electronics

MDPI

Article

Distributed Deep Learning: From Single-Node to Multi-Node Architecture

Jean-Sébastien Lerat [1,2,*], Sidi Ahmed Mahmoudi [2] and Saïd Mahmoudi [2]

[1] Science and Technology Department, Haute École en Hainaut, 7000 Mons, Belgium
[2] Computer Science and Management Department, University of Mons, 7000 Mons, Belgium; sidi.mahmoudi@umons.ac.be (S.A.M.); said.mahmoudi@umons.ac.be (S.M.)
* Correspondence: jean-sebastien.lerat@umons.ac.be

Abstract: During the last years, deep learning (DL) models have been used in several applications with large datasets and complex models. These applications require methods to train models faster, such as distributed deep learning (DDL). This paper proposes an empirical approach aiming to measure the speedup of DDL achieved by using different parallelism strategies on the nodes. Local parallelism is considered quite important in the design of a time-performing multi-node architecture because DDL depends on the time required by all the nodes. The impact of computational resources (CPU and GPU) is also discussed since the GPU is known to speed up computations. Experimental results show that the local parallelism impacts the global speedup of the DDL depending on the neural model complexity and the size of the dataset. Moreover, our approach achieves a better speedup than Horovod.

Keywords: deep learning; frameworks; CPU; GPU; distributed computing

Citation: Lerat, J.S.; Mahmoudi, S.A.; Mahmoudi, S. Distributed Deep Learning: From Single-Node to Multi-Node Architecture. *Electronics* **2022**, *11*, 1525. https://doi.org/10.3390/electronics11101525

Academic Editors: Jose-Luis Sanchez-Romero, Antonio Jimeno-Morenilla, Antonio Martínez-Álvarez and Héctor Migallón

Received: 28 March 2022
Accepted: 9 May 2022
Published: 10 May 2022

1. Introduction

The explosion of data, in terms of computation capabilities, offers new options to analyze data through more complex models. Such models are Artificial Neural Networks (ANN) that are composed of several layers. These models represent the main component of the DL domain, which is a growing trend for both scientific research and enterprises that want to understand their data or automate tasks such as face recognition.

In DL, some particular tasks focus upon complex data, such as images and videos. Image and video classification are Machine Learning (ML) tasks that are carried out by models trained with data in order to recognize predefined identities in images such as animals or handwriting. DL performs well on these kinds of tasks. These DL tasks use a particular kind of layer, named Convolution, in their architecture. Convolution processes a 3D structure—an image that has red, green, and blue channels, identified by their row and column indices, on each of its pixels—in order to extract features such as edges in images. These features are used as inputs to classical Neural Networks.

To train a model for classification, two steps are required: (1) the learning, which involves training the model to fit the data, and (2) the validation, which involves evaluating the the model (e.g., accuracy). The training uses the forward propagation which works by successively applying each layer to an input in order to make a prediction regarding that input, and the back-propagation which consists of measuring the prediction error in order to update the model from the last layer to the first. The model can therefore be used to make predictions or to be tested, which uses only the forward propagation. This paper focuses on the learning phase because it is the most complex one, and includes the behavior of the testing phase. In addition, a use case of image classification is analyzed.

The increase in data also occurs in image and video classification tasks requiring the alternative processing of a significant amount of data. Such an alternative is called Distributed Computing, a well-known and developed field. Even if the scientific literature could successfully apply Distributed Computing in DL, no formal rules to efficiently process

data in terms of time exist. Therefore, the focus of the current work is how to efficiently distribute a DL task without having access to a dense GPU cluster because of the cost.

As a first step, this paper analyzes how to efficiently distribute a DL task in order to decrease the processing time. Multi-threading techniques have been well-optimized in libraries such as cuDNN and MKL-DNN, which are used by frameworks such as TensorFlow and PyTorch. Moreover, multi-processing techniques have been implemented in libraries such as Horovod [1]. This paper shows that taking into account local parallelism can speed up the DDL, does not require intense network communication, and outperforms Horovod. To this end, sequential processing was measured to serve as a baseline. Then the processing was parallelized on a single machine with different setups to choose the highest acceleration measured by a speedup metric. The best setup (i.e., less computing time) is used as a local strategy (i.e., on a single machine) in order to distribute the load across machines, resulting in an efficient DDL.

2. Background

An in-depth analysis of research papers [2] from 2012 to 2017 has been centered on the GPU. GPUs on multi-nodes have been predominant since 2015. Authors argue that this is the accelerating response to increasing workload with desired time constraints. Moreover, a report [3] shows that GPU is better than CPU for DL tasks, especially during matrix multiplication. Network communication is required to apply Distributed Computing in large-scale models. The more the calculation is divided, the more the network communication increases because of the gradient synchronization and data collection for pooling. Another result of this analysis concerns the mechanisms used to parallelize the learning task, called the communication layer. In the last analyzed year (2017), the communication layer was ensured by MPI, Socket, RPC, MapReduce, and Spark by decreasing order of usage rates. According to the analysis, MPI performs well when pooling occurs due to its sparse collective algorithm. The three strategies related to this work are:

Data parallelism [4–6]. In a mini-batch training task, computations are spread out on a set of size k before updating the model. Parallelism is implemented by concurrent computations on m distinct sets of k samples each. The drawback of this strategy is the necessity for the model to be replicated on each compute node.

Model (or network) parallelism [7–9]. In an ANN, the parameters are the weights and bias that are used to optimize the learning task. These parameters, which are distributed amongst the physical computer network (i.e., compute nodes), induce parallelization. The Inputs are formalized by a tensor and broadcast to the physical computer network such that each compute node can process the inputs. The advantage of this strategy is that it can handle huge models. However, it also induces significant network communication overhead due to the replication of inputs.

Hybrid parallelism [10,11]. To reduce the drawbacks of other strategies, the hybrid approach combines them. For instance, data parallelism can be used for convolutional layers and model parallelism for the fully connected layer of a CNN. This scheme requires specific implementations for a specific model.

DDL implies network communication, and this can become an issue for large-scale models because the network latency and load slow down the computations. Different approaches have been considered in order to decrease the network communication, particularly focused upon the synchronization of the gradient. The first method consists of designing a fast access memory [12,13] and making it available to compute nodes as shared memory. This requires high-speed network connection (high bitrate) and memory (high data transfer rate). Another approach consists of sending data to a subset of compute nodes. An algorithm that sends a local gradient to its direct neighbors has been proposed [14]. An alternative focuses on scheduling communication [15,16]. This is how one can quickly provide the gradient to all the compute nodes that have sent a request.

The last set of methods compresses data before sending them to the network layer. The cost of compression, however, is a reduction in accuracy.

A synchronous stochastic gradient descent [17] parallelized the training task on the CPUs with MPI, which increases the number of inputs processed per second, from a factor of 1.8 on two compute nodes to 6.4 on 16 compute nodes.

