Practical Course on High-Performance Computing

Parallel Deep Learning pipelines using Go and MPI

Silin Zhao

MatrikelNr: 21579349

Supervisor: Dr. Patrick Michaelis

September 26, 2022

Contents

1 Introduction

Massage Passing Interface (MPI) has been widely used in model computer science for parallel computation in distributed-memory system. Just like OpemMP share data in shared-memory system, MPI is implemented for massage transferring with multiple kinds of methods, such as collective and non-collective. Those implementation allow us to build a large cluster, in case of huge computation resource is required, such as for HPC application. Meanwhile Artificial Intelligence (AI) demands also enormous computation resource [1] [2]. In order to accelerate the training process, people invent many new technologies (such as GPU, TPU) in hardware architecture [4], but they are still in the direction of single node task. In this report we will try a new method, using MPI to accelerate AI training process with many nodes. In those nodes we can execute our code, or launch a container [3]. This task parallelism is a kind of distributed learning, by training model with multiple nodes to decrease the training time comsumpation. MPI will be used for weights updating between different networks, and each network will be trained with its corresponding training data. In order to launch multiple MPI processes, we use golang as implemented language for our network, because of its comprehensiveness for C language. Because there is no such AI framework implemented in golang currently, we have to write it by ourselves from scratch, and this becomes an open project in Github for others[1]. This report is designed as following, in Section 2 we are going to review the network implement in golang, and persent its behavior as normal neural network. And then in Section 3 we will see MPI and some directives of it, and then in Section 4 we will see how two approaches are implemented for our project. Here how does MPI update the weights for our model will be detailly discussed for our approaches. In Section 5 we will illustrate the results of our project, the speedup diagrams will show us how our training process can be accelerated by increasing training network. At last in Section 6 we summarize the project and its problem, also state how the project implementation can be optimized further.

2 Network implementation

In this chapter we are going to illustrate the structure of neural network, from data forward training to loss calculating according to their labels, and then to error backward propagation. Comparing to the mature deep learning framework model, our project is just a naive approach. Because our implementation only has fully connected layers with active function and data standardization, no convolutional network layer, no residue network. The whole network architecture is determined by its configuration file, which indicate the number of neurons in each layer, batch size of input data, also the number of training epochs.

2.1 Training data

Network starts with the choice of training data, the first one is iris dataset[2]. Iris dataset has 150 instances, each of those has 4 characters, sepal.length, sepal.width, petal.length,

[1] https://github.com/scofild429/go_mpi_network

[2] https://www.kaggle.com/datasets/saurabh00007/iriscsv

petal.width. Our network wants to predict if each instance in validating dataset belongs to its target (Setosa, Versicolor, Virginica) according to its four characters.

We use one-hot coding for its target, because we already know there are three categories. This training dataset is for local personal computer, as shown in the configuration file (in Appendix C) we has 6 neurons in input layer, 4 neurons in hidden layer and 3 neurons in output layer. For this network we only have about 80 parameters to train.

For large HPC cluster we have intel image classification dataset [3], while in this dataset we have 6 categories. Importantly the last layer of network should have the same number of neurons as its categories, and this is set in its corresponding configuration file. For our case, we have 80 neurons in input layer, 10 neurons in hidden layer and 6 neurons in output layer and about 5 million parameters to train.

2.2 Network architecture

In Figure 1 we illustrate the network updating process, which starts from left side to right side as red arrow indicates. Now let us describe this process more in detail. At first we spilt iris dataset to 80, 10 and 10 percent for training, validating, and testing dataset before the initialization of network. But because intel image classification dataset has already separate training and testing dataset, we only need to spilt train dataset to be 80 to 20 percent for training and validating dataset. For next step we need to initialize the model with randomly weights and biases, after that all training data will reformed as mat.Dense[4] (noted as input data X in Figure 1), which is prepared for matrix operation.

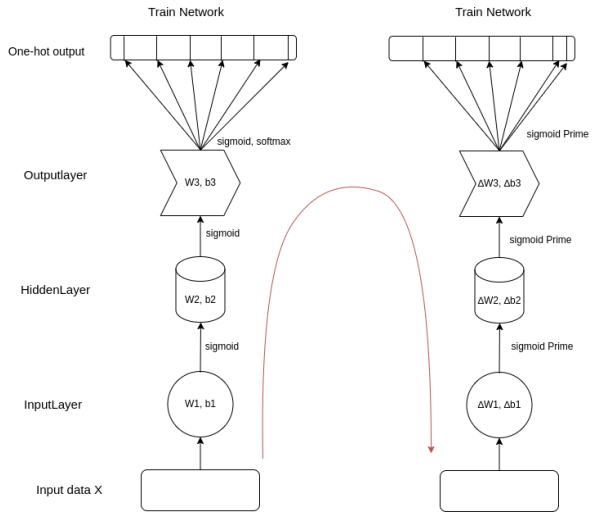

Figure 1: The Illustration of training process from data feeding to loss calculating, and then to weights updating in neural network

We denote training data as X, in each layer we train the data with

$$y = W * X + b \tag{1}$$

[3] https://www.kaggle.com/datasets/puneet6060/intel-image-classification?resource=download
[4] https://github.com/gonum/gonum

, where W and b are the weight matrix and bias matrix of corresponding layer. After that y is applied to sigmoid function,

$$Y = sigmoid(y). \tag{2}$$

Those training process is similar in input layer, hidden layer and output layer, the details can be found in Figure 1 and Appendix A.

