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

Preprint · January 2022

CITATIONS
0

READS
14

3 authors:

Jean-Sébastien Lerat
Haute École en Hainaut
7 PUBLICATIONS **171** CITATIONS
SEE PROFILE

Sidi Ahmed Mahmoudi
Université de Mons
116 PUBLICATIONS **1,203** CITATIONS
SEE PROFILE

Mahmoudi Saïd
Université de Mons
218 PUBLICATIONS **2,439** CITATIONS
SEE PROFILE

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

Anonymous Authors[1]

Abstract

Distributed Deep Learning (DDL) is using a multi-node architecture to apply Deep Learning (DL) counter to Federated Deep Learning (FDL) where entities keep their data and contribute to a common DL task like training a model. The more nodes there are, the more network traffic increases in DDL which requires more time to distribute the load and to apply DL. The state of the art focuses on how to decrease the network traffic but none of them studies how the local parallelism strategy can speedup a multi-node approach. This paper takes an empirical approach to measure the speedup of DDL by using different parallelism strategies on the nodes. Taking into account local parallelism is quite important in order to design a time performing multi-node architecture because DDL depends on the time required by all the nodes. We also address the impact of the computational resource namely the Central Processing Unit (CPU) and the Graphics Processing Unit (GPU) because GPU is known to speedup computations. The 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.

1. Introduction

The explosion of data, as far as the computation capabilities, offer new perspective to analyze data through more complex models. Such models are Artificial Neural Networks composed of several layers. Theses models represent the main component of Deep Learning domain, which is a growing trend for both scientific research and enterprises that want to understand their data or to automate a task like face recognition.

In Deep Learning, some particular tasks focus upon complex data, as are images and videos. Images and Videos Classification is a Machine Learning task that is trained with data in order to recognize predefined identities on images like animals or handwriting. Deep Learning performs well on these kinds of tasks. These Deep Learning tasks use a particular kind of layer, named Convolution, in their architecture. Convolution processes a 3D structure – an image has red, green and blue channels, identified by their row and column indices, on each of its pixel – in order to extract features like edges in images. These features are used as input to classical Neural Networks.

Training a model for classification is composed of two steps: (1) the learning phase which is training the model to fit data and (2) the testing phase which is evaluating the accuracy of the model. The model therefore can be used to make predictions. In Deep Learning, the prediction, called the forward pass, is used in both phases. This works by successively applying each layer on an input in order to make a prediction on that input. In the learning phase, a second step, which is called backward propagation, is also applied. Backward propagation consists in measuring the prediction error in order to update the model from the last layer to the first. In this paper, we will focus on the learning phase because it is the most complex one, and include the behavior of the testing phase. Moreover, we analyze an Image Classification use case.

The increase of data also occurs on images and videos classification tasks requiring alternative processing of this signification amount of data. A such alternative is Distributed Computing, a well known and developed field. Even if the scientific literature successfully applied Distributed Computing on Deep Learning, no formal rules exist to efficiently process data in terms of time. This is the focus of this current work: how to distribute efficiently a DL task.

As a fist step, this paper consists in analyzing how to efficiently distribute a Deep Learning task in order to decrease the processing time. To this end, we have considered a baseline with sequential processing, then parallelized the processing on a single machine with different setups to choose the bigger speedup. The best setup (i.e. less compute time) is used as a local strategy (i.e. on a single machine) in order

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

to distributed the load across machines, resulting in an efficient Distributed Deep Learning. The following sections provide with a detailed discussion about the questions we are investigating.

2. Related Work

An in-depth analysis of research papers (Ben-Nun & Hoefler, 2019) from 2012 to 2017 shows the evolution of DDL. Since 2013, the main hardware processor unit is the GPU. GPUs on multi-nodes becomes preponderant since 2015. Authors argue that it is the acceleration response to increasing workload with desired time constraints. Moreover a report (Hegde & Usmani, 2016) shows that GPU is better than CPU for deep learning task, especially during matrix multiplication. Network communication is an effect of the parallelization of 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 that 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 rate. 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 In a mini-batch training task, computations are spread up on a set of 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 up on each compute node.