A large mini-batch [18] can be used without a loss of accuracy, which enables a larger distribution power to be achieved. By adapting learning parameters, it is possible to keep the same batch size on all nodes. This method was successfully applied on a dense GPU cluster [19] and on a CPU cluster [20].

Another approach to efficiently distribute the load is to split all dimensions [21,22] (sample, operator, attribute, parameter) and spread them across the compute nodes. Nevertheless, this approach does not take into account the network cost, because it does not take advantage of the fact that parallelism offers direct communication between the nodes on the same machine.

This work neither aims to show that DDL is capable of speeding up a DL task nor to decrease network traffic intensity in order to speed up DDL. Instead, it focuses on how local parallelism strategies—how each computer applies parallelism—impact the global acceleration of DDL. Local computer implementation is an aspect that is overlooked when designing a DDL task.

3. Materials and Methods

Our experimental setup is based on a benchmark [23] of DL frameworks as a baseline for speeding up calculation. The benchmark recommends using the pyTorch framework and focuses on two use cases that this work therefore also uses. This section is dedicated to explain and detail the DL task and the different setups used. In this paper, the acceleration factor called speedup is used and defined as $\frac{\bar{t}_b}{\bar{t}_s}$, where $\bar{t}_s$ and $\bar{t}_b$, respectively, stand for the average time of the setup and the baseline. The baseline depends on the device (CPU or GPU). It corresponds to the time needed to process the DL task sequentially on a single CPU core or GPU (see Appendix A for hardware specifications). Table 1 reports the baseline times depending of the device and the use case (see Section 3.1). For example, a parallelism setup which is two times faster than the baseline provides a speedup value of 2. The entire hardware and software configuration is provided in Appendix A.

Table 1. Baseline times (seconds).

Process Mode	ComplexSmall	SimpleBig
Sequential single-core CPU	629.73	879.38
Sequential single GPU	34.25	121.82

3.1. Use Cases

The evaluation of the speedup requires the training of DL networks that differ in complexity and datasets that differ in size. Without a loss of generality with regard to the type of neural networks, the current work focuses on convolutional neural networks (CNN), which are well adapted for image and video classification problems. These networks are well known and used, for example, by the community for the ImageNet Large Scale Visual Recognition Challenge. The public dataset comes from the Computer Sciences Department of the Faculty of Engineering of the University of Mons. The small version of the dataset is composed of 791 photos, and the big version is composed of 6003 photos. Each of these is split into three distinct classes: fire, smoke, and no fire. This dataset is used to generate a Deep Learning model for fire or smoke detection from images.

The two use cases are:

ComplexSmall uses the VGG16 [24] CNN architecture on the small dataset;

SimpleBig uses the AlexNet [25] CNN architecture on the large dataset.

The architectures have been adapted to support a three-class problem.

3.2. Training Task

The training task for the image classification problem has to address how to feed the neural network with images. The latter have to be pre-processed in order to fit the input required by neural networks (AlexNet and VGG16 require an input of 224×224). In this work, such pre-processing is designed based on the original publication of the selected CNN instead of designing the most accurate model. This is why input images are pre-processed to 224×224 with the 3 RGB channels. The goal is to measure and quantify the resource usage of a common learning task. The image pre-processing pipeline follows the sequence:

1. Image crop/scaling to 224×224;
2. Random horizontal flip transformation;
3. RGB-normalization with $\mu = (0.485, 0.456, 0.406)$, $\sigma = (0.229, 0.224, 0.225)$;
4. Conversion to a tensor data structure.

The optimizer is the mini-batch gradient descent with a learning rate $\alpha = 0.001$ and a momentum $\mu = 0.9$. The loss is computed with the cross-entropy method.

3.3. Setups

In this section, parallelism strategies are examined to determine when they can be applied and what the conditions are. In addition, the communication mechanisms on the network are discussed in order to design DDL.

3.3.1. Parallelism

Parallelism consists of the simultaneous execution of multiple computations. There are two main mechanisms in the CPU:

Multi-threading: a single application that runs only once as a process—a program loaded in memory—but that simultaneously executes blocks of instructions, with each block being a thread. The process memory is shared among all threads;

Multi-processing: the same application runs multiple times (on GNU/Linux systems, this application duplicates its whole execution context) and the operating system simultaneously executes each instance of the application, which is called concurrency. Each instance can be multithreaded.

Multiple simultaneous executions are carried out in data parallelism. The model is replicated into each execution, therefore increasing the amount of memory used. Each execution then loads the data and feeds its own Deep Learning tasks. After this step, synchronization of the gradient occurs and concurrency stops in order to ensure that all models—each execution—are identical in memory.

The model is split among available devices in model parallelism. This technique is required when the whole model cannot be loaded in memory on a single device. For example, a neural network is split on a computer composed of two GPUs. At time $t = 0$, the process loads a batch of data from the storage device. At time $t = 1$, the first GPU receives and feeds the first part of the neural network with the data while the process loads a second batch of data. At time $t = 2$, the second GPU receives and feeds the second part of the neural network with the output of the first GPU (i.e., the result of the first batch of data), the first GPU feeds the first part of the model with the second batch, and the process load a third batch of data. At the time $t = 3$, the output of the second GPU produces the final output and the gradient can be computed. Then the model is updated with the back-propagation algorithm. Finally, the training repeats the sequence and continues to load data and feeds the neural network. Pure model parallelism is less efficient than loading the model upon a single device because of the transfer overhead. Pipelining [26] the batch, however, makes the overhead less costly than the gain of parallelism, depending on the batch size and the model complexity. Pipelining on the batch consists of dividing

the batch into distinct sub-batches. Each sub-batch passes through each device, and therefore into each part of the model. While the first sub-batch gets processed by the second device—it has already been processed by the first device—the second sub-batch is being processed by the first device.

In our setups, model parallelism on GPU makes sense: while the CPU prepares data, the first GPU takes approximately the same time as the second GPU to process the respective data due to the split of matrix operations, GPUs having the same capabilities. This of course depends on the balance of the neural network distribution between the GPUs. Neural network architectures (AlexNet and VGG16) are divided into two equal parts (number of layers and synchronous operations such as pooling). AlexNet is split after the 4th convolution layer while VGG16 is split after the 8th convolution layer. On a CPU, such an approach would only result in overhead because it has to transfer data without any gain in processing data on another similar CPU. The CPU remains slow on matrix operations.

