Distributed and Parallel Deep Learning

A case study of image classification on MNIST database

Shiyuan Wang (shiyuan8@illinois.edu)

Xiaokai Yang (xiaokai2@illinois.edu)

Introduction

Deep learning for large amounts of data is computationally intensive and time-consuming. Distributed and parallel algorithms can reduce the training times. To achieve the distribution of the training step, there are two types of parallelism: model parallelism and data parallelism, and the latter is mainly used for speeding the computation of convolutional neural networks with large datasets.

In this project, we focus on data parallelism, where the training data is divided into multiple subsets, and each one of them is run on the same replicated model in a different process. Then we update the model for the next iteration only after all nodes have successfully sent their gradients, after which, the updated model is sent to all processes. This procedure is called synchronous training (compared with asynchronous training: no device waits for updates to the model from any other device).

Comparable counterparts

1. Sequential version (baseline); 2. Parallelized version on multiple processes and CPUs using MPI.

Performance metrics

1. Run time (detailed explanation in section Parallel Deep learning); 2. Predictive accuracy.

Resources

Main packages: mpi4py (link); TensorFlow (link).

Dataset: MNIST dataset. It is a large database of handwritten digits that is commonly used for training various image processing systems. There are 60,000 training images and 10,000 testing images, all in 10 classes. The black and white images are normalized to fit into a 28x28 pixel.

CPU: HPE Apollo 6500 nodes with dual 6248 Cascade Lake CPUs.

Sequential Deep learning

For the architecture of the neural networks, We build a model by ourselves instead of using ResNet50 as stated in our proposal, since the latter one requires images of size 224x224, while the MNIST

dataset is composed of 28x28 images. Even simple fully connected networks can have a good performance on the MNIST dataset. However, the task of this project is not to test the efficiency of the classifier model, but to evaluate the performance of the parallel strategy on computation. In this case, the computation would be model update, and thus we created a relatively complicated model (a model with 690,590 trainable parameters) to increase computational complexity manually. The model summary is shown below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320

max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
)

conv2d_1 (Conv2D)            (None, 11, 11, 128)       36992

max_pooling2d_1 (MaxPooling  (None, 5, 5, 128)        0
2D)

conv2d_2 (Conv2D)            (None, 3, 3, 256)         295168

max_pooling2d_2 (MaxPooling  (None, 1, 1, 256)        0
2D)

