ChainerMN: Scalable Distributed Deep Learning Framework [*]

Takuya Akiba
Preferred Networks, Inc.
akiba@preferred.jp

Keisuke Fukuda
Preferred Networks, Inc.
kfukuda@preferred.jp

Shuji Suzuki
Preferred Networks, Inc.
ssuzuki@preferred.jp

Abstract

One of the keys for deep learning to have made a breakthrough in various fields was to utilize high computing powers centering around GPUs. Enabling the use of further computing abilities by distributed processing is essential not only to make the deep learning bigger and faster but also to tackle unsolved challenges. We present the design, implementation, and evaluation of ChainerMN, the distributed deep learning framework we have developed. We demonstrate that ChainerMN can scale the learning process of the ResNet-50 model to the ImageNet dataset up to 128 GPUs with the parallel efficiency of 90%.

1 Introduction

It has turned out that deep learning achieves far better-predicting performance than existing methods in image recognition, natural language processing, speech recognition and many other fields where machine learning is being applied. The basic technology of neural networks used in deep learning has a long history dating back to the 1950's. As we entered the 2010's, the neural network technology with its long history has made the breakthrough as "deep learning" as described above because it is thought to have successfully combined all the advances of algorithms, large-scale data, and high computing powers. Even today, it would be difficult to achieve an outstanding predicting performance by deep learning if one of the three lacks. In this article, we focus on one of the three pillars supporting deep learning: computing performance.

It has become a standard approach to use highly efficient GPUs for training in many deep learning tasks. Nevertheless, the training process is still time-consuming even with the latest GPUs because models have also grown massive and complex. For example, training Resnet-50 [9] for the ImageNet dataset [7] typically takes as long as one week with a single GPU. Taking a long time on training means you have a limited number of times to do trial and error for models and parameters needed to achieve high accuracy, making it difficult to produce a good predicting performance. It also means there is a limit to the usable data size. Thus, using multiple GPUs in parallel is crucial in accelerating calculation.

We introduce ChainerMN, an add-on package to Chainer [12], a programming framework for deep learning applications written in Python, to provide a distributed learning capability. In the course of developing ChainerMN, we took the following features into consideration:

- **Flexibility:** Chainer is a flexible framework based on its Define-by-Run approach and ChainerMN is designed not to ruin the flexibility aspect. This allows for easy distributed learning even in complex use cases such as dynamic neural networks, generative adversarial networks, and reinforced deep learning.

[*] This paper is based on our invited article in TSUBAME e-Science Journal (ESJ)[3]. ESJ is a non-academic and non-peer-reviewed newsletter from Global Scientific Information Center, Tokyo Institute of Technology.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

- **High performance:** We selected technologies assuming practical workloads in deep learning from the very beginning of designing ChainerMN as well as exercised ingenuity with respect to implementation so that hardware performance is fully utilized.

The rest of the paper is organized as follows. First, we explain the basic elements of distributed deep learning, followed by the design and implementation of ChainerMN. Finally, we will present the results of our evaluation experiment and related work.

2 Preliminaries

2.1 Basics of Deep Learning

We can express the prediction by neural networks against input data x as $f(x;\theta)$ where θ is a parameter for neural networks. Learning in neural networks using backpropagation and stochastic gradient descent or its variations is an iterative algorithm. Each iteration is composed of the following three steps: forward computation, backward computation, and optimization.

In the forward-computation step, first, the prediction $f(x;\theta)$ is calculated against an input data point x. Then, the loss is calculated to represent the difference from the correct output for. Here, the cross entropy and other indicators may be used.

In the backward-computation step, $g = \frac{\delta E}{\delta \theta}$, the gradient of the parameter θ in the direction of decreasing the loss E, is calculated. Gradients for all parameters are calculated using the chain rule while going backward from the output layer to the input layer.

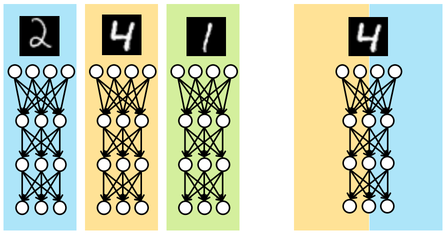
In the optimization step, the parameter θ is updated using the gradient g . The simplest rule is to update θ to $\theta - \mu g$ where μ is a parameter called a learning rate.

In practice, instead of using a single training example in an iteration, the forward and backward calculations are performed simultaneously against multiple training examples and optimization is executed using the average of gradients against all the examples. The input examples used in an iteration is called a minibatch and its size is calleda a batch size. Batch size typically ranges from several tens to several hundreds.