Because of the environment—the available hardware—multi-process data parallelism on a GPU is a bad idea. Multiple processes attempting to access the same GPU are actually discouraged by the hardware designer and usually results in memory overflow. The GPU API requires state information called a context. A context is linked to the data stored in the GPU and the process that uses the data. Multi-process can lead to a situation where the sum of the data linked to contexts overfit the GPU memory. To bypass this limitation, the designer proposes their multi-process service which acts as an interface between the GPU and the processes. Nevertheless, an official example of code (the file simpleIPC_mod.cu of CUDA) contains a comment about using multi-process: "Multiple processes per single device are possible but not recommended." This limitation only enables one process to be carried out per GPU. Data preparation on the CPU must be fast in order to quickly feed the GPU and to overcome this limitation of the process. In this setup, the difference between data and model parallelism resides in synchronization. In the data parallelism setup, after each result is received from the device, processes have to synchronize, in contrast with model parallelism which does not require synchronization, due to its sequential flow. Model parallelism efficiency depends on the transfer time and the waiting time between the synchronization mechanism of GPUs—i.e., when the GPU $i + 1$ has finished processing and waits for the result to be delivered from the GPU i.

The multithreaded data parallelism on a CPU is applied in a single process. Data parallelism is induced by splitting a batch of size n into k smaller batches of size $\frac{n}{k}$, each trained into k distinct threads. Each thread works on a replica of the CNN architecture.

3.3.2. Network Protocol Family

Instead of using a whole framework, including both a programming model and load distribution, a simple network protocol can be used to distribute a computation load. The advantage of using such a protocol is that it reduces the size of the software stack used and simplifies the execution, but this also increases the lines of code required for computation because of the need to use proper network functions. Four protocols are considered:

Socket: Using TCP/IP or UDP communication only. With UDP, network packets are smaller but if the network is fully used, data will be dropped, unlike in TCP/IP, which automatically adapts its behavior. In this kind of implementation, the developer has to design how and which information to send to other nodes;

Remote Procedure Call (RPC): A protocol designed to call a remote procedure or function with parameters. This protocol enables the abstraction of an underlying connection such as TCP/IP and easily allows executing a remote function;

Remote Direct Memory Access (RDMA): Enabling direct access to the memory of a remote computer without involving the operating system. It is characterized by a high throughput with low-latency networking. A disadvantage is that RDMA does not notify the remote computer that a request has been conducted. It is a single-sided method of communicating;

Message Passing Interface (MPI): Not only a protocol but also a norm that specifies how to send messages between remote computers. Like RPC, MPI offers an abstract layer to the developer, but is also able to efficiently send messages among a cluster of computers with different underlying technologies (e.g., RDMA, TCP/IP, ...).

An alternative is to implement the software inside a map-reduce framework. This is a parallelism design pattern enabling the manipulation of a large amount of data by spreading the data and the processing among a cluster. This pattern is well known and used by large companies such as Amazon and Facebook. Spark is a technology from 2014 built upon Hadoop and aimed at speeding up data processing. It runs the whole execution in RAM in real time, unlike Hadoop. It only uses persistent storage when the RAM is not sufficient.

3.4. Proposed Approach

This work assumes that a fast distributed process can be achieved if all computers involved in the DDL perform locally. That is to say that each computer executes the DL task with the parallelism paradigm that is faster on it. To identify the fastest parallelism and the configuration (e.g., number of threads), the speedup metric is used. Figure 1 is a diagram of two computers—the workers—that apply model parallelism, each on two GPUs. Between the nodes, data parallelism is used with the local replication of the datasets. After the processing by the GPU of a batch of data, the average gradient is synchronized through MPI messages. The transfer of data between GPUs on a same host does not require to go through the CPU if GPUs are interconnected. The pyTorch framework supports this feature.

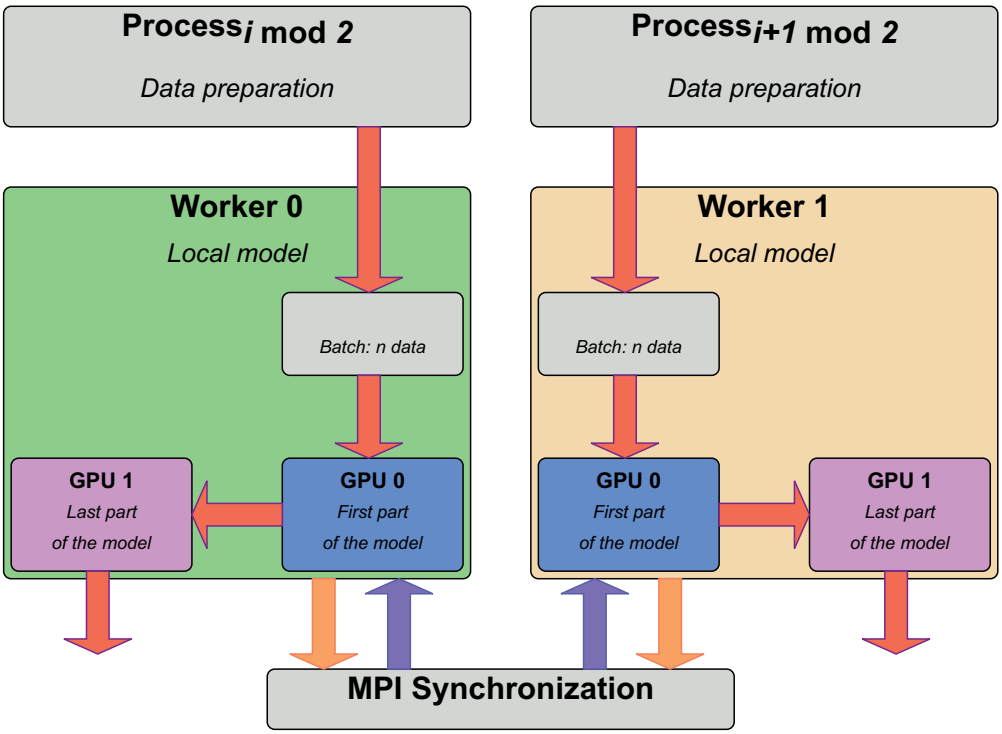


Figure 1. Diagram of DDL via MPI with local (green and orange areas) model parallelism. The worker 0 (resp. 1) has even (resp. odd) processes that load an independent non-overlapping batch of data from the same training set. Then model parallelism applies and produces the output. Instead of directly updating the model with the gradient, it is averaged locally and sent to the other worker. Then the gradient is averaged and the local model of each node is updated at the same time. That is, the two parts of the model that are stored on the two GPUs are updated.

The CPU and the RAM utilization rates are measured by the pidstat utility. They are respectively the total percentage of CPU time used by the task and the tasks currently used share of available physical memory. The other measurements are made with the sar utility.

4. Results

In this section, the behavior and speedup of the three parallelism strategies are discussed. Following the parallelization speedup measurements, the gain on a distributed implementation of the DL task is measured.

4.1. Parallel Deep Learning

Parallel DL involves how to perform the parallelism strategies on a single node.