flatten (Flatten)            (None, 256)               0

dropout (Dropout)            (None, 256)               0

dense (Dense)                (None, 1000)              257000

dense_1 (Dense)              (None, 100)               100100

dense_2 (Dense)              (None, 10)                1010

=================================================================
Total params: 690,590
Trainable params: 690,590
Non-trainable params: 0
_________________________________________________________________
```

Figure 1 Model summary

Before introducing the detailed implementation, we have to know about epoch and batch in deep learning. In an epoch, the whole dataset is passed through the neural network exactly once. If we cannot pass the entire dataset to an algorithm at once, we can divide the dataset into batches, each of which contains a subset of the whole dataset. Then we iterate through those small batches to complete the learning of one epoch. The number of iterations equals dataset size over batch size.

The parameters of the learning algorithm are shown in the table below. Each process will have a mini-batch of size 128, and thus the batch size for one iteration in an epoch would be 128 by the number of processes. Learning rate should also change with batch size accordingly.

Table 1 Model parameters

Batch Size	128 × # of processes

Learning rate	0.001 × # of processes
Epochs	5
Optimizer	Adam
Loss function	Categorical Crossentropy

Parallel Deep learning

In this project, training data is divided into multiple subsets, each of which is run on the same replicated model in a different process. To ensure all those subsets are trained on a consistent model, we need to synchronize the model parameters, i.e. gradients, at the end of each batch.

Ideally, after each process performs gradient descent on its data, we should average these gradients with the MPI call Allreduce, and then we update the model on each node with the averaged gradient. The following graph describes the mechanism.

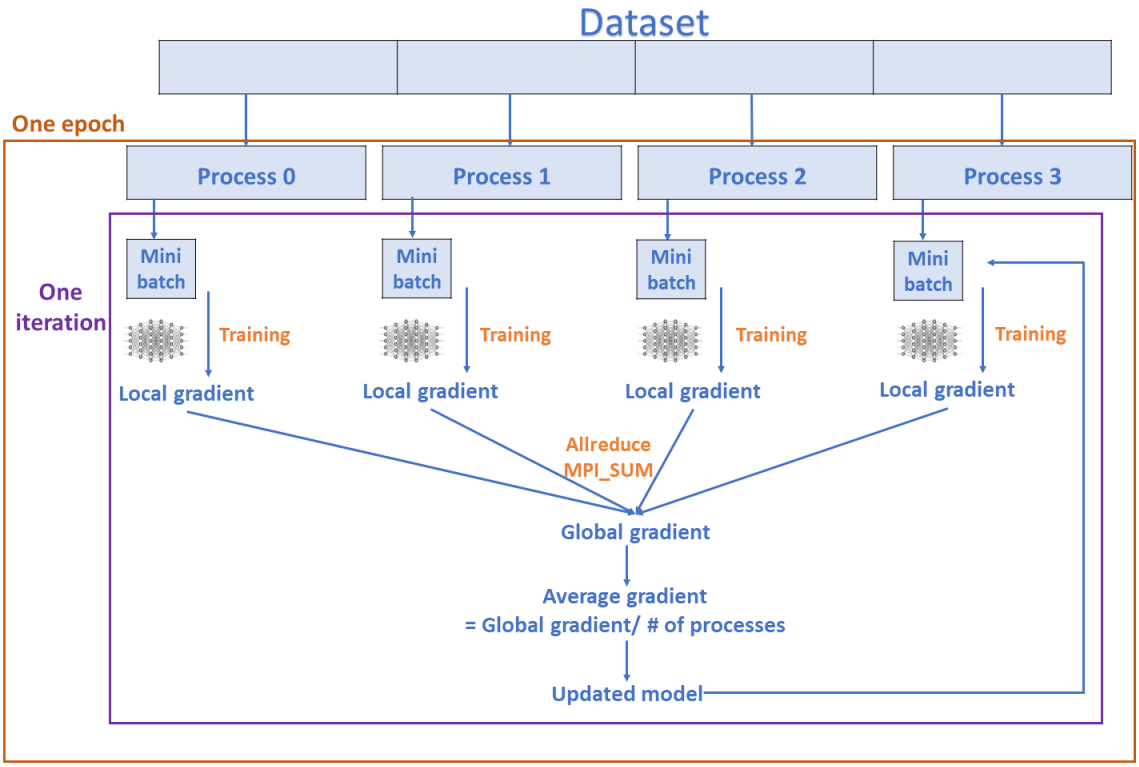


Figure 2 Communication in data parallelism (1)

In mpi4py, before calling Allreduce, we need to define a receive buffer to store the gradient being sent. However, the gradient calculated after each batch is a list of arrays, which is hard to initialize.

Therefore, we first use a Reduce call to calculate the global gradient as the summation of gradients from all the processes to process 0. After averaging the gradient, we broadcast the average gradient from process 0 to each process for the next iteration.

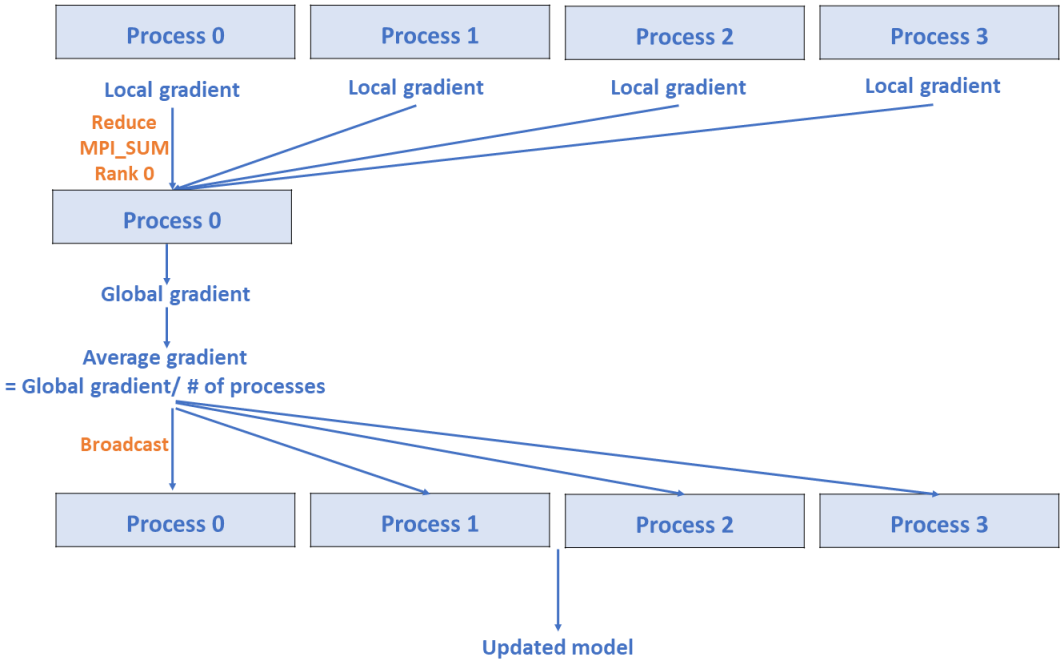


Figure 3 Communication in data parallelism (2)

Performance Comparison

We run our program using a different number of nodes and processes, and figure 4 shows the job submission file for the case "8 processes in 2 nodes". We want to see how the performance varies with the number of processes, and how multiple nodes affect performance. Table 2 shows the summary of our results.