Please note that the above description is based on a standard supervised learning. Nonetheless, in case that neural networks are applied to other algorithms such as unsupervised learning and semi-supervised learning, the parallelizing method we will explain below is applicable and ChainerMN is also usable.

2.2 Data Parallelism and Model Parallelism

There are two major approaches to parallelize training by distributed processing: data parallelism and model parallelism. In data parallelism, each worker has a model replica and calculate gradients of different minibatches. Workers use these gradients to update the model collaboratively. In model parallelism, each worker has a portion of the model and works in cooperation with others to do the calculation for one minibatch. Figure 1 shows the difference between the two approaches.



(a) Data parallelism. (b) Model parallelism.

Figure 1: Data parallelism and model parallelism.

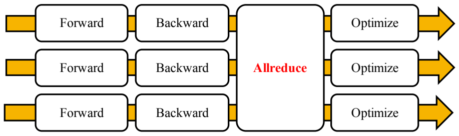


Figure 2: The four steps that constitute an iteration of synchronous data parallelism.

Model parallelism was actively used in the days when GPU memory was small. At present, model parallelism is rarely used in its basic form as data parallelism is being used. In the meantime, some issues with data parallelism have surfaced while research on a new form of model parallelism is underway. Model parallelism and data parallelism can be used at the same time as well.

2.3 Synchronous vs. Asynchronous

In this subsection, we will focus on data parallelism which is commonly used now. Data parallelism has roughly two types in the way workers share gradients; synchronous type and asynchronous type. Each iteration in synchronous type, data parallel deep learning is composed of the following four steps: forward computation, backward computation, `Allreduce` communication, and optimization. Figure 2 illustrates the four steps.

This has an additional step `Allreduce` to the regular iteration described earlier. In this step, workers communicate with each other to find the average of gradients calculated by individual workers and distribute the average. All workers update the model using the gradient they have obtained through the communication. Let b be batch size at each worker and n be the number of workers. The gradient obtained through data parallel computation and workers' communication is equivalent to the gradient in batch size bn in non-parallel computation. This means gradients are calculated using more training data in one iteration, improving the gradient quality and accelerating the learning process.

Asynchronous type, on the other hand, uses special workers called a parameter server. The parameter server controls model parameters. Normal workers send gradients to the parameter server once the gradients are obtained by forward and backward calculations. The parameter server receives and uses the gradients to update the model. Workers receive new model parameters and calculate gradients again.

3 Design and Implementation

3.1 Parallelization Approaches

We discuss the design decision of ChainerMN in this section. As we discussed in section 2, there are two major parallelization approaches and two synchronization approaches. We adopt a synchronous and data parallelism for ChainerMN.

We use the data parallelism because existing deep learning applications would easily be extensible and faster training process through data parallelism was highly expected. Data parallelization is tantamount to increasing a minibatch size in a typical deep learning application and has its advantage of being applicable without having to make significant changes in algorithms and codes of existing programs.

Whether the synchronous or asynchronous type is desirable is also a nontrivial question since different kinds of strategies have been taken in each implementation and results would vary depending on tasks or settings. The paper [11] shows experimental results that the asynchronous type is less stable regarding convergence whereas it is faster to converge in the synchronization. Also, we can benefit from the optimized and proven group communication mechanism of MPI, the de-facto standard communication library interface, while in the asynchronous model the implementation scheme uses a parameter server in general.

3.2 Chainer

Chainer is a framework with its Define-by-Run feature. Define-by-Run is a model that takes advantage of the flexibility of script languages where learning models and computational flows are defined at runtime. On the other hand, in a Define-and-Run approach, a structure of networks is predefined, after which data is input and calculation are done. While potentially easier to optimize performance, this approach is said to lack flexibility.

Chainer provides programming models that enable to define complex neural networks flexibly and to make modifications during runtime thanks to its Define-by-Run approach. This lets researchers and engineers work on new models or complex models through trial and error with ease and therefore is suitable for research and development of machine learning. Upon development, we carefully designed the ChainerMN API with the objective of making it easily portable from existing Chainer programs without putting limitations on the flexibility of Chainer.