4.1.1. Data Parallelism

Table 2 reports the speedup achieved by applying single or multi-process data parallelism on the CPU, with single process data parallelism per GPU. The reported speedups are the best values obtained by varying the number of processes which was 11 processes. A speedup occurs in the two use cases, with a larger acceleration achieved with the multi-process version on the CPU. The acceleration is stronger in the case of SimpleBig. This suggests that when there are more data, the parallelization accelerates the processing further. On the GPU, the speedup is not as great as it is in the case of the CPU. The worst case is SimpleBig, for which the more data there are, the more transfers there are to the GPU. This decreases the speedup.

Table 2. Speedup of data parallelism.

Process Mode	ComplexSmall	SimpleBig
CPU Single-Process	4	6.15
CPU Multi-Process	5.76	15.57
GPU Single-Process	1.57	1.25

Because of the system processes priorities—the scheduling of processes handled by operating systems—and thanks to the use of Python, using threads can only reduce the total amount of time allocated for data loading and learning tasks. Moreover, using Python threads allows pure concurrency tasks to be hindered by the Global Interpreter Lock (GIL). The GIL is a Python mechanism that synchronizes the execution of threads in order to ensure that only one native thread can be executed at a time. Furthermore, native operations—implemented in C, as in the pyTorch framework—executed in a thread can be released while still executing due to the GIL behavior. This is why a multi-process application can increase the speed:

Higher priority: Because the application priority depends on the number of all processes on the system, assuming that all processes have the same priority;

Avoid GIL contention: By training the CNN architecture in the main thread only, no GIL contention occurs. Pure concurrency can happen between distinct processes;

Multithreaded data preparation: This can be achieved for each process. A process will therefore repeat these steps in each epoch:

1. Loading a batch of data;
2. Training model: data are evaluated by the model and the gradient is calculated;
3. Synchronizing the model between processes.

Each thread—only used in step 1—is interpreted by Python only; therefore, no native operations are destroyed by the GIL. When a Python thread executes a system call for data, asking for data access to the operating system, it can wait and let another thread process it until data are available.

The evolution over time of the CPU, RAM, and the number of threads in Figure 2 shows that the CPU is fully utilized in a single process and multi-process. However, the thread allocation behavior differs, although on average both allocate the same number of threads. In the single process, the creation and destruction of threads give a high frequency of change

in the number of threads. This suggests that the framework uses one thread per image. The behavior is smoother in the multi-process. Additionally, the percentage of RAM utilization is smoother in the multi-process and requires less memory than in the single process. The ComplexSmall use case requires more RAM than the SimpleBig use case, which is explained by the fact that more parameters have to be maintained in memory.

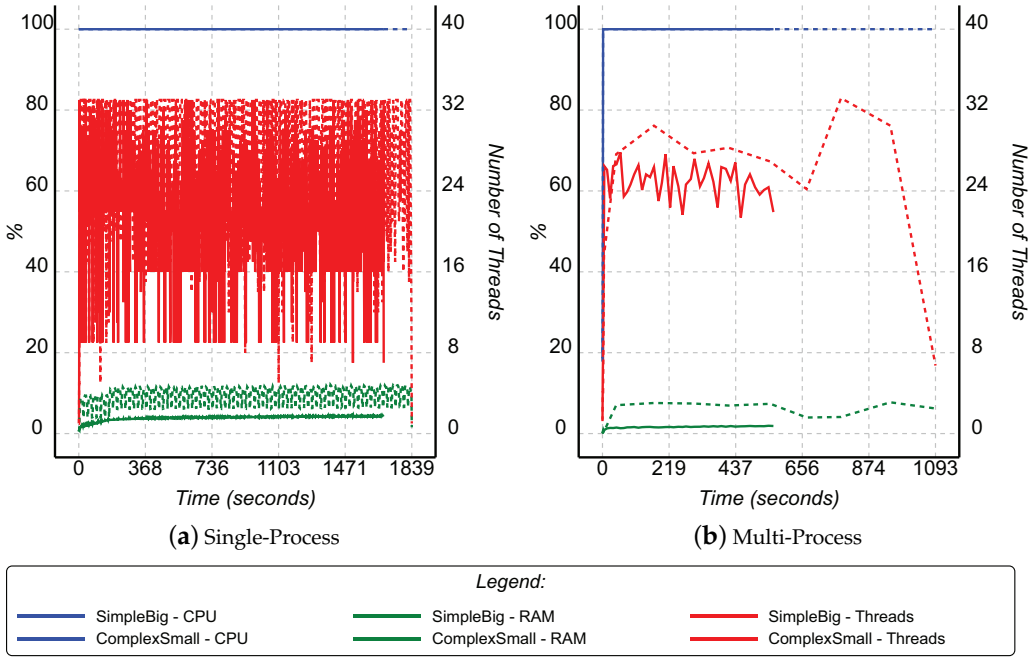


Figure 2. Resources utilization rates on CPU data parallelism. The x-axis is the time (seconds), the left y-axis is the percentage of the CPU (resp. RAM) utilization in blue (resp. green) and the right y-axis is the number of threads in red. The SimpleBig use case is shown in solid lines, and the ComplexSmall use case is shown in dashed lines. Figure 2a shows the results of the single process, while Figure 2b shows the results of the multi-process.

The DL task on the GPU behaves more constantly than in the CPU, as reported in Figure 3. When the DL task starts, the framework adapts its behavior by increasing its number of threads to an almost constant value of 13 and its CPU utilization to nearly 100%. The RAM utilization is lower than all CPU-based use cases except in the SimpleBig under multi-process data parallelism. This is explained by the fact that the model is stored in the GPU memory unlike previously. The exception occurs on the SimpleBig in multi-process data parallelism because the model is quite simple and there are fewer images per time unit loaded in the RAM.