Model (or network) parallelism In an ANN, parameters are used to optimize the learning task (e.g. weights). These parameters, distributed amongst the network (i.e. compute nodes), induce the parallelization. Inputs are formalized by a tensor and broadcasted to the network in such way that each compute node can process them. The advantage of this strategy is that it can handle huge models. But it also induces significant network overheads due to the replication of inputs.

Hybrid parallelism combines previous strategies in a way that reduces their respective drawbacks. 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 implementation for a specific model.

DDL implies network communication, and that becomes 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 in designing a fast access memory (Lim et al., 2017; 2018) and making it available to compute nodes as shared memory. This requires speed network connection (high bitrate) and speed memory (high data transfer rate). Another approach consists in sending data to a subset of compute nodes. An algorithm that sends local gradient to its direct neighbors has been proposed (Cong & Bhardwaj, 2017). An alternative focuses up on scheduling the communication (Hashemi et al., 2018; Tsai et al., 2018). The issue focuses upon the way to quickly provide all the requesting compute nodes with the gradient. The last set of methods compress data before sending them to the network layer. The cost of compression is a reduction of accuracy. These methods are based upon two generic principles of communication reduction:

Gradient Quantization requires to quantify gradients in order to reduce their size, leading to a loss of precision. In this setup, fewer bits are sent onto the network but the challenge is to achieve satisfactory model accuracy. The Hadamard product performs the gradient quantization in an efficient way (Wen et al., 2017), data being processed two times faster with a maximum loss of accuracy of 2.7%.

Gradient Sparsification only sends a subset of parameters to be updated. A heuristic selection of the weights to be updated, based upon the magnitude of weights during the update step, has been proposed (Sattler et al., 2019). The bigger lost of accuracy (-0.9%) occurs when the compression rate is the most significant ($\times 37208$). Another recent gradient sparsification is (Kuang et al., 2019)

These methods allow to reasonably deal with large scale models and to distributed them amongst a large cluster of compute nodes. The most efficient methods in terms of time are those that have a cost on accuracy.

DDL was successfully applied without compression mechanisms. A synchronous stochastic gradient descent (Das et al., 2016) parallelized the training task on CPU with MPI which increases the number of inputs processed by second, from a factor 1.8 on two compute nodes to 6.4 on 16 compute nodes.

SparkNet (Moritz et al., 2015) is an architecture aware of the fact that Spark is not designed to support asynchronous and communication intensive tasks. It is important to remind that this kind of tasks is the major characteristic of DDL. A test is made with the Caffe framework with synchronous learning on GPU. SparkNet outperforms simple data parallelism in term of speedup but shows off its limits

when too much time is required to synchronize the gradient . Consequently, SparkNet becomes less efficient when it comes to parallelization. Spark is an easy solution to speedup learning with a small number of nodes, especially when no methods to restrain communication data are used, like in a time series use-case (Hussain et al., 2018).

In this work we neither consider to reduce the network traffic nor to show that DDL can speedup a DL task but how local parallelism strategies can speedup the DDL. Local node implementation is an aspect that is overlooked when designing a DDL task.

3. Methods

Our experimental setups is based on a benchmark (Lerat et al., 2021) of DL frameworks as a baseline for speedup calculation. The benchmark recommends to use the pyTorch framework and focuses on two use cases that this work therefore also uses. In this section we explain the DL task and all the setups.

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 loss of generality on 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 datasets 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 composed of 6,003 photos. Each of these are 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 is using the VGG16 (Simonyan & Zisserman, 2015) CNN architecture on the small dataset.

SimpleBig is using the AlexNet (Krizhevsky et al., 2012) CNN architecture on the big dataset.

