a model can be created, with each worker assigned a portion of the training data to perform the forward and backward passes independently. The model replicas can then synchronize either their gradients or updated parameters, depending on the algorithm being used. Among existing approaches, distributed data parallel based on SSGD is a dominant strategy due to its minimally intrusive nature [24, 25]. This has two implications: First, it ensures mathematical equivalence between the results obtained from distributed training and those obtained from local execution; second, it does not require any changes to the architecture of the DNN models or their internal operations, and changes to the optimizer may be unnecessary or equivalently implemented. The above features make SSGD-based distributed data parallelism the dominant training method today, and it performs well on homogeneous dedicated clusters, such as Facebook's well-known work training ImageNet in an hour [20].

2.1.2 Heterogeneity in data centers and clouds

Resource heterogeneity is a common feature in modern data centers and cloud environments where applications are often deployed on clusters of servers (VMs) with varying capacities and sizes. To reduce the high computational cost of distributed model training, utilizing low-cost transient VMs such as low-priority transient VMs as much as possible is a crucial technique for reducing training costs [26, 27]. This approach may involve using a heterogeneous cluster consisting of multiple transient VMs with different GPUs in a DDL task [26]. However, the use of transient VMs can also be subject to performance interference from unrelated tasks issued by the cloud provider, or even be shut down, making the assumption that all workers have equal and constant performance in a DDL task invalid. In summary, DDL tasks that can tolerate the heterogeneity of resources and the performance fluctuations of the cluster can benefit greatly from running on the modern data center.

2.1.3 Training challenges in heterogeneous environments

Performing DDL tasks in heterogeneous environments can result in a problem known as stragglers. There are two types of stragglers: static stragglers caused by

using different GPU models and dynamic stragglers caused by sharing resources. SSGD-based approaches are often ineffective due to these stragglers. While static stragglers are more problematic, dynamic stragglers cannot be ignored.

To solve the straggler problem in distributed deep learning, researchers have explored various algorithms beyond SSGD. However, these methods often improve the cluster's throughput rate but sacrifice statistical efficiency. The throughput rate measures the computational speed of a cluster, typically quantified by the average number of samples trained per second or the average elapsed time per iteration. Statistical efficiency evaluates the effectiveness of the distributed optimization algorithm by examining the convergence speed, which is determined by the relationship between training time, validation accuracy, and training loss. These research results suggest that static stragglers are more detrimental to statistical efficiency and can even nullify the benefits of some algorithms entirely [14]. Up to this point, the majority of techniques employed for DDL on heterogeneous clusters have encountered significant difficulties in achieving high levels of both statistical efficiency and throughput rate.

2.2 Related work

2.2.1 Bypassing stragglers with relaxed synchronization

The delays caused by stragglers are mainly due to synchronization constraints. Therefore, a straightforward idea for addressing this issue is to relax the synchronization constraints. For instance, ASGD [16] completely relaxes the synchronization constraints for all worker threads and proves that the strategy can be used for non-convex optimization problems. However, asynchronous training leads to stale gradients, which is caused by the gradients that are calculated based on an older version of the local model. Stale gradient usually can negatively impact the convergence speed and final convergence accuracy of the DNN model. To overcome this challenge, there are some major approaches as follows:

The first approach aims to compensate for the stale gradients by using various techniques. For example, Zheng et al. proposed a method based on Taylor expansion and Hessian approximation to compensate for stale gradients [28]. SHAT [29] improved the model update algorithm by considering the difference between the workers' model and the global model to improve the convergence of asynchronous training.

The second approach focuses on finding a trade-off between the level of synchrony and the training speed by using limited synchrony instead of strict synchrony as in the case of SSGD. A notable example is the SSP [14] algorithm proposed by Ho et al., which updates the model with a staleness threshold, reducing staleness and improving the final accuracy of the model. Another example is the N-Soft algorithm [30], which updates the model based on the average of the gradients of $\frac{1}{n}$ ready workers, while n is the cluster size. There have been several recently proposed methods that combine the ASGD and SSGD, such as [31] and [32].

Based on the experimental results of these approaches, although these approaches show an improvement over ASGD, they still result in a loss of accuracy compared to SSGD. Therefore, to the best of our knowledge, all current approaches that aim to reduce training time using ASGD inevitably lead to some reduction in the accuracy of the model.

2.2.2 Eliminating stragglers by load balancing

Load balancing is a crucial aspect of distributed computing and can be accomplished through either static or dynamic load balancing techniques. However, traditional load balancing assumes that workloads can be partitioned and transferred sequentially, which is not suitable for DDL workloads that process data in batches. To address this issue, several researchers have developed static and dynamic load balancing techniques specifically for DDL.

Static load balancing assigns the workload to each task based on its performance before the task begins, as Moreno-Álvarez et al. [18] have done. However, this method is not adaptable to changes in worker performance and may not be effective for modern deep learning workloads that process data in batches using parallel processing accelerators.

To overcome the limitations of static load balancing to accommodate performance fluctuations in the training process, many dynamic load balancing methods have been proposed. Dynamic load balancing is a more flexible technique that redistributes or coordinates the workload at runtime.

FlexRR [17], proposed by Harlap et al., monitors the real-time progress of workers and shifts the workload from slower workers to faster workers as needed. While this method is generally more effective than static load balancing, it also has high computational and communication costs for progress monitoring, status collection, and workload migration, which may not be appropriate for resource-intensive DDL tasks.

Similar to N-Soft [30], Eagar-SGD [33] uses decentralized partial synchronization for improving the training efficiency of heterogeneous clusters. DLB [22], LB-BSP [34], and BOA [19] integrate static and dynamic load balancing based on SSGD to improve training efficiency but still face challenges in handling performance fluctuations quickly. But these methods cannot rebalance rapidly when meeting performance fluctuations.

While these approaches above achieve effective load balancing on heterogeneous clusters, they either introduce asynchronicity that impairs model convergence or require several epochs to rebalance the load. Thus, there is still room for improvement in terms of efficiency.

3 Load-balanced batching method

This section discusses the LBB method, which is designed to alleviate the straggler problem in DDL (distributed deep learning) tasks. The section begins with an analysis of the time model of each training iteration to formulate the load balancing

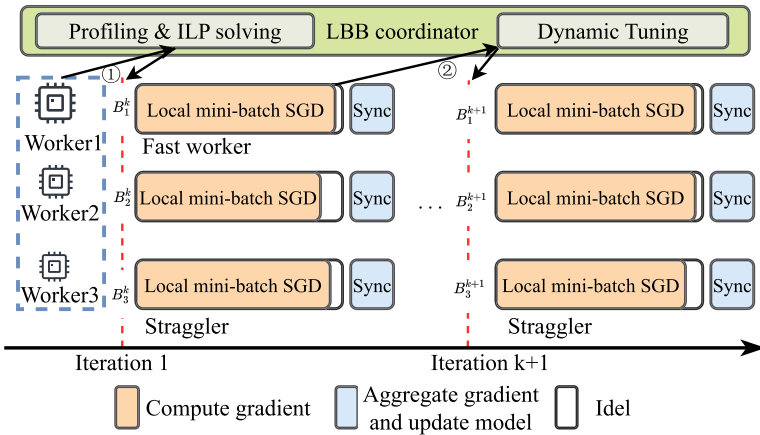


Fig. 1 Training DNN model with fixed batch size on heterogeneous devices

problem. Then, the LBB coordination method is explained in detail along with its implementation details. The overall architectural diagram is depicted in Fig. 1. LBB first collects performance information by running several batches on each GPU, and then LBB formulates load balancing strategies for various GPUs based on the analysis results. Subsequently, the CPU monitors the GPU during task execution to detect any dynamic dropout issues and takes appropriate proactive measures. The LBB method aims to improve the efficiency of DDL training by minimizing the impact of stragglers on the overall training time. Table 1 shows the symbols commonly used in this paper and their corresponding interpretations.