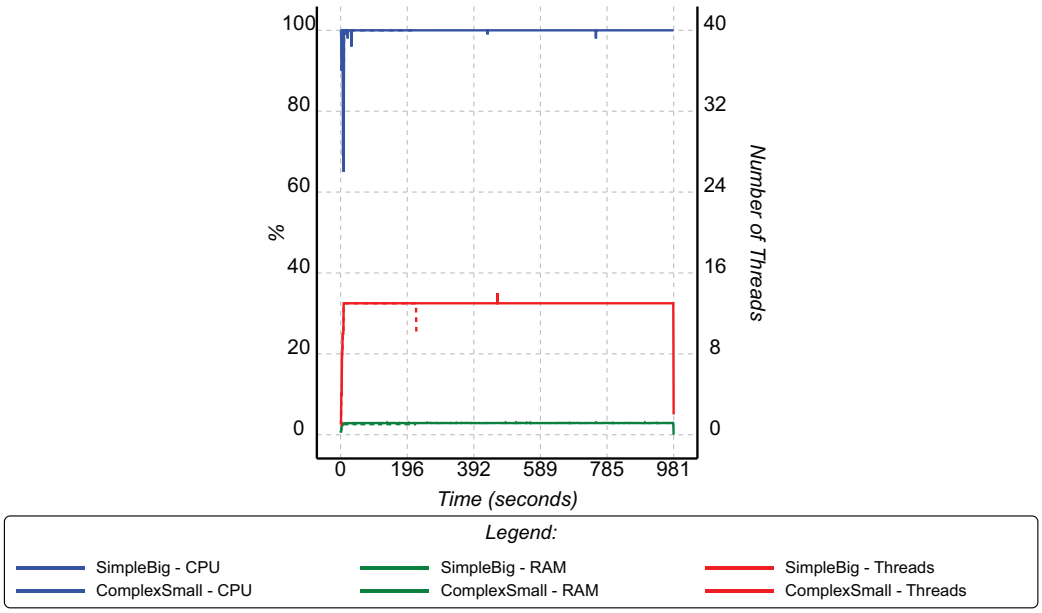


Figure 3. Resources utilization rates on GPU data parallelism. The x-axis is the time (seconds), the left y-axis is the percentage of the CPU (resp. RAM) utilization in blue (resp. green), and the right y-axis is the number of threads in red. The SimpleBig use case is shown in solid lines, and the ComplexSmall use case is shown in dashed lines.

4.1.2. Model Parallelism

Best parameters of the model parallelism setup on GPU with the pipeline is faster than in the previous setup, as shown in Table 3. Clearly, model parallelism outperforms data parallelism. Moreover, the hybrid approach is faster than data parallelism but slower than model parallelism. The number of processes used on the model parallelism (one process per GPU and other processes to prepare and load the data) is eight and four for the hybrid approach.

Table 3. Speedup of model and hybrid parallelism.

Parallelism	ComplexSmall	SimpleBig
Model	6.74	8.01
Data and Model	2.84	3.85

Figure 4a shows the number of threads and the utilization of the CPU and the RAM over time. As in the GPU data parallelism, the RAM is still used at a low percentage, but the number of threads varies little and remains around four. The utilization of the CPU highly varies in the case of the ComplexSmall use case but stays around 8% in the SimpleBig use case. Because the number of threads is quite similar in both use cases and the RAM utilization does not change, this behavior in terms of CPU comes from the data transfer from the CPU to the GPU and from the GPU to the CPU during the synchronization. Indeed, the model complexity is bigger in the ComplexSmall use case.

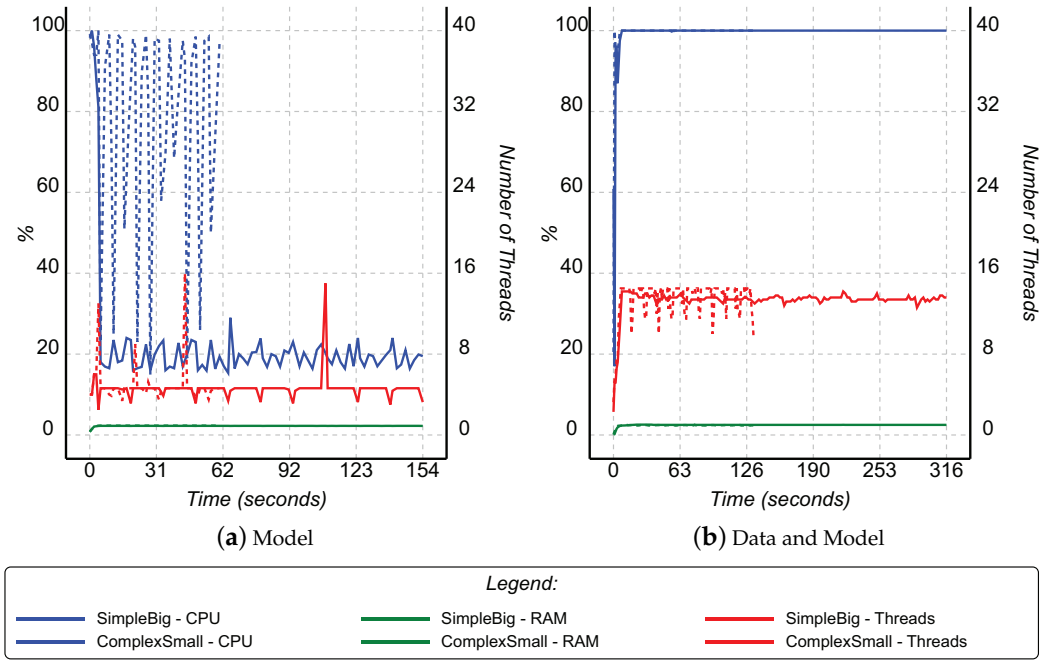


Figure 4. Resource utilization rates on the GPU model parallelism. The x-axis is the time (seconds), the left y-axis is the percentage of the CPU (resp. RAM) utilization in blue (resp. green), and the right y-axis is the number of threads in red. The SimpleBig use case is shown in solid lines, and the ComplexSmall use case is shown in dashed lines. Figure 4a shows the results of GPU model parallelism while Figure 4b shows the results of hybrid data model parallelism.

The hybrid approach is shown in Figure 4b. The behaviors differ from the model parallelism and become more similar to the GPU data parallelism. In the two use cases, the CPU becomes used at nearly 100% but the DL task does not require more RAM. The average number of threads becomes 14. Note that this approach requires more space on each GPU because of the model replication. This limitation implies that the batch size must be smaller than in the previous setup, so that the GPU can handle it. Fewer data (i.e., a smaller batch) are processed per time unit, meaning that more processes would be useful in order to counter this effect. Nevertheless, more processes induce more replications and, therefore, more memory requirements. This is why the batch size is reduced, allowing the model replication and fewer processes to be handled, compared to the previous setup.