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

3.2. Training Task

The training task of the image classification problem has to take care of how to feed the neural network with images. The latter have to be preprocessed in order to fit the input required by neural networks. 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 we pre-process input images to a 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 sequences:

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 tensor data structure

The optimizer is the stochastic gradient 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 we discuss about which parallelism strategy can be applied and what are the conditions. Moreover we also discuss about the network communication in order to design DDL.

3.3.1. Parallelism

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

Multi-threading a single application runs only once as a process – a program loaded in memory – but is simultaneously executing blocks of instructions, each block being a `thread`. The process memory is shared among all threads.

Multi-processing the same application runs multiple times[1] and the operating system simultaneously executes each instance of the application, which is called `concurrency`. Each instance can be multi-threaded.

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. This process is illustrated in Figure **??**.

[1] On GNU/Linux systems, this application duplicates its whole execution context.

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. The Figure **??** illustrated this process under two GPUs. The process loads a batch of data. These data are therefore sent to the first device which applies the first part of the model. Other data are then sent to the second device which applies the second part of the model and produces the final output. Pure model parallelism is less efficient than loading the model upon a single device because of the transfer overhead. Pipelining (Huang et al., 2019) the batch however makes the overhead less costly than the gain of parallelism, depending of the batch size and the model complexity. Pipelining on the batch consists in 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 CPU prepares data, the first GPU takes approximately the same time than the second GPU to process their respective data due to the split of matrix operations, GPUs having the same capabilities. On 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. CPU remains slow on matrix operations.

Because of the environment – the available hardware –, data parallelism with multi process on GPU is a bad idea. Multiple processes trying to access the same GPU is actually discouraged by NVIDIA – the designer – and usually results in memory overflow. This limitation only enables one process per GPU. Data preparation on CPU must be fast in order to quickly feed the GPU and to overcome this limitation of process. In this setup, the difference between data and model parallelism resides in synchronization. In the data parallelism setup, after each result received from the device, processes have to synchronize, at the opposite of model parallelism that does not require synchronization, due to its sequential way. Model parallelism efficiency depends of 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 delivered from the GPU i.

The multi-threaded data parallelism on 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.

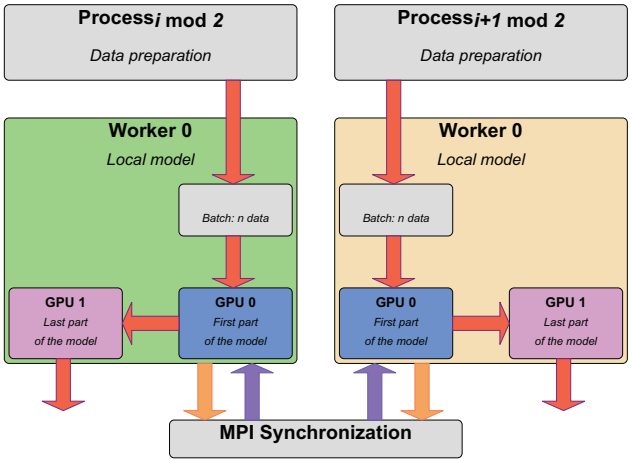


Figure 1. Diagram of DDL via MPI with local (green and orange areas) model parallelism. Each node exchanges MPI messages to update the gradient.

3.3.2. Network Protocol Family

Instead of using a whole framework including both 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 software stack used and simplifies the execution, but also increases the lines of code required for computation because of the need to call proper network functions. Four protocols are considered:

Socket using TCP/IP or UDP communication only. With UDP, network packet are smaller but if the network is fully used, data will be dropped counter to TCP/IP that will 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 to abstract underlying connection like TCP/IP and easily allows to execute 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 high-throughput with low-latency networking. A disadvantage is that RDMA does not notify the remote computer that a request has be done. It is a single-sided way 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 large amount of data by spreading the data and the processing among a cluster. This pattern is well known and used by large companies like Amazon and Facebook. Spark is a technology from 2014 built upon Hadoop and aimed at speeding up the data processing. It ran the whole execution in RAM in realtime, unlike Hadoop does. It only uses persistent storage when the RAM is not sufficient.