3.1 Iteration time modeling and problem formulation

To solve the straggler problem in synchronous SGD, it is important to ensure that each worker’s execution time is as uniform as possible during each iteration. To study and analyze this process, a model is required. The duration of an iteration T_{itr} consists of two parts: computation time T_{comp} and communication time T_{comm} . The computation time T_{comp} is determined by the local batch size, while the communication time T_{comm} depends on the straggler’s delay and the communication settings of the cluster.

3.1.1 Computation time modeling

The computation time for training a deep learning model can be analyzed as the sum of the operation time for each hidden layer in the network. For each iteration in the training process, computations are performed on each layer, including feed

forward and back propagation operations. For example, in a convolution neural network (CNN), there are two main types of hidden layers: convolution layers and fully connected layers. Given a mini-batch size B , the feed forward operation time for a convolution layer is denoted by $T_{\text{conv}}(B)$, and the forward computation time for a fully connected layer is denoted by $T_{\text{fc}}(B)$, respectively. In cuDNN, the underlying implementation library of mainstream DNN operators' forward and backward passes is implemented as matrix multiplication [35], and the computation time can be expressed as a linear function of the input batch size. In fact, modern DNN models consist of different hidden layers, and their utilization of GPU resources varies. High-end GPUs exhibit a significant unsaturation effect for light computational tasks, resulting in inefficient GPU utilization. This unsaturation effect is represented by the insignificant increase in computation time as the task size (batch size) increases, instead of the expected linear growth. There are a large number of experiments indicating that the overall relationship between batch size and computation time shows a linear correlation. However, when the batch size is small, a nonlinear correlation is observed, meaning that computation time does not increase proportionally with the batch size. Therefore, a cubic polynomial can be used to fit the relationship between batch size and computation time for all GPUs.

LBB uses a cubic polynomial to more accurately model GPU performance during profiling phase, while a linear function is used for online coordination during the training loop. The variables a_0, a_1, a_2, a_3 are employed to fit the direct relationship between T_{comp} and B . The performance of worker $_i$ can then be expressed as Equation (1).

$$T_{\text{comp}_i}(B) = \begin{cases} a_3^i B^3 + a_2^i B^2 + a_1^i B^1 + a_0 \\ a_1^i B + a_0 \end{cases} \quad (1)$$

3.1.2 Communication time modeling

LBB employs an all-reduce communication approach, which typically involves multiple nodes participating simultaneously rather than individually, to minimize the total communication time in the cluster. The all-reduce operation is typically performed by a group of multiple workers and requires all workers in the group to be ready before data transfer can begin. In heterogeneous environments, when faster workers complete their computations, they must wait for stragglers to finish their computations and initiate all-reduce communication before the actual data transfer can begin. Therefore, the actual data transfer time is defined as the time from when the last worker completes its computation to the end of communication.

The process is shown in Fig. 2. As a result, communication time in a single iteration consists of two components: waiting time and data transfer time. The waiting

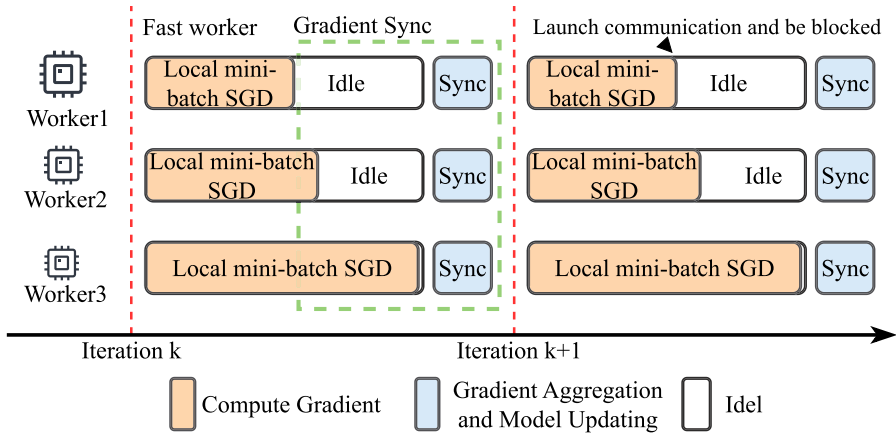


Fig. 2 Training DNN model with fixed batch size on heterogeneous devices

Table 1 Main symbols and their respective explanations

Symbols	Explanations
n	Cluster size
D	Dataset
$worker_i$	GPU_i
B	Global batch size
B_i	The local batch size of worker $_i$
B_{lim_i}	The maximum local batch size on worker $_i$
T_{comp_i}	The computation time on worker $_i$
T_{comm_i}	The time required for gradient aggregation communication on worker $_i$
T_{trans}	Fixed communication time overhead
T_{iter}	Time during one iteration

time is determined by the computation time of the fastest worker and the slowest straggler. The communication time T_{comm_i} for each worker can be calculated using $T_{comm_i} = \max\{T_{comp_j}\} - T_{comp_i} + T_{trans}$, where T_{trans} is the time required for data transfer. Since all-reduce is a synchronous and blocking communication method, the value of T_{trans} will be the same for all workers and depends primarily on the size of the communication volume and communication configuration of the cluster. T_{comm} can be considered as a function related to the local batch size of all workers. Therefore, to optimize communication time, it is crucial to reduce the waiting time, which can be achieved by mitigating the straggler effect.

3.1.3 Local and global iteration time

This paper defines the local execution time of worker $_i$ as the sum of the time required for gradient calculation, denoted by T_{comp_i} , and the time required for gradient

aggregation communication, denoted by T_{comm_i} . The j th iteration time for worker $_i$ can be expressed as follows, based on the modeling described in Sects. 3.1.1 and 3.1.2:

$$T_{\text{itr}_i}^j(B) = T_{\text{comp}_i}^j + T_{\text{comm}_i}^j \tag{2}$$

For the global iteration time in the j th iteration T_{iter}^j , the calculation time is $\max\{T_{\text{comp}_i}^j\}$ and the communication time is $\min\{T_{\text{comm}_i}^j\}$. For a cluster, an iteration is time consuming:

$$T_{\text{iter}}^j = \max\{T_{\text{comp}_i}^j\} + \min\{T_{\text{comm}_i}^j\} = \max\{T_{\text{comp}_i}^j\} + \min\{\max\{T_{\text{comp}_i}^j\} - T_{\text{comp}_i}^j + T_{\text{trans}}\},$$

where T_{trans} is constant and therefore is not the optimization target of this paper, and the minimum value of $\max\{T_{\text{comp}_i}^j\} - T_{\text{comp}_i}^j$ is 0, so this equation is converted to $T_{\text{iter}}^j = \max\{T_{\text{comp}_i}^j\} + T_{\text{trans}}$.

3.1.4 Problem formulation

The batch coordinator is responsible for coordinating the local batch size of each worker under a given global batch size. As discussed in Sect. 3.1, shortening the computation time of each iteration can be achieved by minimizing $T_{\text{iter}} = \min(\max\{T_{\text{comp}_i}\} + T_{\text{trans}})$, where T_{trans} is a constant. This problem can be transformed into minimizing $\max\{T_{\text{comp}_i}\}$, which can be further transformed into minimizing $T_{\text{comp}_i} + \min(\max\{T_{\text{comp}_i}\} - \min\{T_{\text{comp}_i}\})$, where T_{comp_i} is a function of the local batch size. Therefore, the LBB’s task is to find the optimal local batch size vector that minimizes $\max\{T_{\text{comp}_i}\} - \min\{T_{\text{comp}_i}\}$, which balances the workload across workers and reduces overall training time.

$$\begin{aligned} & \min_{\vec{B}=(B_1, B_2, \dots, B_n)} \left(\max_{i \in \{1, 2, \dots, n\}} \{T_{\text{comp}_i}\} - \min_{i \in \{1, 2, \dots, n\}} \{T_{\text{comp}_i}\} \right) \\ & \text{s.t.} \quad T_{\text{comp}_i} = T_{\text{comp}_i}(B_i); \\ & \quad \sum_{i=1}^n B_i = B, B \geq 0. \end{aligned} \tag{3}$$

In Eq. (3), B_i is constrained to be greater than or equal to 0 and less than B_{lim_i} , where B_i equal to 0 means that worker $_i$ should not participate in the training task. This is typically because the worker’s computational power is not sufficient enough to compensate for the communication overhead. Additionally, B_i should be limited by the size of worker $_i$ ’s GPU memory, denoted by B_{lim_i} . Finally, the sum of all local batch sizes (B_i) should be equal to the global batch size (B), since B is an important hyperparameter defined by the training task.