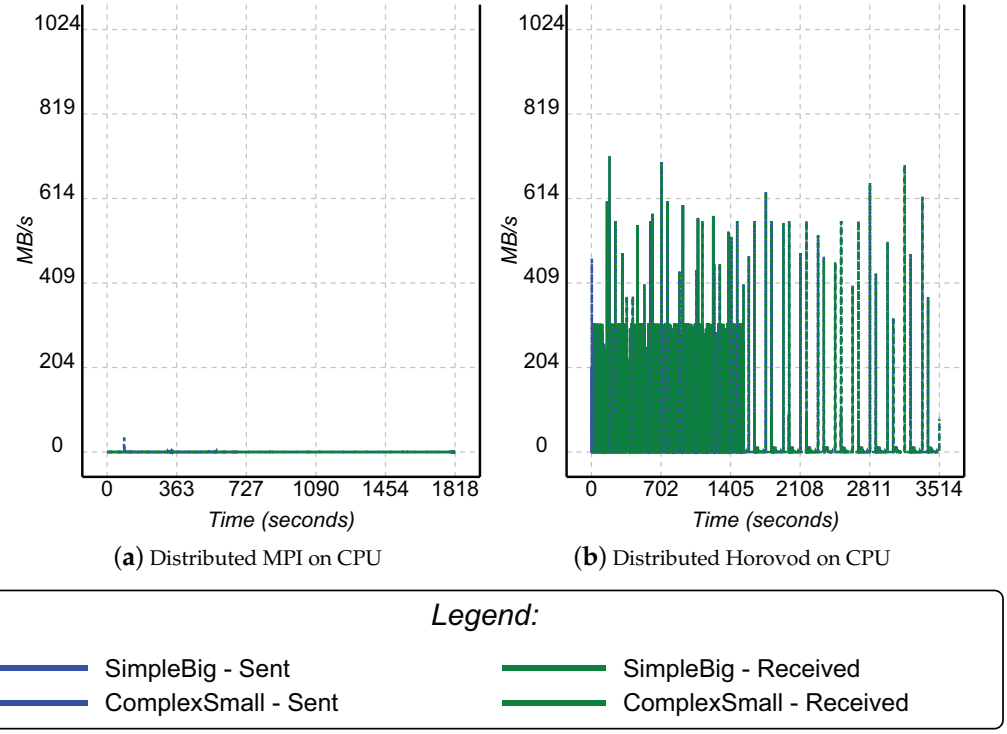
4.2. Distributed Deep Learning

The DDL task is executed using three methods: the proposed approach implementing the distribution of load with MPI; the Spark framework; and the Horovod API, which was designed to perform on DDL.

Table 4 shows the speedup results achieved by Distributed Computing technologies. Spark is only executed on the CPU because Spark does not natively support the GPU. Clearly, the Spark implementation is less efficient than a single node parallelized version. The acceleration even becomes negative in the SimpleBig use case. The more data there are, the harder it is for Spark to perform the processing quickly. Horovod can speed up the process in all cases and outperform Spark, but it fails to halve the computation time. This can be explained by the intensity of the network traffic shown in Figure 5. The proposed approach avoids network communication which is slower than the exchange of data in RAM. By optimizing parallelism on the computers, less network traffic is required and the DDL can process faster.

Table 4. Speedup of Distributed Deep Learning.

Technology	ComplexSmall		SimpleBig	
	CPU	GPU	CPU	GPU
MPI	12.11	4.13	26.62	11.79
Spark	1.14	-	0.53	-
Horovod	1.79	1.91	5.78	1.57



(**a**) Distributed MPI on CPU (**b**) Distributed Horovod on CPU

Figure 5. Network traffic on two computers with multiprocessing and multithreading capabilities. The x-axis shows the time (seconds) and the y-axis shows the network traffic (mega bytes per second) of sent data in blue and received data in green. The SimpleBig use case is in solid lines, and the ComplexSmall use case is in dashed lines. Figure 5a is the local parallelism optimization approach proposed in this paper and Figure 5b is Horovod, the state-of-the-art framework.

The MPI version executed provides the best speedup and outperforms Horovod. On the CPU, the best speedup on the two use cases is achieved with DDL. This is not the case on the GPU. With enough data, which is the SimpleBig use case, the speedup reaches a value of 11.79 that is better than the GPU model parallelism. The ComplexSmall use case still accelerates the process but less than the GPU model parallelism. The amount of data is not sufficient to balance the cost of the network synchronization. The network communication was analyzed during the DL tasks, revealing that no bottleneck occurred. A peak usage of 35 KiB was observed on a dedicated 10 GB Ethernet connection.

The best local parallelism strategy on the CPU was multi-process data parallelism with a utilization of around 100% of the CPU. This strategy is used in the distributed version but the CPU becomes very inactive after a period, as shown in Figure 6. At the beginning, the CPU loads the input data from the storage in the RAM. At that point, the CPU is active. After all the data have been loaded, the CPU becomes mostly inactive because it has to update the model then apply gradient synchronization through the network, which has latency. This latency causes the CPU to wait a response and to be mostly inactive. Nevertheless, the CPU spent a low percentage of time (around 0% as shown in Figure 6) in the IOwait state, which means that it efficiently loads the input data into RAM while minimizing the latency due to the read operations on the storage device. The behavior

on the GPU differs because the CPU has to transfer data from the CPU to GPU and from the GPU to the CPU.

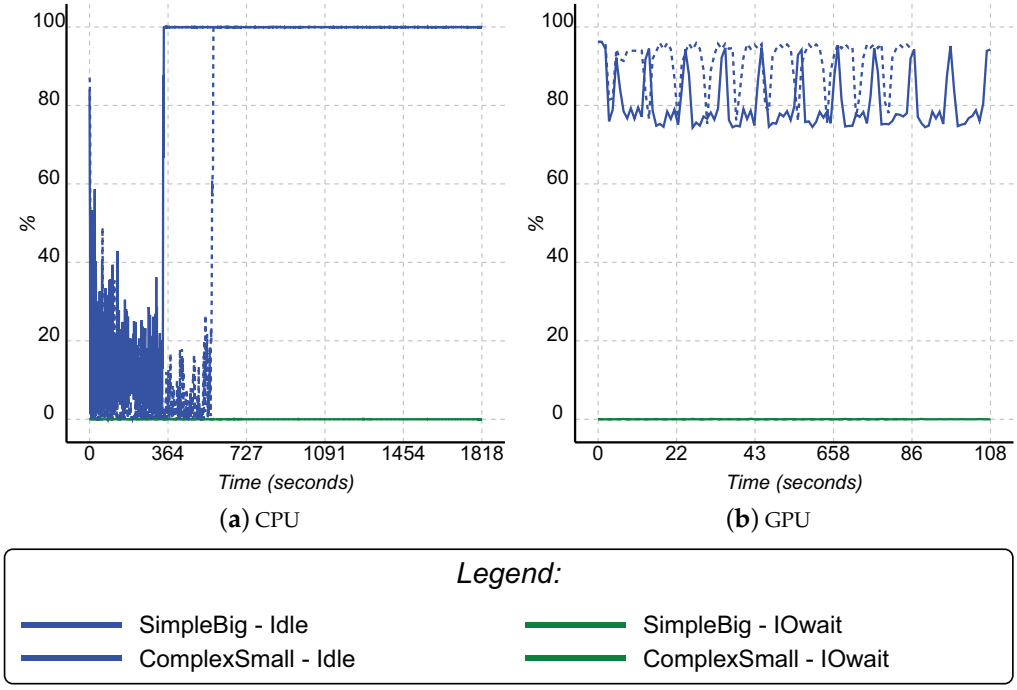


Figure 6. Resource utilization rates on the MPI distributed version. The x-axis shows the time (seconds) and the y-axis shows the percentage of time that the CPU is inactive in blue, the CPU was idle during which the system had an outstanding disk I/O request in green. The SimpleBig use case is shown in solid lines, and the ComplexSmall use case is shown in dashed lines. Figure 6a shows the results of the setup running on the CPU, while Figure 6b shows the results on the GPU.