4. Results

In this section we discuss the behavior and speedup of the three parallelism strategies. Then we measure the gain on a distributed implementation of the DL task.

4.1. Parallel Deep Learning

Parallel DL is how performs all the parallelism strategies on single node.

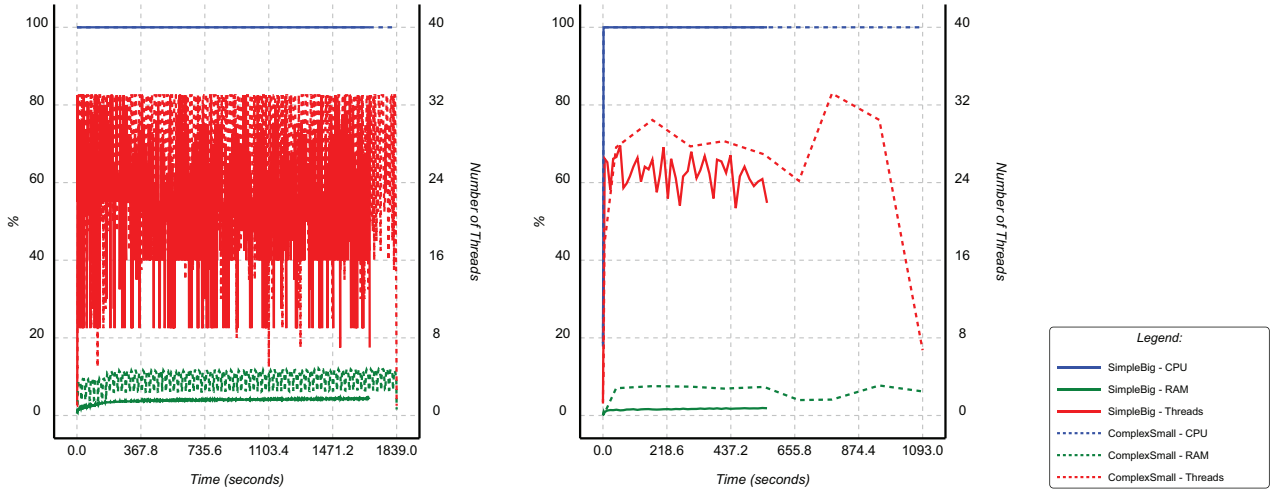
4.1.1. Data Parallelism

Table 1 reports the speedup of applying single or multi process data parallelism on the CPU, and single process data parallelism per GPU. A speedup occurs in the two use cases with a bigger acceleration with the multi process version on the CPU. The acceleration is stronger in the case of SimpleBig. This suggests that the more data there will be, the more the parallelization on the CPU accelerates the processing. On the GPU the speedup is not as great as in the case of the CPU. The worst case is the SimpleBig usecase meaning that the more data, the more transfers to the GPU. This decreases the speedup.

Table 1. 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 system process priorities – the scheduling of processes handled by operating systems – and thanks to the use of Python, using threads only can reduce the total amount of time allocated on data loading and learning task. Moreover, using Python threads allows pure concurrency tasks to be hindered by the 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 it it the case for the considered deep learning 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:



(a) Single-Process.

(b) Multi-Process.

Figure 2. Resources utilization rate on CPU data parallelism for single process and multi process respectively in Figures 2(a) and 2(b). 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 blue. The SimpleBig use case is in solid lines and the ComplexSmall use case is in dashed lines.

Higher priority because the application priority depends of the number of all process on the system if all process 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.

multi-threaded data preparation can be made for each process. A process will therefore repeat these steps in each epoch:

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

Each thread – only used at step 1 – is interpreted by Python only, no native operation are therefore 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 single process and multi-process. However, the threads 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 suggest that the framework use one thread per image. The behavior is smoother in multi process. Also the percentage of RAM utilization is smoother in multi process and requires less memory than in the single process. The ComplexSmall use case requires more RAM than the SimpleBig use case wich is explained because more parameters have to be maintained in memory.