3.2 LBB coordination algorithm

Algorithm 1 Profiling phase

Require: \vec{worker}_i , cluster size n ; M : DNN model ; D : Dataset; B : Global batch size B

Ensure: $T_{comp_i}(\cdot)$: performance model for each worker; \vec{B}_i : The local batch size corresponding to $worker_i$.

- 1: **function** WORKER(M, D)
- 2: Generate simulated input data d' similar to Dataset D in the worker's dedicated memory
- 3: L_i ▷ sample points list for every worker
- 4: **for all** $worker_i (i \in \mathbb{N}, 1 \leq i \leq \text{cluster size})$ **do**
- 5: **for** bs in $\{2^p | p \in \mathbb{N}, p \geq 1\}$ **do**
- 6: Generate simulated data with batch size bs .
- 7: Start=time() ▷ record compute time
- 8: $M(b')$ ▷ forward
- 9: $M'(b')$ ▷ backward
- 10: **if** Out of Memory(OOM) occur **then**
- 11: $B_{lim_i} = bs/2$
- 12: Break
- 13: **end if**
- 14: Duration=time()-Start ▷ record compute time
- 15: $L_i.append(bs, \text{duration})$
- 16: send L_i to coordinator
- 17: **end for**
- 18: $B_{lim_i} = bs$
- 19: **end for**
- 20: **return** L_i, B_{lim_i}
- 21: **end function**
- 22:
- 23: **function** LBB COORDINATOR(\vec{L}_i, \vec{B}_i)
- 24: **for all** L_i **do**
- 25: $a_3, a_2, a_1, a_0 = \text{result from fit the 3.1.1 by } L_i$
- 26: **end for**
- 27: Substitute $T_{comp_i}(\cdot)$ into Eq. (3)
- 28: $\vec{B}_i = \text{solution to the Eq. (3)}$
- 29: distribute \vec{B}_i to all worker, and prepare for training.
- 30: **end function**

Algorithm 2 LBB coordinator algorithm

Require: n : cluster size; CT_{comp_i}, CB_i : current compute time and current batch size of $worker_i$; B : global mini batch; RT, FT : rapid-tuning threshold and fine-tuning threshold for SE .

Ensure: B_i : local batch size for $worker_i$ in the next iteration.

- 1: **function** BUILD_ROUGHMODEL($\{CB_i\}, \{CT_{comp_i}\}$)
- 2: **for all** $worker_i$ **do**
- 3: $T_{comp_i}(b) = \frac{CT_{comp_i} - a_0}{CB_i} \times b + a_0$ \triangleright Characterize workers' performance with a new model $T_{comp_i}(\cdot)$, where a_0 is defined in Eq. (1)
- 4: **end for**
- 5: **end function**
- 6:
- 7: **function** DYNAMIC_TUNING($\{T_{comp_i}\}, \{CB_i\}$)
- 8: **if** $SE < FT$ **then**
- 9: Pass \triangleright well balancing now, FT default set to 0.05
- 10: **else if** $SE < RT$ **then** $\triangleright RT$ default set to 0.3
- 11: $T_{\max}, i_{t \max} = \max_{i \in \{1, 2, \dots, n\}} \{CT_{comp_i}\}$
- 12: $T_{\min}, i_{t \min} = \min_{i \in \{1, 2, \dots, n\}} \{CT_{comp_i}\}$
- 13: $B_{i_{t \max}} = CB_{i_{t \max}} - \lfloor \frac{step}{2} \rfloor$ \triangleright Adjusting local batch size
- 14: $B_{i_{t \min}} = CB_{i_{t \min}} + \lceil \frac{step}{2} \rceil$ \triangleright Adjusting local batch size
- 15: **else** $\triangleright SE > RT$
- 16: BUILD_ROUGHMODEL($\{CB_i\}, \{CT_{comp_i}\}$)
- 17: $\vec{B}_i = \arg, \min_{\vec{x}} f(\vec{x})$
- 18: **end if**
- 19: **end function**
- 20: $\vec{B}_i =$ DYNAMIC_TUNING($\{CB_{comp_i}\}, \{CT_{comp_i}\}$)
- 21: Distribute \vec{B}_i to all worker as the batch size for next iteration.

Algorithm 3 $worker_i$ training algorithm

Require: M :DNN model; D : Dataset; B_i : local batch size.

Ensure: CB_i : Current batch size; CT_i : corresponding computation time.

- 1: **while** training not finished **do**
- 2: $CB_i = B_i \leftarrow$ LBB Coordinator
- 3: get a batch data $input$ from Dataset D , and $|input| = B_i$
- 4: calculate local gradient g_i and record the duration CT_{comp_i}
- 5: Fut=ALL-Reduce($\frac{CB_i}{B} \times g_i$, op=sum, non-blocking=True) \triangleright aggregation gradient with CB_i/B as weight, Fut is an async operation handler
- 6: send (CB_i, CT_{comp_i}) to LBB coordinator
- 7: Fut.wait() \triangleright wait for gradient communication finished
- 8: **end while**

LBB is designed to reduce the computation time for each iteration by coordinating the local batch size of all workers. This is achieved by minimizing the difference between the longest and shortest computation times. LBB uses an independent

process called the LBB coordinator to accomplish this task, which ensures that the training process is not disturbed by the LBB coordinator. The LBB coordinator has three main functions: profiling, evaluation of straggler effects, and coordination of local batch sizes for all workers. These functions are explained in detail in this section. Additionally, this section also discusses how to perform gradient aggregation under the LBB-guided training method.

3.2.1 Profiling phase

The paper proposes a profiling phase to obtain suitable parameters for fitting each worker's performance model in LBB. To collect the runtime information of each worker as fitting samples, LBB introduces an optional profiling phase before training, which is described in function *worker()* of Alg. 1. The profiling phase samples data pairs of local iteration time and local mini-batch size, with sampling points set at 2^p . When the local batch size is small, this allows for dense sampling, as the relationship between GPU computation time and batch size may not be linear. Lines 10–12 describe when an OOM (Out of Memory) error is encountered during sampling. The approximate batch size upper bound of the current worker is recorded. Alg. 1 exhibits a time complexity of $O(n \log k)$ and a space complexity of $O(n(k + \log k))$, where n represents the cluster size and k denotes the maximum batch size. This suggests that the time-space cost of Alg. 1 is relatively modest. The profiling phase is a one-time execution carried out prior to the commencement of training. It typically entails the execution of hundreds of dataset samples on each compute node. In comparison with the entire training process, this step incurs a time overhead of only two to three seconds, which is considered negligible. After the profiling phase, the coordinator fits the performance models of each worker based on these sample points in line 25 and then solves Eq. (3) to get the optimal batch size for all workers in line 28. The local batch size for all workers is then set based on the results. Although the profiling phase is optional, the coordinator can also fit the performance models of each worker based on real-time performance during training.