```
#!/bin/bash
#
#SBATCH -t 48:00:00
#SBATCH --nodes=2
#SBATCH --ntasks-per-node=4
#SBATCH --output=2n8p.out
#SBATCH --error=2n8p.err
#SBATCH -p ctessum

mpiexec -n 8 python3 mpi.py
```

Figure 4 Job submission file

Table 2 Result summary

Number of nodes	# of processes per node	Total number of processes	Time to read the file (sec)	Time to update gradient for each batch (sec)	Time for each epoch (sec)	Test loss	Test accuracy (%)	Total runtime (sec)
1	1	1	0.53	0.14	142.04	0.0441	98.80	595.27
1	2	2	0.55	0.15	82.06	0.0559	98.43	416.88
1	4	4	0.54	0.17	34.40	0.0659	98.28	193.12
1	8	8	0.55	0.15	20.52	0.0912	97.20	103.67
2	1	2	0.57	0.15	81.99	0.0521	98.48	415.21
2	2	4	0.55	0.16	41.03	0.0664	98.32	207.78
2	4	8	0.53	0.18	20.52	0.1115	96.75	97.32
4	1	4	0.88	0.13	41.06	0.0591	98.47	215.06
4	2	8	0.89	0.13	20.53	0.1942	94.03	130.8

Let first take a look at the meaning of those performance metrics.

Each process loads the whole MNIST dataset and initializes the model parameters in parallel, note that the latter one actually takes negligible time. “Time to read the file (sec)” refers to the time for all processes to finish loading the dataset.

Each process updates gradients for a fixed batch size (128 per process) during one iteration in parallel, and “Time to update gradient for each batch (sec)” refers to the time for all processes to finish one update on the gradient. We record this time every 10 batches and take the average over all batches in all epochs.

“Time for each epoch (sec)” includes time for processes to compute their own gradient and communicate with each other during one epoch. Note that the communications among processes happen at the end of every batch. Also note that we have 5 epochs, and the results show that each epoch has a similar run time. Therefore, the time we record in this table would be the median among 5 epoch times. For example, for the case “8 processes in 2 nodes”, the runtime for five epochs would be (20.52s, 20.52s, 20.53s, 20.52s, 20.53s), and in 20.52s is recorded in the table.

The models trained under different cases are evaluated by “Test loss” and “Test Accuracy”. “Total

runtime" of the whole program under different cases is also recorded.

Fig 5 shows a simplified structure of the program and corresponding performance metrics. Fig 5 shows the output of the case "8 processes in 2 nodes".

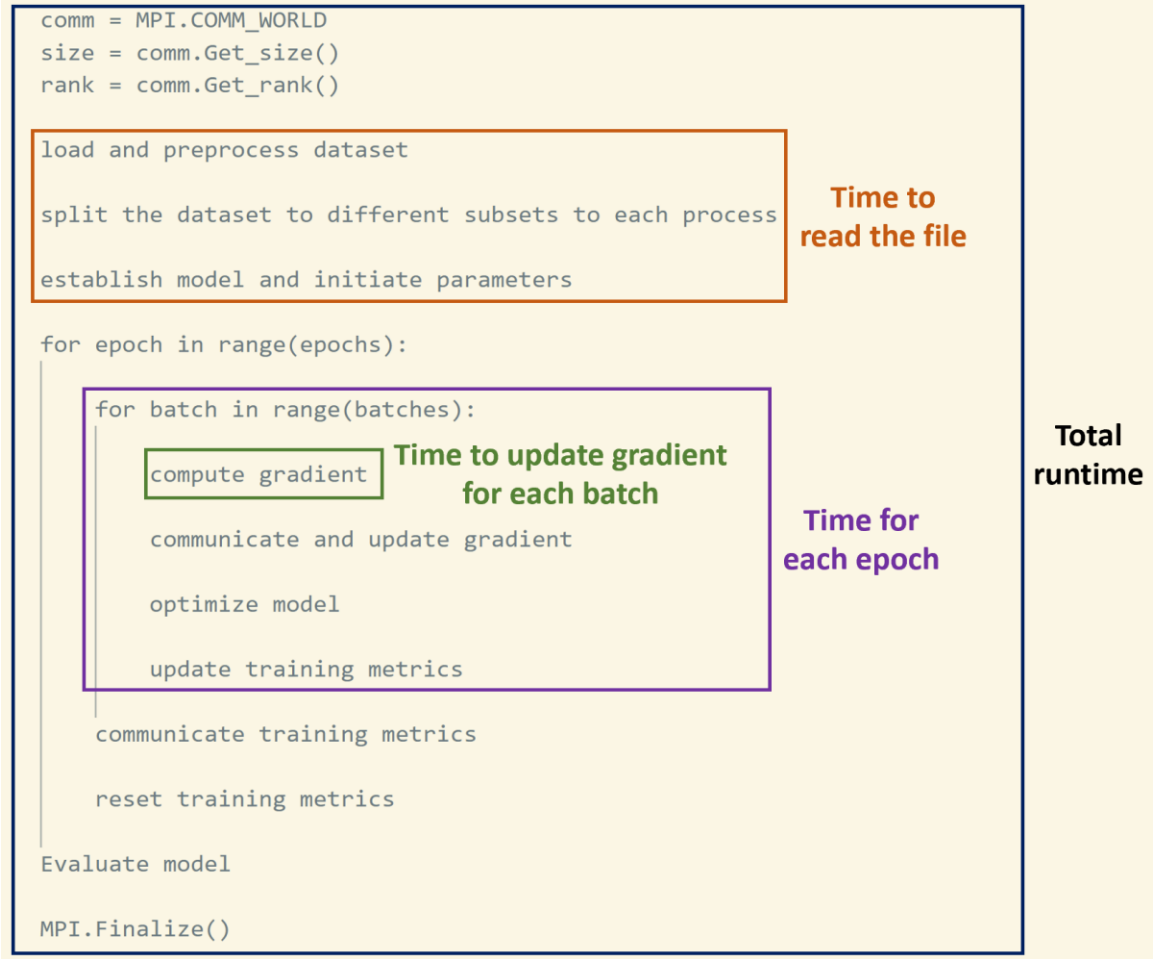


Figure 5 Simplified structure of the program

```
Time taken to read file: 0.53s

Start of epoch 0
At step 10, Update gradient for each batch taken: 0.18s
At step 20, Update gradient for each batch taken: 0.18s
At step 30, Update gradient for each batch taken: 0.18s
At step 40, Update gradient for each batch taken: 0.18s
At step 50, Update gradient for each batch taken: 0.19s
Time taken: 20.52s
Training acc over epoch: 0.1375, Training loss (for one epoch) at step: 2.2564