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 the SimpleBig in multi process data parallelism. This is explained because 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 they are less images per time unit loaded in the RAM.

4.1.2. Model Parallelism

Best parameters of model parallelism on GPU with pipeline is faster than in the previous setup as shown in Table 2. Clearly model parallelism outperforms data parallelism. Moreover the hybrid approach which consists of applying

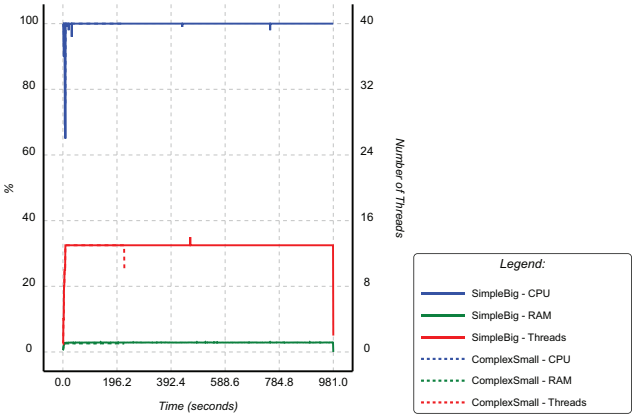


Figure 3. Resources utilization rate on GPU data parallelism. The x-axis is the time (seconds), the left y-axis is the percentage of the CPU (resp. RAM) utilization in blue (resp. green) and the right y-axis is the number of threads in blue. The SimpleBig use case is in solid lines and the ComplexSmall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

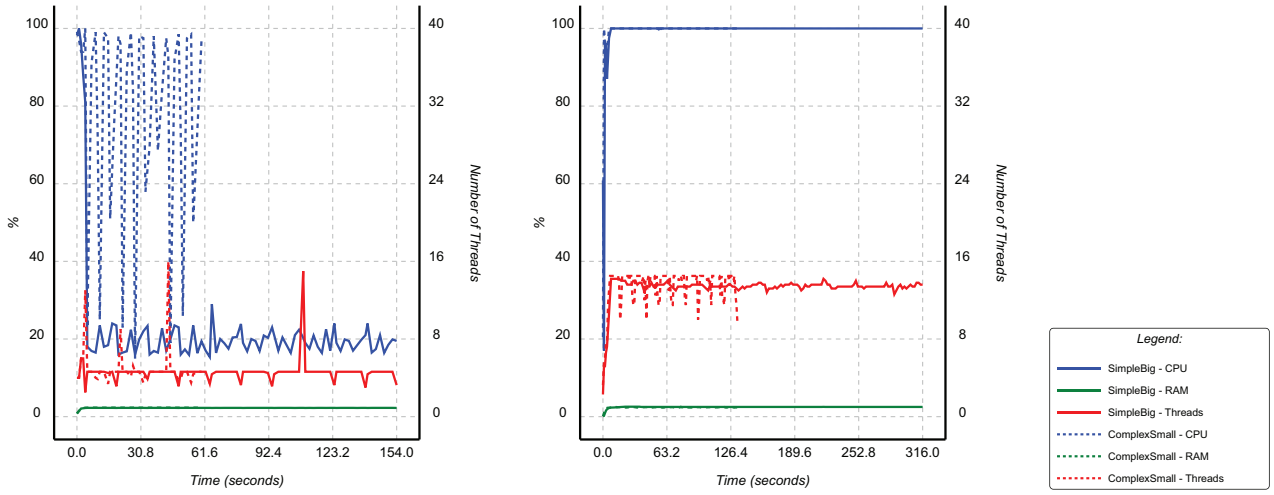
Table 2. Speedup of Model and Hybrid Parallelism

Parallelism	ComplexSmall	SimpleBig
Model	6.74	8.01
Data and Model	2.84	3.85

both parallels ism is faster than data parallelism but slower than model parallelism.