3.2.2 Straggler effect

Before discussing the behavior of the coordinator during training, it is necessary to define an indicator to describe the straggler effect in the cluster. For an iteration, if stragglers exist, other nodes will wait for the stragglers. The longest idle time is determined by the difference between the longest gradient computation time and the shortest gradient computation time. This paper denotes the straggler effect SE as the proportion of the longest idle time to the total computation time in an iteration, i.e., the straggler effect represents the proportion of time wasted due to slow nodes in the total iteration time as shown in Eq. (4) where \bar{T}_{comp} denotes the average computation time. The coordinator collects the real-time computation time of each worker during the computation and calculates the SE. Based on the SE, the coordinator performs various coordination actions. In LBB, if SE is less than RT as defined in Alg. 2, it indicates that the stragglers are eliminated, achieving load balanced in the cluster. The assessment of straggler effects is conducted on the CPU and does not impact

GPU training. This process will take a few milliseconds, resulting in negligible time overhead.

$$SE = \frac{\max \{T_{\text{comp}_i}\} - \min \{T_{\text{comp}_i}\}}{\overline{T}_{\text{comp}}} \quad (4)$$

3.2.3 Batch coordinator

The behavior of LBB coordinator is determined by two thresholds called the fine-tuning threshold and rapid-tuning threshold. If SE is below the fine-tuning threshold, it means that the delay time caused by stragglers is minimal, and the local batch size for all workers does not need to be adjusted. If SE is smaller than the rapid-tuning threshold, it means that the straggler is only slightly behind the other nodes, so LBB coordinator will use fine-tuning. Furthermore, if SE is greater than the rapid-tuning threshold, it indicates that the straggler is significantly lagging behind the other nodes, resulting in significant performance waste, and LBB coordinator will use rapid-tuning. Algorithm 2 explains the coordination methods of the LBB coordinator. For fine-tuning, as shown in lines 10–14, the local batch size of the worker who takes the least time in the current iteration is increased. Conversely, the local batch size of the worker that takes the longest time in the current iteration is decreased. The increase or decrease step in the batch size is usually 1. For rapid-tuning, it is only adopted when SE is larger than rapid-tuning threshold. When SE is higher than the rapid-tuning threshold, it indicates that the current local batch size is not suitable for real-time GPU performance due to thermal throttling or other computation tasks interfering with the GPU's computational resources, leading to significant fluctuations in worker performance. In such cases, the coordinator replaces the current performance model with a new one based on newly collected information about computation time and local batch size as shown in function *Build_RoughModel()* in Alg. 2. The LBB coordinator uses the last collected CB_i and CT_{comp_i} to build the immediate performance model, as shown in Eq. (1). Here, a_0^i can be estimated from a_0 in the performance model of the profiling phase as a rough estimate of the startup overhead of worker $_i$, while a_1^i can be estimated by $\frac{CT_{\text{comp}_i} - a_0^i}{B_i}$ as the real-time throughput. According to the experimental validation of Fig. 3, it can be assumed that the modified performance model can also reflect the actual performance.

In Alg. 2, the LBB coordinator performs a batch update for each worker, repeated n times, and this process requires a request for constant level space, which is released when it is used up. This indicates that Alg. 2 can complete the task in a very short time and using very little space.

The LBB coordinator uses the new performance model in Eq. (3) to calculate the new local batch size for each worker, which is called rapid-tuning. Through several experiments, it has been found that the straggler effect of the cluster usually decreases to below 0.1 after one iteration of rapid-tuning, indicating that rough balancing has been achieved. After a few rounds of fine-tuning, the execution time of all workers

can be balanced quite well. This will be confirmed with experiments in Sect. 4.4. This simple model is used to achieve rapid-tuning in the batch coordination phase, because slowly adjusting the batch size to balance the execution time of all workers in the cluster through fine-tuning requires more iterations and is less timely.

Furthermore, rapid-tuning cannot guarantee perfect cluster balancing at once, resulting in SE remaining above the rapid-tuning threshold and continuous rapid-tuning without effectively reducing SE. To address this issue, LBB introduces an observation window where only fine-tuning is allowed to avoid continuous rapid-tuning. LBB also prioritizes fine-tuning for workers that are significantly slower than other nodes to achieve better performance. These are some miscellaneous details of the implementation of the LBB that are not described in Alg. 2 for conciseness and ease of reading.

LBB employs Alg. 3 to dynamically adjust the batch size throughout the training process, ensuring that dynamic stragglers do not significantly impede training efficiency. In Alg. 3, there are a total of m iterations, each worker has to execute m times, so the time complexity is $O(nm)$, also only need to apply for the constant level space and release it after use, so the space complexity is $O(1)$.

3.2.4 Gradient aggregate

During each iteration of the training process, the task of worker _{i} is as follows: Firstly, obtain the local batch size for this iteration from the LBB coordinator, denoted as B_i or CB_i . Next, retrieve a batch of data from the dataset as input, calculate the corresponding gradient g_i , and record the time used for this iteration, denoted as CT_{comp_i} . The gradient communication and the sending of (CB_i, CT_{comp_i}) to LBB coordinator happen at the same time, so the overhead of LBB is overlapped. LBB initiates non-blocking gradient communication via *torch.distribution.all-reduce* to achieve the above process. At the end of each training round, the gradients computed by all workers are aggregated. This is achieved by taking the weighted average of each worker's gradient, where the weight is the local batch size for each worker. To synchronize the gradients, each worker computes its local weighted gradient using $G_{local} = \frac{G_i B_i}{B}$, where G_i and B_i are the gradient and batch size of worker _{i} , respectively. Then, all workers communicate with each other to aggregate the global gradient by summing all local weighted gradients. By correcting with $\frac{G_i B_i}{B}$, each sample is updated equally in terms of parameter updating, regardless of any inconsistency in batch size. Consequently, for an independent and identically distributed dataset, LBB can achieve the same level of statistical efficiency as SSGD, which is considered the optimal approach. This conclusion is verified in Sect. 4.3.

4 Experiment

This section presents experimental results that demonstrate the effectiveness of LBB. Firstly, we verify the accuracy of the performance model in fitting the actual GPU performance. Next, we investigate and compare the training speed of LBB in

the presence of static stragglers with various synchronous training methods, including synchronous SGD (SSGD), One-Shot Averaging (OSA), and local SGD [36] with different local step(H) configurations. In addition, asynchronous methods such as N-Soft Sync are compared with LBB to provide a comprehensive understanding of its throughput rate. Furthermore, we evaluate the impact of dynamic batch tuning and demonstrate how it can handle cluster performance fluctuations. Overall, the experimental results demonstrate that LBB outperforms the other synchronous and asynchronous methods in terms of both training speed and throughput rate, and that dynamic batch tuning can further improve its performance in handling varying cluster conditions.

4.1 Experiment setup

Several classic and representative deep learning models, such as ResNet-18 [37], ViT-B/16 [38], ShuffleNet v2 [39], and EfficientNet-B1 [40], are trained on CIFAR10 and CIFAR100 datasets with a global batch size of 512 and a learning rate of 0.1 (except for the ViT-B/16 model, which had a learning rate of 0.0001). The training is conducted for 120 epochs with cosine annealing learning rate decay. To simplify the experiment, SGD optimization method without momentum is used in experiments.

All experiments are conducted using a custom machine with four GPUs, consisting of two GTX 1070s (worker_1 and worker_2), one Tesla M40 (worker_3), and one GTX 750 (worker_4) added as a straggler to the cluster. The GPUs are of three different models from two architectures as shown in Table 2. One GTX 1070 is directly connected to the CPU via PCIe 3.0 x8, while the other GPUs are connected via PCIe 3.0 x4 to bypass the limitations of the B660 chipset on the motherboard while each lane of PCIe 3.0 can provide up to 985 MB/s connection speed.