5. Discussion

DDL is a tool to speed up the execution of a DL task. In the scientific literature [2], works distribute the workload on machines mainly using MPI or Apache Spark. As explained in Section 2, the problem of DDL is the cost of network communication because the more the task is distributed to compute nodes, the more the network load increases. In this paper, the distribution of DL tasks is performed with MPI which has been shown to perform better [23] than tools based on the MapReduce paradigm (e.g. Apache Spark). Usually, MPI is used to distribute the computing load over the entire infrastructure. Therefore, there are several computing processes on the same machine. This is not the case in this work, MPI runs only one computing task on a single machine. The computing task will itself create several processes/threads in order to allow them to exchange information directly. For example, on the GPU, the data will be associated with a single execution context. This avoids loading and unloading data unnecessarily.

The DDL method used is based on the assumption that a fast local (on each machine) parallelization allows us to speed up the whole distributed task. This is why it is important to understand: how best to parallelize locally. To this end, data parallelism, model parallelism, and a hybrid approach have been considered. Parallelism was applied on the CPU and the GPU with two use cases to highlight the effect of a complex model and the effect of the amount of data. The results have shown that DDL can be slower than pure parallelism on small datasets. On a single computer, the multi-process data parallelism is faster on the CPU and the model parallelism is faster on the GPU.

There are studies that have already focused on how to decompose a DL task in a distributed way by exploiting different parallelization methods including a hybrid approach [27]. The problem is that by doing this, the network communication is intensified. For example when the neural network is divided [28] on several machines. Each machine has to communicate both

to exchange the gradient but also to exchange data between the different parts of the model. However, network communication is the major problem of DDL. Several methods [14–16,29–32] can be used to reduce the amount of network traffic, but this comes at a cost in terms of accuracy.

The proposed approach in this paper takes advantage of parallelization and distributes the load using MPI. The results of this DDL methodology show not only that the computation has been accelerated but also that the required network communication is low, in contrast to the state-of-the-art framework called Horovod [1]. In addition, Horovod is outperforming in terms of acceleration. Future work focuses on a larger scale distribution and the effect that data location has on DDL speedup.

6. Conclusions

In this paper, a novel way to speed up the Distributed Deep Learning has been proposed by focusing on how parallelism is implemented on computers. This customized approach speeds up calculations, saves time and reduces network communication. Future works are to exploit the approach of this paper on a cloud and an edge computing infrastructure, as well as to propose a deployment method for Distributed Deep Learning. It is also interesting to understand how the location of the data affects the results.

Author Contributions: Investigation, J.-S.L.; supervision, S.A.M. and S.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially funded by Fédération Wallonie-Bruxelles (JCM/TP/BS/mo/c999).

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article. Dataset is available at https://github.com/Belegkarnil/forestfire (accessed on August 15, 2021).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

GPU	Graphics Processing Unit
ANN	Artificial Neural Networks
CNN	Convolutional Neural Networks
CPU	Central Processing Unit
DDL	Distributed Deep Learning
DL	Deep Learning
GIL	Global Interpreter Lock
ML	Machine Learning
MPI	Message Passing Interface
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RPC	Remote Procedure Call

Appendix A. Hardware and Software Configuration

The two physical computers are composed of the following devices:

- Graphic card: 2× EVGA GeForce RTX 2080 Ti 11 GB GDDR6 with emphEVGA NVLink
- Power supply: Seasonic Prime 1300 W Gold
- Case: Corsair Obsidian 750 D Airflow
- CPU: AMD 2950 X Ryzen Threadripper 16 cores and 32 threads
- CPU Cooling system: Watercooling Enermax LIQTECH TR4 II 360 RGB
- Additional case FAN: be quiet! Silent Wings 3 120 mm PWM High-Speed with SilverStone FF122
- Additional device: DVD ASUS DRW-24D5MT
- Motherboard: ASRock Fatal1ty X399 Professional Gaming
- RAM: Corsair Vengeance LPX Series Low Profile 128 GB (8 × 16 GB) DDR4 2933 Mhz

- Storage:
  - Samsung SSD 970 PRO M.2 PCIe NVMe 1 To
  - Samsung SSD 860 EVO 4 To
- Network: computers are directly linked with their 10 GBASE-T Ethernet connector (AQUANTIA Marvell 10 LAN Gigabit), external network is on the 1000 Base-T Ethernet connector (Intel I211-AT)

The operating system is an Ubuntu 20.04 LTS with the specific software of Table A1 and the Python 3 packages of Table A2. The virtual python environment of each framework is reported in Table A3 for the CPU and the GPU.

Table A1. Installed software.

Python Related Setup	NVidia Software
python 3.9.7	nsight-compute 2021.3.0
pip3 9.0.1	nsight-systems 2021.5.1
pbr 5.4.4	libcudnn8_8.2.4 + cuda11.3
stevedore 3.4.0	cuda-repo-ubuntu_11.3
virtualenv-clone 0.5.7	tensorRT 8
virtualenvwrapper 4.8.4	

Table A2. Common pip3 package.

numpy 1.21.3	tensorboard 2.6.0
Pillow 8.4.0	tensorflow-estimator 2.6.0
opencv-python 4.5.4.58	termcolor 1.1.0
Keras 2.6.0	tf-slim 1.1.0
Keras-Applications 1.0.8	tf2cv 0.0.18
Keras-Preprocessing 1.1.2	

Table A3. The pip3 packages used for the framework installation.

Framework	CPU	GPU
pyTorch	torch 1.8.2 + cpu	torch 1.8.2
pyTorch	torchvision 0.9.2 + cpu	torchvision 0.9.2
TensorFlow	tensorflow 2.5.2	tensorflow-gpu 2.5.2