Start of epoch 1
At step 10, Update gradient for each batch taken: 0.18s
At step 20, Update gradient for each batch taken: 0.20s
At step 30, Update gradient for each batch taken: 0.18s
At step 40, Update gradient for each batch taken: 0.18s
At step 50, Update gradient for each batch taken: 0.18s
Time taken: 20.52s
Training acc over epoch: 0.6449, Training loss (for one epoch) at step: 0.9933

Start of epoch 2
At step 10, Update gradient for each batch taken: 0.18s
At step 20, Update gradient for each batch taken: 0.18s
At step 30, Update gradient for each batch taken: 0.19s
At step 40, Update gradient for each batch taken: 0.17s
At step 50, Update gradient for each batch taken: 0.17s
Time taken: 20.53s
Training acc over epoch: 0.9153, Training loss (for one epoch) at step: 0.2820

Start of epoch 3
At step 10, Update gradient for each batch taken: 0.17s
At step 20, Update gradient for each batch taken: 0.17s
At step 30, Update gradient for each batch taken: 0.17s
At step 40, Update gradient for each batch taken: 0.17s
At step 50, Update gradient for each batch taken: 0.17s
Time taken: 20.52s
Training acc over epoch: 0.9375, Training loss (for one epoch) at step: 0.2092

Start of epoch 4
At step 10, Update gradient for each batch taken: 0.19s
At step 20, Update gradient for each batch taken: 0.18s
At step 30, Update gradient for each batch taken: 0.16s
At step 40, Update gradient for each batch taken: 0.17s
At step 50, Update gradient for each batch taken: 0.16s
Time taken: 20.53s
Training acc over epoch: 0.9487, Training loss (for one epoch) at step: 0.1719
Test loss: 0.11152157932519913
Test accuracy: 0.9674999713897705
Total run time: 97.32s
```

Figure 6 Output of the case "8 processes in 2 nodes"

Weak scaling refers to how the execution time varies with the number of processes for a fixed problem size per process. Ideally, when we keep work per process constant, the total compute time stays fixed. In our project, "Time to read the file" and "Time to update gradient for each batch" are metrics demonstrating the weak scaling of our program. The graph below shows our program's weak scaling on one node.

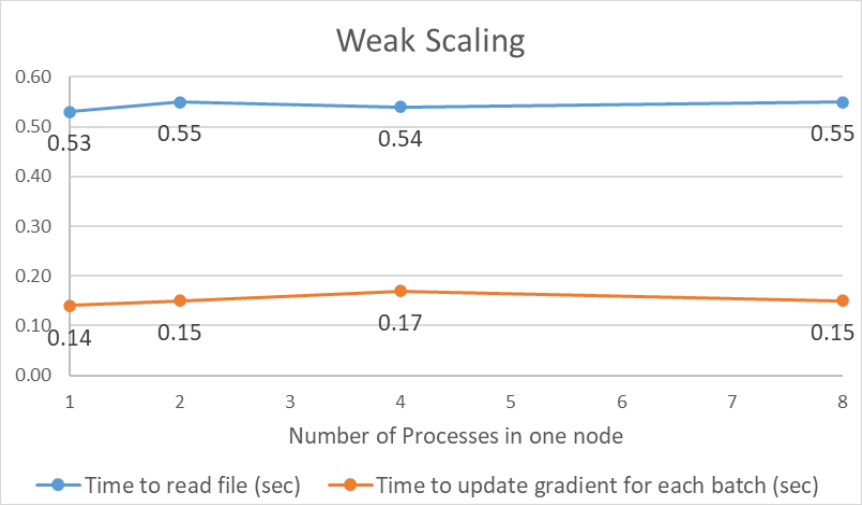


Figure 7 Weak scaling (different # of processes in one node)

Strong scaling refers to how the execution time varies with the number of processes for a fixed, overall problem size. More processes would have a better performance within a certain range, and eventually, more processes do not help. In our project, "Time for each epoch" and "Total runtime" are metrics demonstrating the strong scaling of our program. The graph below shows our program's strong scaling on one node. Overall, dividing the workload into more processes results in a shorter execution time. However, as the number of processes increases, it's clear that the decreasing rate of execution time gradually slows down, and the curve eventually plateaus. This sub-linearity is due to overheads of communication and synchronization.