The benchmark datasets chosen for this research are CIFAR10 and CIFAR100. The CIFAR10 dataset comprises 60,000 32×32 color images divided into 10 classes, each consisting of 6000 images. It is organized into 50,000 training images and 10,000 test images, separated into five training batches and one test batch, with 10,000 images each. Similarly, the CIFAR100 dataset comprises 100 classes of 600 images each, with 500 training images and 100 test images per class. Although both datasets contain diverse images, CIFAR100 has 100 classes, while CIFAR10 has only 10. These datasets are commonly used in the deep learning community to evaluate image recognition models.

Table 2 Experiment environment

	Model	Architecture	CUDA cores	Single precision performance	GPU memory
worker_1	Tesla M40	Maxwell	3072	6.844 Tflops	24GB GDDR5
worker_2	GTX 1070	Pascal	1920	6.463 Tflops	8GB GDDR5
worker_3	GTX 1070	Pascal	1920	6.463 Tflops	8GB GDDR5
worker_4	GTX 750	Maxwell	512	1.111 Tflops	2GB GDDR5

Table 3 Comparison of computation time with and without load balancing

	SSGD		Balanced	
	Local batch size B_i	T_{comp_i} (ms)	Local batch size B_i	T_{comp_i} (ms)
worker ₁	128	81.81	167	104.94
worker ₂	128	81.49	167	104.14
worker ₃	128	85.39	158	103.46
worker ₄	128	392.92	20	104.41
Cluster	512	392.92	512	104.94

Table 4 Accuracy of the performance models for different workers in LBB

	Profiling-based solution			Rapid-tuning-based solution		
	B_i	\widehat{T}_{comp_i} (ms)	Error (%)	B_i	\widehat{T}_{comp_i} (ms)	Error (%)
worker ₁	166	103.68	1.20	165	108.97	3.84
worker ₂	166	103.09	- 1.01	165	109.66	5.30
worker ₃	160	104.02	- 0.52	160	110.24	0.41
worker ₄	20	37.60	- 6.52	22	92.06	- 11.83
Cluster	512	104.02	- 0.87	512	110.24	0.50

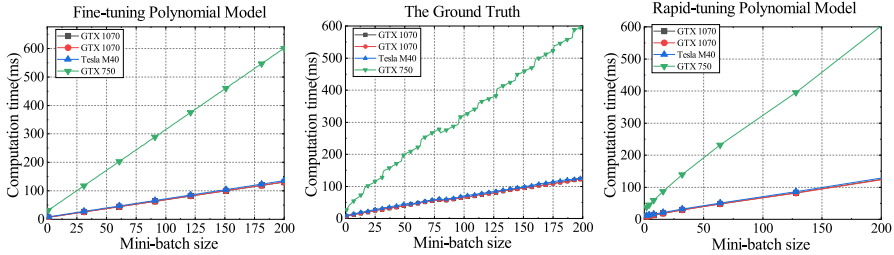


Fig. 3 Comparing GPU performance for training ResNet-18 on CIFAR10: GTX 1070s and Tesla M40 show similar results, while GTX 750 lags behind

4.2 Verifying performance model and mitigation of static stragglers

To verify the accuracy of LBB’s profiling and constructed performance models, a validation is performed in this section. By collecting performance sample points from each worker during the profiling phase, LBB is able to construct performance models for each worker, as shown in Fig. 3. The resulting models showed that workers of the same type had similar performance trends, while workers of different types had significantly different performance trends, providing valuable insights into the coordination process. Pearson correlation and Spearman correlation can be calculated from the experimental results in Fig. 3. The Pearson correlation assesses the strength of the linear relationship between two variables, while

the Spearman correlation is a nonparametric statistic that gauges the monotonic relationship between two variables. The relationship between each worker's computation time and its local batch size is well fitted with the Pearson correlation coefficient and the Spearman correlation coefficient between the fitted and actual values of all workers exceeding 0.99. This theoretically proves the validity of our method. These results demonstrate a strong correlation between the local batch size and computation time, affirming the accuracy of our performance model in reflecting the actual performance of all workers in the cluster.

Table 3 shows the unbalanced computation time caused by the load imbalance of SSGD, and the overall computation time of the cluster is dragged down by the straggler. Table 4 shows the mini-batch sizes coordinated by LBB based on the performance model, with the global mini-batch size set to 512. The table shows that the performance model effectively captures the relationship between computation time and local batch size for each worker. But there is an exception, during the rapid-tuning phase, when LBB uses a simple linear regression model, the worker₄ (GTX 750) exhibits poor prediction accuracy, which is consistent with the results shown in Fig. 3. This is due to the fact that the GTX 750's graphics memory is not sufficient for the corresponding training task, which causes PyTorch to try to use a different low-level implementation for computing the operators in the DNN model, resulting in a more nonlinear relationship between its batch size and actual computation time. This is caused by enabling the *torch.backends.cuDNN.benchmark* parameter, which is a common method to accelerate DNN model training.

LBB coordinator uses the performance model to determine the appropriate batch size for each worker and then compares the predicted computation time to the actual computation time. For most workers, the deviation rate is low, indicating that the performance model is accurate and the batch coordination strategy is appropriate. Therefore, the static straggler is greatly mitigated or even eliminated. As shown in Fig. 4, by assigning appropriate local batch sizes to different workers, the time to train ResNet-18 on CIFAR10 for one epoch is reduced from an average of 62.44 s to 22.12 s.

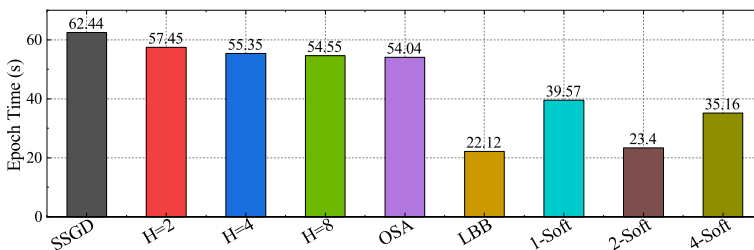


Fig. 4 Comparison of ResNet-18 training times per epoch with different methods: $H = 2$ means that model does a synchronization every two iterations, 1-Soft means 1-Soft Sync, and OSA means One-Shot Averaging

4.3 Training efficiency

In this section, the training efficiency of LBB will be reviewed before introducing dynamic stragglers. The efficiency of LBB will be described in terms of statistical efficiency and training throughput. First, the statistical efficiency of LBB will be verified by comparing it with SSGD to see whether it affects the statistical efficiency of synchronous training. Then, the statistical efficiency of LBB will be compared with other methods used to mitigate stragglers. Finally, the throughput and end-to-end training time of LBB will be compared with other algorithms.

Statistical efficiency: Before discussing throughput and dynamic coordination, it is necessary to verify the statistical efficiency of LBB. In this experiment, the statistical efficiency of LBB is verified in two steps. First, four representative models are trained using LBB and SSGD on the CIFAR10 and CIFAR100 datasets. This step is performed to verify that LBB, as a method based on SSGD and adapted to heterogeneous clusters, does not compromise the excellent convergence of SSGD. As shown in Fig. 5, LBB showed a statistical efficiency very similar to SSGD in all eight training tasks. Next, it is necessary to compare the statistical efficiency of LBB with that of other parallel algorithms. As shown in Fig. 4, methods based on synchronization, such as SSGD, local SGD, and LBB, exhibit highly similar convergence and are significantly better than the asynchronous N-Soft Sync method. In conclusion, we have verified that LBB maintains the excellent statistical efficiency of SSGD and is not inferior to other parallel algorithms.