References

1. Sergeev, A.; Del Balso, M. Horovod: Fast and easy distributed deep learning in TensorFlow. *arXiv* **2018**, arXiv:1802.05799.
2. Ben-Nun, T.; Hoefler, T. Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis. *ACM Comput. Surv. (CSUR)* **2019**, *52*, 1–43.
3. Hegde, V.; Usmani, S. *Parallel and Distributed Deep Learning*; Technical Report; Stanford University: Standford, CA, USA, 2016; pp. 1–8.
4. Jiang, J.; Cui, B.; Zhang, C.; Yu, L. Heterogeneity-aware distributed parameter servers. In Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, IL, USA, 14–19 May 2017; pp. 463–478.
5. Zhang, K.; Chen, X.W. Large-Scale Deep Belief Nets with MapReduce. *IEEE Access* **2014**, *2*, 395–403.
6. Le, Q.V.; Ngiam, J.; Coates, A.; Lahiri, A.; Prochnow, B.; Ng, A.Y. On optimization methods for deep learning. In Proceedings of the 28th International Conference on Machine Learning, Bellevue, WA, USA, 28 June–2 July 2011.
7. Dryden, N.; Maruyama, N.; Benson, T.; Moon, T.; Snir, M.; Van Essen, B. Improving strong-scaling of CNN training by exploiting finer-grained parallelism. In Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), Tianjin, China, 4–6 December 2019; pp. 210–220.
8. Coates, A.; Huval, B.; Wang, T.; Wu, D.; Catanzaro, B.; Andrew, N. Deep learning with COTS HPC systems. In Proceedings of the International Conference on machine learning, Atlanta, GA, USA, 17–19 June 2013; pp. 1337–1345.
9. Lee, S.; Purushwalkam, S.; Cogswell, M.; Crandall, D.; Batra, D. Why m Heads Are Better Than One: Training a Diverse Ensemble of Deep Networks. *arXiv* **2015**, arXiv:1511.06314.

10. Krizhevsky, A. One Weird Trick for Parallelizing Convolutional Neural Networks. *arXiv* **2014**, arXiv:1404.5997.
11. Ben-Nun, T.; Levy, E.; Barak, A.; Rubin, E. Memory access patterns: The missing piece of the multi-GPU puzzle. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2015; pp. 1–12.
12. Lim, E.J.; Ahn, S.Y.; Choi, W. Accelerating training of DNN in distributed machine learning system with shared memory. In Proceedings of the 2017 International Conference on Information and Communication Technology Convergence (ICTC), Jeju Island, Korea 18–20 October 2017; pp. 1209–1212.
13. Lim, E.J.; Ahn, S.Y.; Park, Y.M.; Choi, W. Distributed deep learning framework based on shared memory for fast deep neural network training. In Proceedings of the 2018 International Conference on Information and Communication Technology Convergence (ICTC), Jeju, Korea, 17–19 October 2018; pp. 1239–1242.
14. Cong, G.; Bhardwaj, O. A hierarchical, bulk-synchronous stochastic gradient descent algorithm for deep-learning applications on GPU clusters. In Proceedings of the 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), Cancun, Mexico, 18–21 December 2017; pp. 818–821.
15. Hashemi, S.H.; Jyothi, S.A.; Campbell, R.H. TicTac: Accelerating Distributed Deep Learning with Communication Scheduling. *arXiv* **2018**, arXiv:1803.03288.
16. Tsai, C.Y.; Lin, C.C.; Liu, P.; Wu, J.J. Communication scheduling optimization for distributed deep learning systems. In Proceedings of the 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), Singapore, 11–13 December 2018; pp. 739–746.
17. Das, D.; Avancha, S.; Mudigere, D.; Vaidynathan, K.; Sridharan, S.; Kalamkar, D.; Kaul, B.; Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv* **2016**, arXiv:1602.06709.
18. Goyal, P.; Dollár, P.; Girshick, R.; Noordhuis, P.; Wesolowski, L.; Kyrola, A.; Tulloch, A.; Jia, Y.; He, K. Accurate, large minibatch SGD: training ImageNet in 1 hour. *arXiv* **2017**, arXiv:1706.02677.
19. Jia, X.; Song, S.; He, W.; Wang, Y.; Rong, H.; Zhou, F.; Xie, L.; Guo, Z.; Yang, Y.; Yu, L.; et al. Highly scalable deep learning training system with mixed-precision: training ImageNet in four minutes. *arXiv* **2018**, arXiv:1807.11205.
20. You, Y.; Zhang, Z.; Hsieh, C.J.; Demmel, J.; Keutzer, K. Imagenet training in minutes. In Proceedings of the 47th International Conference on Parallel Processing, Eugene, OR, USA, 13–16 August 2018; pp. 1–10.
21. Jia, Z.; Lin, S.; Qi, C.R.; Aiken, A. Exploring hidden dimensions in parallelizing convolutional neural networks. In Proceedings of the 35th International Conference on Machine Learning, Stockholm, Sweden, 10–15 July 2018; pp. 2279–2288.
22. Jia, Z.; Zaharia, M.; Aiken, A. Beyond Data and Model Parallelism for Deep Neural Networks. *Proc. Mach. Learn. Syst.* **2019**, *1*, 1–13.
23. Lerat, J.S.; Mahmoudi, S.A.; Mahmoudi, S. Single node deep learning frameworks: Comparative study and CPU/GPU performance analysis. In *Concurrency and Computation: Practice and Experience*; John Wiley and Sons Ltd: Chichester, United Kingdom, 2021; p. e6730.
24. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2015**, arXiv:1409.1556v6.
25. Krizhevsky, A.; Sutskever, I.; Hinton, G.E. Imagenet classification with deep convolutional neural networks. *Adv. Neural Inf. Process. Syst.* **2012**, *25*, 1097–1105.
26. Huang, Y.; Cheng, Y.; Bapna, A.; Firat, O.; Chen, M.X.; Chen, D.; Lee, H.; Ngiam, J.; Le, Q.V.; Wu, Y.; et al. GPipe: Efficient training of giant neural networks using pipeline parallelism. *arXiv* **2019**, arXiv:cs.CV/1811.06965.
27. Teerapittayanon, S.; McDanel, B.; Kung, H.T. Distributed deep neural networks over the cloud, the edge and end devices. In Proceedings of the 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), Atlanta, GA, USA, 5–8 June 2017; pp. 328–339.
28. Disabato, S.; Roveri, M.; Alippi, C. Distributed Deep Convolutional Neural Networks for the Internet-Of-Things In *IEEE Transactions on Computers*; IEEE Computer Society: Washington, DC, United States **2021**, *70*, 1239–1252.
29. Wen, W.; Xu, C.; Yan, F.; Wu, C.; Wang, Y.; Chen, Y.; Li, H. Terngrad: Ternary Gradients to Reduce Communication in Distributed Deep Learning. *Adv. Neural Inf. Process. Syst.* **2017**, *30*, 1509–1519.
30. Sattler, F.; Wiedemann, S.; Müller, K.R.; Samek, W. Sparse binary compression: Towards distributed deep learning with minimal communication. In Proceedings of the 2019 International Joint Conference on Neural Networks (IJCNN), Budapest, Hungary, 14–19 July 2019; pp. 1–8.
31. Kuang, D.; Chen, M.; Xiao, D.; Wu, W. Entropy-based gradient compression for distributed deep learning. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications, Hyderabad, India, 17–20 November 2019; pp. 231–238.
32. Li, S.; Hoefler, T. Near-Optimal Sparse Allreduce for Distributed Deep Learning. *arXiv* **2022**, arXiv:2201.07598.