Listing 1: Example of ChainerMN

```
model = L.Classifier(MLP(args.unit, 10))

# Create a communicator
comm = chainermn.create_communicator()

# Distribute a dataset
train = chainermn.scatter_dataset(train, comm, shuffle=True)

# Create and use multi_node_optimizer
optimizer = chainermn.create_multi_node_optimizer(
        chainer.optimizers.Adam(), comm)
        optimizer.setup(model)

# Use Chainer's Trainer class to simplify
# a forward−backward−optimization loop
train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
updater = training.StandardUpdater(train_iter, optimizer, device=device)
trainer = training.Trainer(updater, (args.epoch, 'epoch'), out=args.out)
```

3.3 API Design

We describe the API design goal of ChainerMN, followed by a description of minimal steps to extend an existing deep learning program written in Chainer to support distributed execution using ChainerMN.

The design goal of ChainerMN is to achieve high performance without sacrificing the flexibility of Chainer. In the Define-by-Run approach, the model structure and other parameters may differ between iterations. Thus, ChainerMN assumes that the model structures are identical between workers merely in a single iteration. Communication for gradient exchange happens just before the optimization step of Chainer, and the step is transparent to other Chainer component. This design puts a minimal restriction on existing Chainer programs. Programmers can write any code before or after the optimization step, including updating the model structure dynamically, as long as it is consistent between workers.

Listing 1 shows a simplified ChainerMN program of a model to solve MNIST classification problem [10]. For a complete program code, refer to ChainerMN's code repository [1]. There are three major steps: *(1)* add a communicator component, *(2)* create and use `mutli_node_optimizer`, and *(3)* add code to distribute a dataset.

A process of modifying an application starts by adding a communication component called `Communicator` to existing Chainer programs. A communicator is a central component of ChainerMN, and it is designed after MPI's communicator concept and controls all inter-process communication in ChainerMN program.

`mutli_node_optimizer` is the most important component in ChainerMN. It wraps Chainer's normal optimizer and exchanges the gradient across processes using `Allreduce` operation before optimizing the model. `multi_node_optimizer` behaves identically the same way as the original optimizer except for the communication, so the extension is seamlessly integrated into Chainer's existing `Trainer` ecosystem.

On top of this, basic porting can be done just by adding the scattering step which distributes data for data parallel computations. One needs to split the dataset into chunks and distribute them over the processes. This operation is also known as `Scatter` in MPI. Other parts, i.e.`Iterator`, `Updater`, and `Evaluator` do not need to be changed in basic use cases. Because of this API design, it allows various Chainer programs to be ported with minimal modifications while making the most of the advantage given by Define-by-Run.

3.4 Implementation and Performance Optimization

The communication pattern of synchronous and data parallel deep learning applications is relatively simple from the point of view of HPC applications. Roughly speaking, the only major communication is `Allreduce`, a process to exchange gradients which are training and evaluation results. Auxiliary parts include Scatter, which arranges necessary data over distributed processes before starting training.

As mentioned above, one of the design goals of ChainerMN is to achieve high performance by leveraging existing and proven HPC technologies. Allreduce is a step that especially requires speed because it runs in every training iteration and needs to process a large amount of data. We minimized the communication time by using NCCL [2] library developed by NVIDIA. NCCL is a highly-optimized communication library which provides a faster Allreduce operation between NVIDIA GPUs within and across nodes.

4 Evaluation

4.1 Experimental Environment and Settings

We conducted our experiments on our in-house cluster. It consists of 32 computing nodes. Each node is equipped with two Intel Xeon CPUs (E5-2623 v3, 3.00 GHz, four cores for each), 128 GB of main memory, and four GeForce GTX TITAN X GPUs. Thus, we used 128 GPUs in total. The nodes are interconnected by Mellanox Infiniband FDR 4X. We used CUDA version 8, Python version 3.5, Mvapich2 2.2 and Chainer version 1.2 running on Ubuntu 14.04 LTS.

To demonstrate the performance and scalability of ChainerMN, we used ResNet-50 [9] model and ImageNet [7] dataset. Since the dataset is large and the majority part of access is read, we copied all the dataset to all computing nodes' local SSD in advance.

For simplicity and fairness or the experiment, the model parameters and the rest of configuration details were based on the original ResNet-50 paper [9] except a few differences. We did not employ color augmentation and scale augmentation for training, and 10-crop prediction and fully-convolutional prediction for validation. We used 32 as the batch size per GPU, which means 4096 for 128 GPUs.