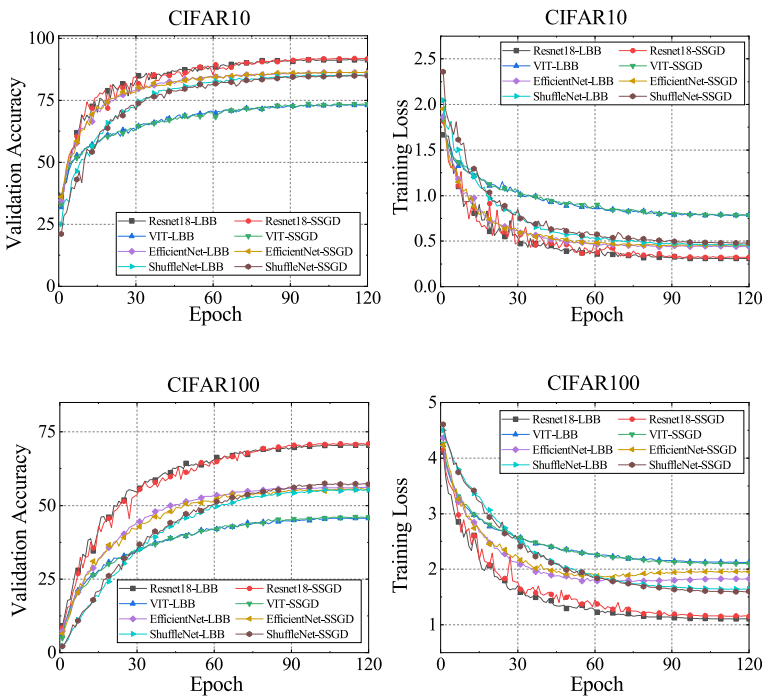


Fig. 5 Training different DNN models on CIFAR10 and CIFAR100 with SSGD and LBB

In this experiment, the throughputs of different training strategies are reflected in the time used for each epoch, with shorter times indicating higher throughput. For synchronous training methods, SSGD, local SGD, and OSA can be seen as similar methods with different local step sizes. As the local step size increases, the number of communications and the communication cost decrease, and this is reflected in the overall reduction in time consumption, as shown in the left half of Fig. 4. However, even though LBB has no reduction in the number of communications, it still has a high throughput rate due to the good load balancing. Therefore, compared to SSGD, local SGD-based methods (local SGD and OSA), and 2-Soft, LBB can reduce the per-epoch time by 64.57%, 59%, and 5.4%, respectively.

Overall, LBB's statistical efficiency is comparable to SSGD and better than asynchronous parallel methods, while maintaining a comparable throughput rate to asynchronous parallel methods. These factors ultimately lead to a significant increase in model training speed, achieving higher model accuracy in less time without compromising model convergence. As shown in Fig. 6, LBB and 2-Soft Sync are the fastest to complete 120 epochs, taking less than 3000 s. However, using LBB resulted in significantly higher convergence accuracy and

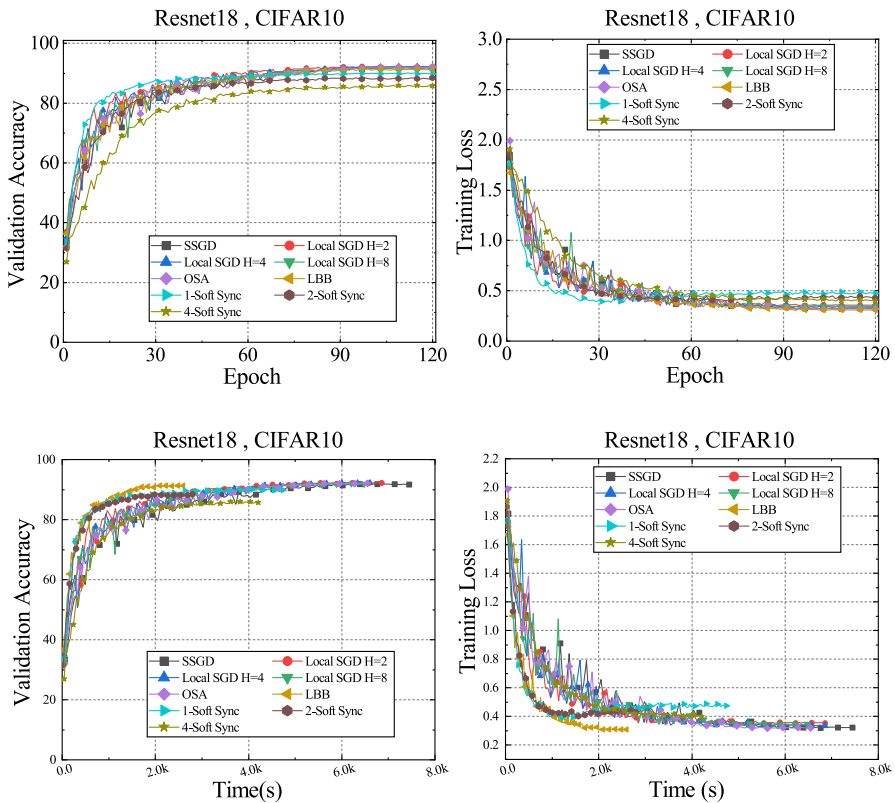


Fig. 6 Training ResNet-18 on CIFAR10 with different algorithm

lower validation loss than training with 2-Soft Sync. Although other synchronous parallel strategies are able to converge the model to similar accuracy as LBB, they take significantly more time. Compared to OSA, LBB saved almost half the time, while compared to SSGD, LBB saved about 70% of the time. The performance of the synchronous training algorithms is mainly affected by the presence of severe stragglers, the GTX 750, which significantly slows down other workers. This also indicates that LBB is effective in dealing with extreme static stragglers.

4.4 Verifying mitigation of both static and dynamic stragglers

This section presents a case study that illustrates how LBB mitigates the straggler effect in distributed deep learning systems. In this study, we conducted experiments using the four-GPU machine mentioned above to train ResNet-18. We designed a program to simulate the situation where each worker is disturbed in a controlled manner. Specifically, the program cyclically increases the computation time of a particular GPU by about 50 ms. The changes in computation time are shown in Fig. 7a. This figure shows that the computation time of the disturbed GPUs increased by about 50 ms (as indicated by the upward vertices), while the computation time of the GPU with the disturbance removed decreased by about the same amount (as indicated by the downward vertices). To mitigate dynamic stragglers, the LBB coordinator continuously monitors SE across the cluster. When significant perturbations are detected, the LBB coordinator quickly coordinates the batch size of each worker using a rapid-tuning mechanism. Figure 7c illustrates this mechanism, where the LBB coordinator adjusts the local batch size of each worker in real time, and the SE decreases rapidly. The rebalancing of computation times and batch sizes is shown in Fig. 7b, where the batch size of each worker is adjusted to ensure optimal performance of the distributed system. This mechanism effectively ensures that the entire cluster can train deep neural network models efficiently even in the presence of disturbances.

Specifically, as shown in Fig. 7, at the beginning of training, all workers are in a balanced computation time, and SE is below the tuning threshold most of the time, so almost no batch coordination occurs. Then, near iteration 25, the Tesla M40 in the cluster suffers from our introduced disturbance, the computation time increases, it becomes a new straggler, and SE increases dramatically, as shown in Fig. 7a, c. The LBB coordinator monitors this change, and SE exceeds the rapid-tuning threshold, so the rapid-tuning mechanism is triggered. Subsequently, SE still exceeds the fine-tuning threshold, so LBB continues to fine-tune the batch size until SE falls below the fine-tuning threshold. At iteration 140, SE rises dramatically again and exceeds the rapid-tuning threshold due to a newly introduced disturbance for GTX 750. However, rapid-tuning is not triggered immediately due to the remedy mentioned at the end of Sect. 3.2 being triggered.