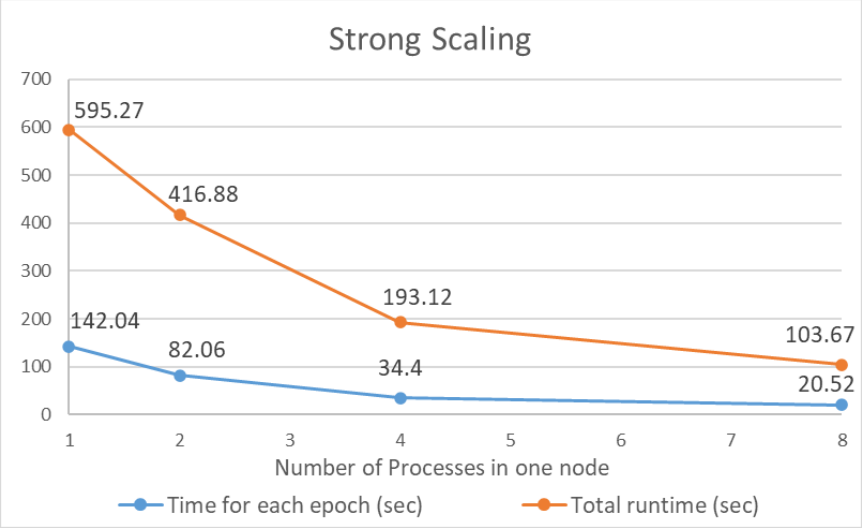


Figure 8 Strong scaling (different # of processes in one node)

Now let's look at how running our program with multiple nodes affects its parallel performance. Here we compare node number 1, 2, 4 with 4 and 8 processes. Each compute node is a multi-processor parallel computer in itself. It requires communication to exchange data between compute nodes. We can see that when we increased the number of nodes, the overall trend of runtime is increasing, with an exception "Total runtime" for 8 processes. More nodes does not necessarily mean better performance, since the communications among nodes are more expensive than within one node, and it's only worthwhile when the computation is heavy, outweighing the cost of communication. In our project, the best performance is achieved when we are using 8 processes across 2 nodes (4 processes each node).

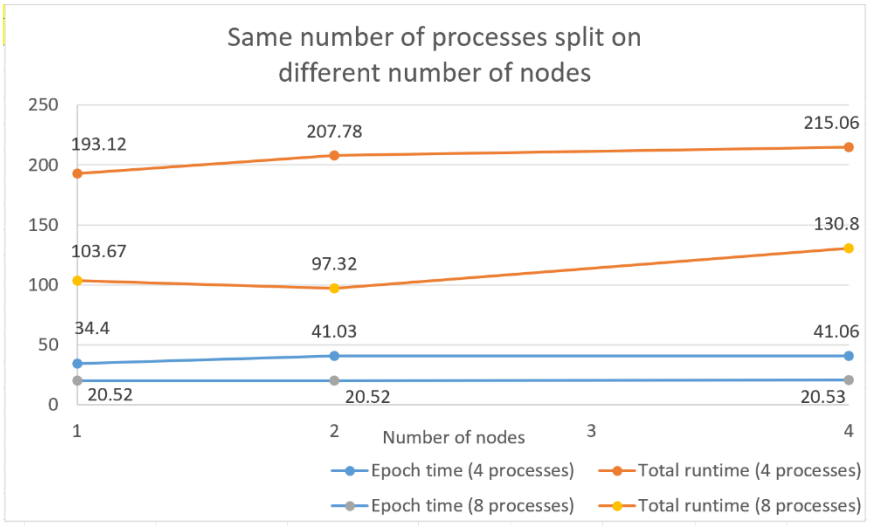
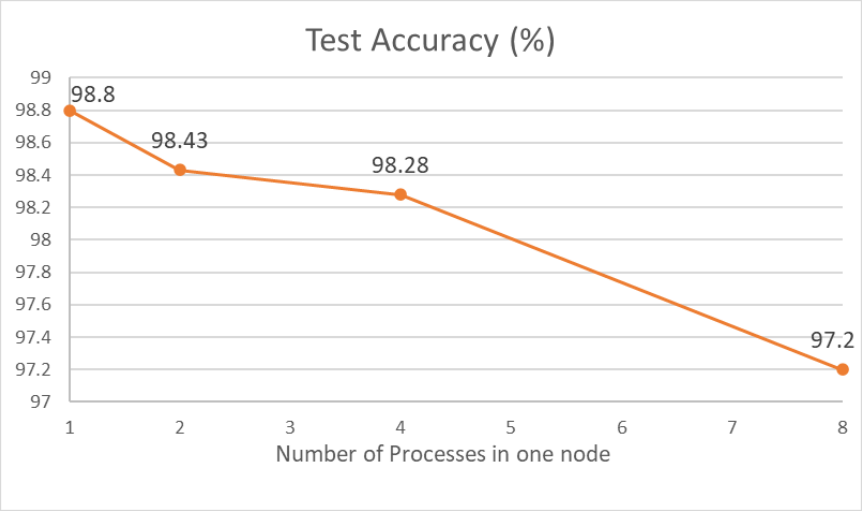


Figure 9 Effect of different number of nodes

The graph below shows the test accuracy with different number of processes on one node. We can see that all cases give us high accuracy, but with a decreasing trend as the number of processes increases. This is because as we increase the number of processes, we increase the batch size accordingly – We let every process receive 128 samples during each iteration, and thus the batch size would be 128 × # of processes. The slight drop on accuracy is because large batch size will lead to poor generalization and slower convergence to the optima in deep learning.



Further Improvement

1. The runtime of this program using one GPU is shown below. The total runtime is only half of our best runtime. We have written code using TensorFlow's distributed API (i.e. MirroredStrategy) to distribute learning among different GPUs. However, this code need to load gcc/8.3.0 while we only have gcc/7.2.0 in campus cluster. However, we should keep in mind that the cost of memory copy between system and GPU memories is expensive, so when the computation is not so heavy, copying inputs and outputs between system memory and GPU memory would be time-consuming. Studies show that distributed GPU computing starts to show its power if the dataset for leaning is huge enough and the neural network is deep enough.