Figure 4(a) 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 4. The utilization of the CPU highly varies in the case of the ComplexSmall use case but stays around of 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 CPU to GPU and from GPU to CPU during the synchronization. Indeed, the model complexity is bigger in the ComplexSmall use case.

The hybrid approach shown in Figure 4(b). The behaviors change to be more similar to the GPU data parallelism. In the two use cases, the CPU becomes used at nearly 100% but the DL tasks 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 the batch size to be more reduced than in the previous setup, so that the GPU can handle it. Less data (i.e. smaller batch) being processed per time unit, more processes would be useful in order to counter this effect. Nevertheless, more process induce more



(a) Model. (b) Data and Model.

Figure 4. Resources utilization rate on GPU model parallelism and hybrid data-model parallelism respectively in Figures 4(a) and 4(b). 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 blue. The SimpleBig use case is in solid lines and the ComplexSmall use case is in dashed lines.

Table 3. 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	-

replication and, therefore, more memory requirements. This is why batch size is reduced, allowing the model replication and less processes to be handled, compared to the the previous setup.

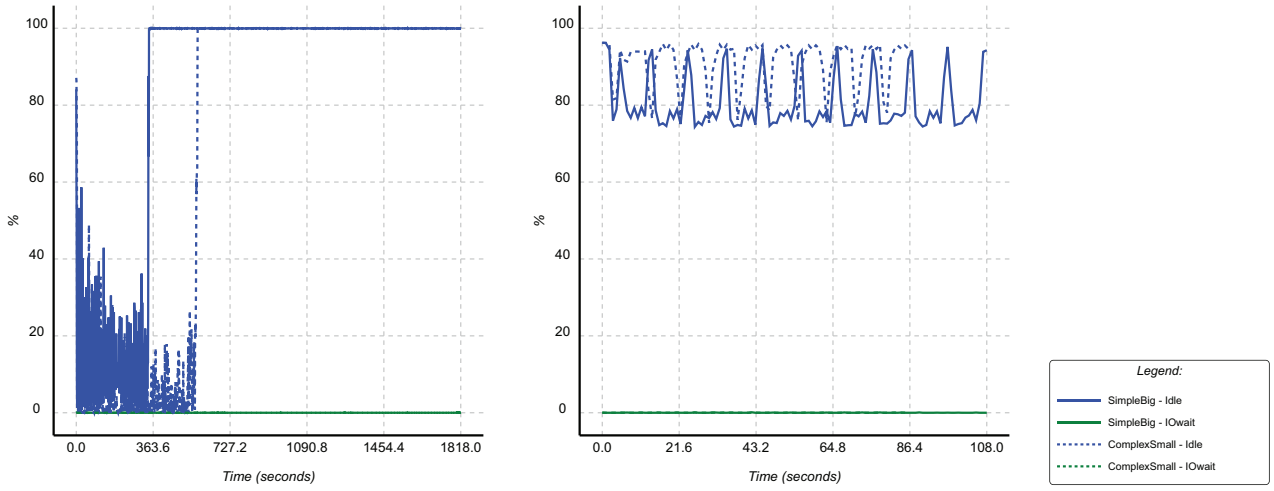
4.2. Distributed Deep Learning

Table 3 shows the speedup results of disctributed computing technologies. Spark is only executed on the CPU because Spark does not nativley 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 is, the harder it is for Spark to perform the processing quickly. The MPI version executed on the CPU provides the best speedup on the two use cases. 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 best 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 enough to balanced the cost of the network synchronization. The network communication were analyzed during the DL tasks revealing that no bottleneck occurred. A peak usage of 35KiB were observed on a dedicated 10GB Ethernet connection.