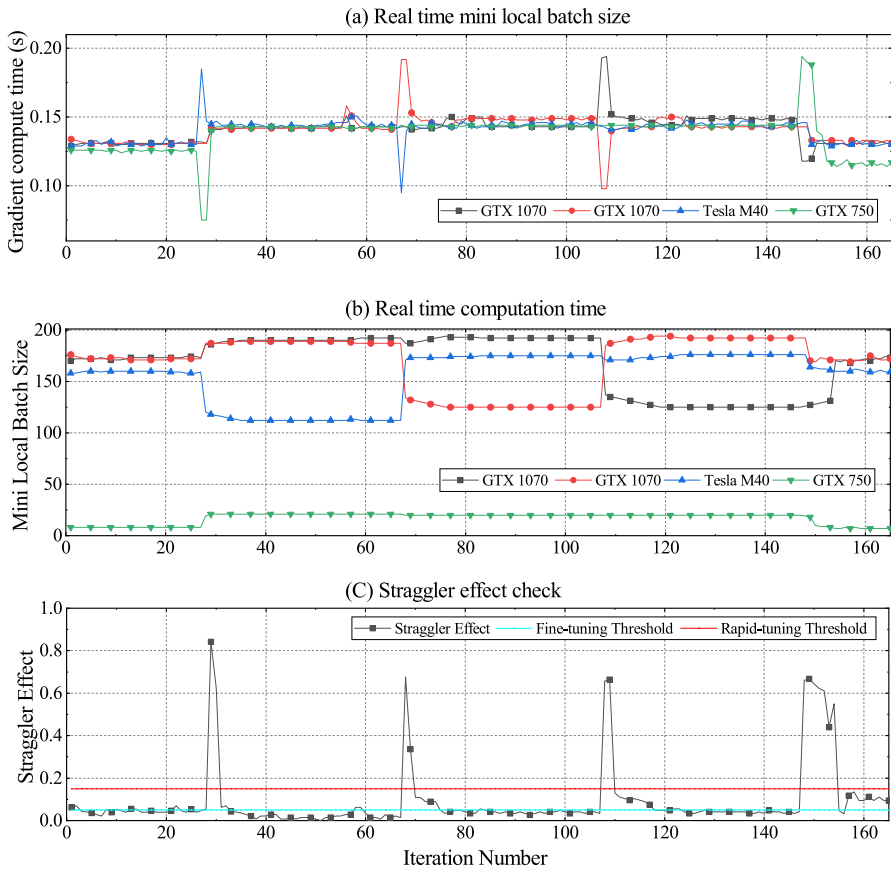


Fig. 7 Graph **a** shows the change in computation time when a disturbance is intentionally injected into each worker. Graph **c** shows the LBB coordinator monitoring the straggler effect of the cluster, while graph **b** shows how it coordinates the batch size of all workers

5 Conclusion

The straggler problem resulting from a heterogeneous GPU cluster is a bottleneck of data parallelism with synchronous strategies. To alleviate this problem, this paper proposes an innovative load balancing method LBB designed for data parallelism in heterogeneous environments. LBB reduces waiting costs by assigning appropriate local batch sizes to all workers before and during training, which is implemented in PyTorch by the LBB coordinator at a low cost. The performance model of GPU workers and formulation of load balancing problems built by LBB accurately reflect the heterogeneous cluster's performance. These provide a strong theoretical foundation for LBB's load balancing. Based on these, LBB can greatly mitigate static stragglers before training. It also rebalances severe dynamic stragglers rapidly, while mild dynamic stragglers can be eliminated through batch size fine-tuning. Extensive experimental results demonstrate LBB's effectiveness in load balancing and

improving utilization in heterogeneous clusters. As a result, LBB maintains high convergence speed based on synchronous training while effectively addressing straggler issues. In the future, we plan to integrate LBB with communication optimizations, which will further increase LBB's efficiency and scalability.

Acknowledgments The authors would like to acknowledge the support of National Natural Science Foundation of China under grant No. 62376226, the Shaanxi's Key Research and Development Program under grant 2023-ZDLNY-63, the Xianyang's Key Research and Development Program under grant No. L2022-ZDYF-NY-019, and the Key Research and Development Program of Shaanxi under grants No. 2019ZDLNY07-06-01 and No. 2020NY-098.

Author contributions FY proposed the idea, participated in the protocol design, authored the main sections of the paper, and produced the figures and tables. BL supervised the research and proofread the manuscript as the supervisor. HG participated in constructing the experimental workflow.

Funding This work is supported by National Natural Science Foundation of China under grant No. 62376226, the Shaanxi's Key Research and Development Program under grant 2023-ZDLNY-63, the Xianyang's Key Research and Development Program under grant No. L2022-ZDYF-NY-019, and the Key Research and Development Program of Shaanxi under grants No. 2019ZDLNY07-06-01 and No. 2020NY-098.

Data availability The public datasets of CIFAR10 and CIFAR100 [23] used in the research are available at <https://www.cs.toronto.edu/kriz/cifar.html>.

Code availability The authors will release LBB implementation for reproducibility after it is organized. The code will be released on <https://github.com/FLYING37520/LBB>.

Declarations

Conflict of interest The authors declare that they have no competing interests as defined by Springer or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethics approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser L, Polosukhin I (2017) Attention is all you need. In: *Advances in neural information processing systems*, vol 30
2. Jiang P, Ergu D, Liu F, Cai Y, Ma B (2022) A review of Yolo algorithm developments. *Proc Comput Sci* 199:1066–1073. <https://doi.org/10.1016/j.procs.2022.01.135>
3. Saharia C, Chan W, Saxena S, Li L, Whang J, Denton E, Ghasemipour SKS, Ayan BK, Mahdavi SS, Lopes RG, Salimans T, Ho J, Fleet DJ, Norouzi M (2022) Photorealistic text-to-image diffusion models with deep language understanding. [arXiv:2205.11487](https://arxiv.org/abs/2205.11487) [cs.CV]
4. Ramesh A, Pavlov M, Goh G, Gray S, Voss C, Radford A, Chen M, Sutskever I (2021) Zero-shot text-to-image generation. In: *Proceedings of the 38th International Conference on Machine Learning*, vol 139, pp 8821–8831
5. Radford A, Kim JW, Hallacy C, Ramesh A, Goh G, Agarwal S, Sastry G, Askell A, Mishkin P, Clark J, Krueger G, Sutskever I (2021) Learning transferable visual models from natural language supervision. In: *Proceedings of the 38th International Conference on Machine Learning*, vol 139, pp 8748–8763. <https://proceedings.mlr.press/v139/radford21a.html>