4.2 Resulting Model Accuracy

One of the factors making distributed deep learning difficult is that improving throughput does not necessarily mean better learning efficiency. We note that the batch size 4096 is a healthy setting where the learning efficiency and the resulting model accuracy are maintained, as shown by Goyal et al. [8]

The resulting model achieved over 71% top-1 accuracy on ImageNet validation, which is a fair result considering the minor difference in the experimental configuration. It has been reported that one can achieve higher validation accuracy by employing multiple techniques such as learning rate scaling and gradual warmup for such large minibatch sizes [8]. These techniques are, however, beyond the scope of this paper because they are algorithm-level improvements and can be realized on any well-implemented distributed deep learning framework. Accuracy and scalability results of extremely large-scale experiments using ChainerMN are reported by the authors [5] along with more advanced techniques.

4.3 Scalability Result

Figure 3 shows the scalability of ChainerMN up to 128 GPUs. In this figure, ChainerMN scales well up to 128 GPUs. Table 1 shows the relative runtimes over one-GPU execution. In this table, ChainerMN on 128 GPUs achieved 79 % and 90 % parallelization efficiency of the one-GPU and one-node (four GPUs) executions, respectively. It means that the parallelization efficiency of ChainerMN on 128 GPUs was as high as the state-of-the-art [8].

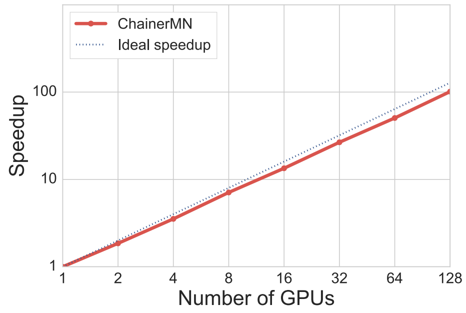


Figure 3: Scalability of ChainerMN

Table 1: Relative speed-up and parallelization efficiency

#GPUs	Speed-up	Par. Eff.
1	1.00	100.00%
2	1.85	92.66%
4	3.53	88.34%
8	7.09	88.67%
16	13.42	83.88%
32	26.63	83.22%
64	50.52	78.94%
128	101.32	79.16%

5 Related Work

Chen et al. [6] showed the result of large-scale experiments using MxNet. They used up to 10 instances of Amazon EC2, although scalability or parallel efficiency was not shown. Abadi et al. [4] reported the result of a large-scale experiment on Google's internal cluster using TensorFlow. The cluster is equipped with a shared network and K80 GPUs. The result was shown up to 200 workers with diminishing returns for both synchronous and asynchronous coordination models. The largest published result as of writing is by Goyal et al. [8] They demonstrated the scalability of Caffe2, which is written in C++, and the parallel efficiency was near-90% using 256 GPUs. The GPUs were connected with NVIDIA NVLink and 50Gbit Ethernet network.

6 Conclusions

We have described the design and implementation of ChainerMN and demonstrated its scalability. Chainer and ChainerMN are designed to have both high flexibility and scalability with its primary object of accelerating research and development in deep learning. We will continue making improvements by tackling challenges such as model parallel, overlapping communication and computation, asynchronous computation among workers, optimized communication by compressed gradients, and fault tolerance.

Acknowledgements

The authors thank K. Ueno, T. Mitsuishi, and N. Yoshifuji for help on the development of ChainerMN. We also thank T. Sudo, Y. Doi, G. Watanabe, R. Okuta, and M. Sakata for help for experiments. We are grateful to T. Miyato and S. Tokui for fruitful discussions as well.

References

[1] ChainerMN. `https://github.com/chainer/chainermn`, 2017.

[2] NVIDIA Collective Communications Library (NCCL). `https://developer.nvidia.com/nccl`, 2017.

[3] TSUBAME e-Science Journal. `http://www.gsic.titech.ac.jp/TSUBAME_ESJ`, 11 2017.

[4] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.

[5] T. Akiba, S. Suzuki, and K. Fukuda. Extremely Large Minibatch SGD: Training ResNet-50 on ImageNet in 15 Minutes (to appear). *NIPS 2017 Workshop: Deep Learning At Supercomputer Scale*, 12 2017.

[6] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. 12 2015.

[7] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[8] P. Goyal, P. Dollár, R. B. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778, 2016.

[10] Y. Lecun and C. Cortes. The MNIST database of handwritten digits.

[11] X. Pan, J. Chen, R. Monga, S. Bengio, and R. Jozefowicz. Revisiting distributed synchronous sgd. *ICLR Workshop Track, 2016*, 02 2017.

[12] S. Tokui, K. Oono, S. Hido, and J. Clayton. Chainer: a next-generation open source framework for deep learning. In *LearningSys*, 2015.