```
Time taken to read file: 0.38s

Start of epoch 0
469it [00:08, 53.48it/s]
Time taken: 8.87s
Training acc over epoch: 0.8999, Training loss (for one epoch) at step: 0.3168

Start of epoch 1
469it [00:10, 46.78it/s]
Time taken: 10.13s
Training acc over epoch: 0.9682, Training loss (for one epoch) at step: 0.1047

Start of epoch 2
469it [00:08, 53.82it/s]
Time taken: 8.82s
Training acc over epoch: 0.9775, Training loss (for one epoch) at step: 0.0742

Start of epoch 3
469it [00:08, 53.19it/s]
Time taken: 10.35s
Training acc over epoch: 0.9811, Training loss (for one epoch) at step: 0.0625

Start of epoch 4
469it [00:08, 54.58it/s]Time taken: 8.70s
Training acc over epoch: 0.9829, Training loss (for one epoch) at step: 0.0575
Total run time: 47.42s
```

2. In this project, each process loads the MNIST dataset independently without using MPI-IO calls, since the dataset is directly loaded from an open-source library, and we find it hard to incorporate it with MPI-IO calls. However, what we should keep in mind is that MPI does provide noncontiguous I/O in files, which is very beneficial to distributed deep learning: Only the part of the file described by the file view is visible to the process, and each process can read partial dataset without accessing the whole file. Those calls can be used when the training set is very large.

3. This project only focus on data parallelism, and we could partition the network layers across multiple processes. This is called model parallelism, which is more complex, and would be useful if the network is deep enough and we have many parameters to train.

GitHub Repository

https://github.com/Saran-Wang/MPI-Deep-Learning

Reference

Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y. and He, K., 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677.

https://mpi4py.readthedocs.io/en/stable

https://keras.io/guides/writing_a_training_loop_from_scratch

https://web.stanford.edu/~rezab/classes/cme323/S16/projects_reports/hedge_usmani.pdf

https://medium.com/mini-distill/effect-of-batch-size-on-training-dynamics-21c14f7a716e

https://towardsdatascience.com/train-a-neural-network-on-multi-gpu-with-tensorflow-42fa5f51b8af