6. Krizhevsky A, Sutskever I, Hinton GE (2017) Imagenet classification with deep convolutional neural networks. *Commun ACM* 60(6):84–90. <https://doi.org/10.1145/3065386>
7. Shoeybi M, Patwary M, Puri R, LeGresley P, Casper J, Catanzaro B (2020) Megatron-LM: training multi-billion parameter language models using model parallelism. [arXiv:1909.08053](https://arxiv.org/abs/1909.08053) [cs.CV]
8. Brown T, Mann B, Ryder N, Subbiah M, Kaplan JD, Dhariwal P, Neelakantan A, Shyam P, Sastry G, Askell A, Agarwal S, Herbert-Voss A, Krueger G, Henighan T, Child R, Ramesh A, Ziegler D, Wu J, Winter C, Hesse C, Chen M, Sigler E, Litwin M, Gray S, Chess B, Clark J, Berner C, McCandlish S, Radford A, Sutskever I, Amodei D (2020) Language models are few-shot learners. In: *Advances in neural information processing systems*, vol 33, pp 1877–1901
9. Tang Z, Shi S, Chu X, Wang W, Li B (2020) Communication-efficient distributed deep learning: a comprehensive survey. [arXiv:2003.06307](https://arxiv.org/abs/2003.06307) [cs.CV]
10. Gan S, Jiang J, Yuan B, Zhang C, Lian X, Wang R, Chang J, Liu C, Shi H, Zhang S, Li X, Sun T, Yang S, Liu J (2021) Bagua: scaling up distributed learning with system relaxations. *Proc VLDB Endow* 15(4):804–813. <https://doi.org/10.14778/3503585.3503590>
11. Jiang J, Cui B, Zhang C, Yu L (2017) Heterogeneity-aware distributed parameter servers. *Association for Computing Machinery*, New York, pp 463–478. <https://doi.org/10.1145/3035918.3035933>
12. Narayanan D, Santhanam K, Kazhmiaka F, Phanishayee A, Zaharia M (2020) Heterogeneity-aware cluster scheduling policies for deep learning workloads. In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp 481–498. <https://www.usenix.org/conference/osdi20/presentation/narayanan-deepak>
13. Kim H, Song C, Lee H, Yu H (2023) Addressing straggler problem through dynamic partial all-reduce for distributed deep learning in heterogeneous GPU clusters. In: *IEEE International Conference on Consumer Electronics (ICCE)*, pp 1–6. <https://doi.org/10.1109/ICCE56470.2023.10043527>
14. Ho Q, Cipar J, Cui H, Lee S, Kim JK, Gibbons PB, Gibson GA, Ganger G, Xing EP (2013) More effective distributed ML via a stale synchronous parallel parameter server. In: *Advances in neural information processing systems*, vol 26
15. Kavarakuntla T, Han L, Lloyd H, Latham A, Akintoye SB (2021) Performance analysis of distributed deep learning frameworks in a multi-GPU environment. In: 20th International Conference on Ubiquitous Computing and Communications (IUCC/CIT/DSCI/SmartCNS), pp 406–413. <https://doi.org/10.1109/IUCC-CIT-DSCI-SmartCNS55181.2021.00071>
16. Keuper J, Pfreundt F-J (2015) Asynchronous parallel stochastic gradient descent: a numeric core for scalable distributed machine learning algorithms. In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments. MLHPC '15*. Association for Computing Machinery, New York. <https://doi.org/10.1145/2834892.2834893>
17. Harlap A, Cui H, Dai W, Wei J, Ganger GR, Gibbons PB, Gibson GA, Xing EP (2016) Addressing the straggler problem for iterative convergent parallel ML. In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, pp 98–111. <https://doi.org/10.1145/2987550.2987554>
18. Moreno-Alvarez S, Haut JM, Paoletti ME, Rico-Gallego JA, Diaz-Martin JC, Plaza J (2020) Training deep neural networks: a static load balancing approach. *J Supercomput* 76:9739–9754
19. Yang E, Kang D-K, Youn C-H (2020) BOA: batch orchestration algorithm for straggler mitigation of distributed DL training in heterogeneous GPU cluster. *J Supercomput* 76:47–67
20. Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K (2018) Accurate, large minibatch SGD: training imagenet in 1 hour. [arXiv:1706.02677](https://arxiv.org/abs/1706.02677) [cs.CV]
21. Tao Z, Li Q (2018) eSGD: communication efficient distributed deep learning on the edge. In: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. USENIX Association, Boston
22. Ye Q, Zhou Y, Shi M, Sun Y, Lv J (2022) DLB: a dynamic load balance strategy for distributed training of deep neural networks. *IEEE Trans Emerg Top Comput Intell*. <https://doi.org/10.1109/TETCI.2022.3220224>
23. Krizhevsky A, Hinton G et al (2009) Learning multiple layers of features from tiny images
24. Li S, Zhao Y, Varma R, Salpekar O, Noordhuis P, Li T, Paszke A, Smith J, Vaughan B, Damania P et al (2020) Pytorch distributed: experiences on accelerating data parallel training. [arXiv:2006.15704](https://arxiv.org/abs/2006.15704) [cs.CV]
25. Gitman YYI, Ginsburg B (2017) Scaling SGD batch size to 32k for imagenet training. [arXiv:1708.03888](https://arxiv.org/abs/1708.03888) [cs.CV]
26. Li S, Walls RJ, Xu L, Guo T (2019) Speeding up deep learning with transient servers. In: *IEEE International Conference on Autonomic Computing (ICAC)*, pp 125–135. <https://doi.org/10.1109/ICAC.2019.00024>

27. Li S, Walls RJ, Guo T (2020) Characterizing and modeling distributed training with transient cloud GPU servers. In: IEEE 40th International Conference on Distributed Computing Systems (ICDCS), pp 943–953. <https://doi.org/10.1109/ICDCS47774.2020.00097>
28. Zheng S, Meng Q, Wang T, Chen W, Yu N, Ma Z-M, Liu T-Y (2017) Asynchronous stochastic gradient descent with delay compensation. In: International Conference on Machine Learning, vol 70, pp 4120–4129. PMLR. <https://proceedings.mlr.press/v70/zheng17b.html>
29. Ko Y, Kim S-W (2022) SHAT: a novel asynchronous training algorithm that provides fast model convergence in distributed deep learning. Appl Sci. <https://doi.org/10.3390/app12010292>
30. Zhang W, Gupta S, Lian X, Liu J (2016) Staleness-aware async-SGD for distributed deep learning. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, pp 2350–2356
31. Li S, Mangoubi O, Xu L, Guo T (2021) Sync-switch: hybrid parameter synchronization for distributed deep learning. In: IEEE 41st International Conference on Distributed Computing Systems (ICDCS), pp 528–538. <https://doi.org/10.1109/ICDCS51616.2021.00057>
32. Zhao X, Papagelis M, An A, Chen BX, Liu J, Hu Y (2019) Elastic bulk synchronous parallel model for distributed deep learning. In: IEEE International Conference on Data Mining (ICDM), pp 1504–1509. <https://doi.org/10.1109/ICDM.2019.00198>
33. Li S, Ben-Nun T, Girolamo SD, Alistarh D, Hoefer T (2020) Taming unbalanced training workloads in deep learning with partial collective operations. Association for Computing Machinery, New York, pp 45–61. <https://doi.org/10.1145/3332466.3374528>
34. Chen C, Weng Q, Wang W, Li B, Li B (2020) Semi-dynamic load balancing: efficient distributed learning in non-dedicated environments. Association for Computing Machinery, New York, pp 431–446. <https://doi.org/10.1145/3419111.3421299>
35. Chetlur S, Woolley C, Vandermersch P, Cohen J, Tran J, Catanzaro B, Shelhamer E (2014) cuDNN: efficient primitives for deep learning. [arXiv:1410.0759](https://arxiv.org/abs/1410.0759) [cs.CV]
36. Stich SU (2018) Local SGD converges fast and communicates little. [arXiv:1805.09767](https://arxiv.org/abs/1805.09767) [cs.CV]
37. He K, Zhang X, Ren S, Sun J (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp 770–778
38. Dosovitskiy A, Beyer L, Kolesnikov A, Weissenborn D, Zhai X, Unterthiner T, Dehghani M, Minderer M, Heigold G, Gelly S et al (2020) An image is worth 16x16 words: transformers for image recognition at scale. [arXiv:2010.11929](https://arxiv.org/abs/2010.11929) [cs.CV]
39. Ma N, Zhang X, Zheng H-T, Sun J (2018) Shufflenet v2: practical guidelines for efficient CNN architecture design. In: Proceedings of the European Conference on Computer Vision (ECCV)
40. Tan M, Le Q (2019) Efficientnet: rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning, pp 6105–6114

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Feixiang Yao¹ · Zhonghao Zhang¹ · Zeyu Ji¹ · Bin Liu^{1,2,3} · Haoyuan Gao¹

✉ Zeyu Ji
zeyu.ji@nwfufu.edu.cn

✉ Bin Liu
liubin0929@nwsuaf.edu.cn

Feixiang Yao
feixiang@nwsuaf.edu.cn

Zhonghao Zhang
2022051057@nwsuaf.edu.cn

Haoyuan Gao
gaohaoyuan@nwsuaf.edu.cn

- ¹ College of Information Engineering, Northwest A&F University, Yangling, China
- ² Key Laboratory of Agricultural Internet of Things, Ministry of Agriculture and Rural Affairs, Northwest A&F University, Yangling, China
- ³ Shaanxi Key Laboratory of Agricultural Information Perception and Intelligent Service, Northwest A&F University, Yangling, China