The best local parallelism strategy on the CPU were 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 after a period very inactive as shown in Figure 5. At the beginning, the CPU loads the input data from the storage in the RAM. At that point the CPU is active. After the whole 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 a latency. This latency causes the CPU to wait a response and to be mostly inactive. Nevertheless the CPU spent a low percentage of time in the IOwait state which means that it efficiently loads the input data in 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.

5. Conclusion

In this paper we proposed a novel way to speedup the DDL. We speedup the distribution of a DL task by local speedup of all the nodes. To this end we have considered the data parallelism, the model parallelism and a hybrid approach. Parallelism were applied on the CPU and the GPU with two use case to highlight to effect of a complex model and the effect of the amount of data. With a small dataset we shown that DDL can be slower than parallelism. Nevertheless, other setups results in a speedup up to 26.63 on the CPU



(a) CPU. (b) GPU.

Figure 5. Resources utilization rate on the MPI distributed version running on the CPU and the GPU respectively in Figures 5(a) and 5(b). The x-axis is the time (seconds) and the y-axis is the percentage of time that the CPU is inactive in blue, the CPU were idle during which the system had an outstanding disk I/O request in green. The SimpleBig use case is in solid lines and the ComplexSmall use case is in dashed lines.

and 11.79 on the GPU which is better than the state of the art. In future work, we are interested in exploring how the source of data affects the results and to deploy our solution on cloud computing.

References

Ben-Nun, T. and Hoefler, T. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

Cong, G. and Bhardwaj, O. A hierarchical, bulk-synchronous stochastic gradient descent algorithm for deep-learning applications on gpu clusters. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pp. 818–821. IEEE, 2017.

Das, D., Avancha, S., Mudigere, D., Vaidynathan, K., Sridharan, S., Kalamkar, D., Kaul, B., and Dubey, P. Distributed deep learning using synchronous stochastic gradient descent. *arXiv preprint arXiv:1602.06709*, 2016.

Hashemi, S. H., Jyothi, S. A., and Campbell, R. H. Tictac: Accelerating distributed deep learning with communication scheduling. *arXiv preprint arXiv:1803.03288*, 2018.

Hegde, V. and Usmani, S. Parallel and distributed deep learning. In *Tech. report, Stanford University*, pp. 1–8, 2016.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., and Chen, Z. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

Hussain, L., Banarjee, S., Kumar, S., Chaubey, A., and Reza, M. Forecasting time series stock data using deep learning technique in a distributed computing environment. In *2018 International Conference on Computing, Power and Communication Technologies (GUCON)*, pp. 489–493. IEEE, 2018.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Kuang, D., Chen, M., Xiao, D., and Wu, W. Entropy-based gradient compression for distributed deep learning. In *2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pp. 231–238. IEEE, 2019.

Lerat, J.-S., Mahmoudi, S. A., and Mahmoudi, S. Single node deep learning frameworks: Comparative study and cpu/gpu performance analysis. *Concurrency and Computation: Practice and Experience*, pp. e6730, 2021.

Lim, E.-J., Ahn, S.-Y., and Choi, W. Accelerating training of dnn in distributed machine learning system with shared memory. In *2017 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1209–1212. IEEE, 2017.

Lim, E.-J., Ahn, S.-Y., Park, Y.-M., and Choi, W. Distributed deep learning framework based on shared memory for fast deep neural network training. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pp. 1239–1242. IEEE, 2018.

Moritz, P., Nishihara, R., Stoica, I., and Jordan, M. I. Sparknet: Training deep networks in spark. *arXiv preprint arXiv:1511.06051*, 2015.

Sattler, F., Wiedemann, S., Müller, K.-R., and Samek, W. Sparse binary compression: Towards distributed deep learning with minimal communication. In *2019 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2019.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.1556.

Tsai, C.-Y., Lin, C.-C., Liu, P., and Wu, J.-J. Communication scheduling optimization for distributed deep learning systems. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 739–746. IEEE, 2018.

Wen, W., Xu, C., Yan, F., Wu, C., Wang, Y., Chen, Y., and Li, H. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pp. 1509–1519, 2017.