Y stands for the output of our network, which has the same size as our one-hot coding target(y), so that we can apply L2 loss function for the output as following,

$$L = \frac{1}{2} \sum_{i=1}^{batchsize} (Y^i - y^i)^2. \tag{3}$$

As to the right side of Figure 1, the weight matrix and bias matrix in input layer, hidden layer and output layer are updated similarly as following,

$$W \leftarrow W + \eta \frac{\partial L}{\partial W}, b \leftarrow W + \eta \frac{\partial L}{\partial b} \tag{4}$$

η is learning rate which is defined in configuration file, and the derivative of weight and bias of each layer can be determined by backward propagation based on chain rule as in Appendix A.

2.3 Network implementation

In our project we only implement the neural network for 3 fully connected layers, however there are many advanced technologies can be applied, such as Reside Network, Convolutional Network. But those implementations in golang is too complicated, so there are out of the script of this report.

Meanwhile we have many options for active function, while after many times testing, we choice sigmoid function for alle layers. But the implementations of tanh function and Rule function can also be found in our source code.

As for the gradient explosion and vanishing we implement normalization [5] and standardization[6] function before active function. This will keep the weights of the network always in a sensible region for active function. But in our code only standardization is used because of the performance reason.

In the evaluation step, we compare the index of biggest output component of each instance, if the index is equivalent to the index of one-hot code where 1 stands, we estimate that it is correctly predicted.

Also the batch size should be carefully discussed. When we train the model, we denote one epoch for the case that when the whole training data has been trained complete once. And in each epoch, we have different methodologies to update the weights. Batch Gradient Descent (BGD) will update the weights only after feeding all training data into network, while Stochastic Gradient Descent (SGD) will update its weights after each instance has been fed.

[5] $x^{'} = \frac{x - min(x)}{max(x) - min(x)}$

[6] $x^{'} = \frac{x - \mu}{\sigma}$

But Mini-batch Gradient Descent (MBGD) has a better performance for fast convergence and robustness to avoid falling into local minimal points. So in our project we use the MBGD, and determine batch size to be 4 as in configuration file.

2.4 Network performance

In order to convince ourselves to belive that, our model in correctly implemented, we offer many implementations with python code. They are from illustrious platform sklearn, pytorch and also from scratch.

In Figure 2 we illustrate the result of our network. This result is from iris dataset, and our model is trained with personal computer. The left side show us how the loss decrease with the training epoch increasing, because we fixed our learning rate, so the decrease becomes much more slowly later, after 100 epochs we have loss about 0.5. At the same time, we have a very great circumstances for accuracy performance, it continually increases up to almost 100 percent.

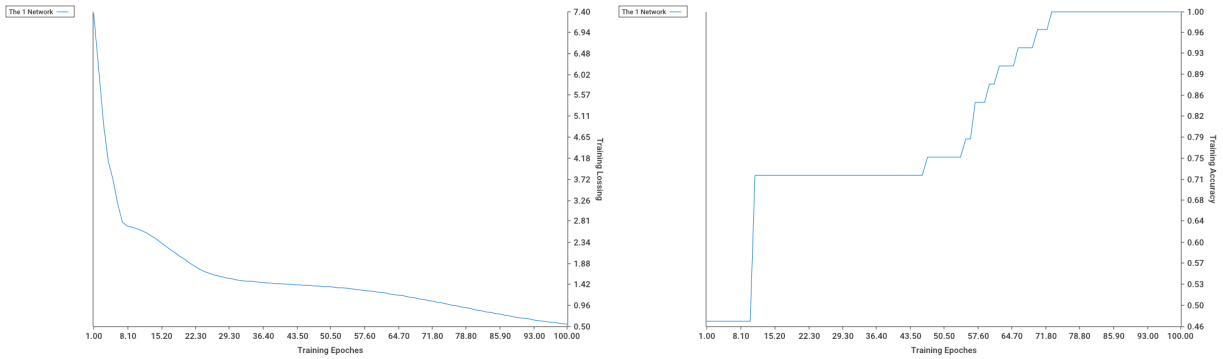

Figure 2: The Illustration of loss decreasing and accuracy increaing for 100 training epochs

3 MPI for distributed memory system

Unlike OpenMP, MPI [7] is designed for massage passing in a distributed system, which has its own memory address space. In practice, each process in MPI can be treated as an individual runing process, and those processes can parallelly run in multiple cores of one CPU, or multiple nodes in supercomputer cluster. The communication between those processes can be classified as two difference types [5], namely non-collective (point-to-point, with sending and receiving) and collective (all vs all or all vs point, such as with reducing).

Each process has its own rank, which is identifier of those processes. For point-to-point communication, such as sending and receiving, its destination or provenance process should be identified by calling. Those codes can only be executed in the code block, which we special identify. While collective directives do not need to be in a special code block, which means all processes should be able to execute this instruction, for example broadcasting and reducing.

[7] https://www.open-mpi.org/

In our report, our MPI is implemented with C. In order to compat this requirement, we choose golang as program language to build our network. MPI package for golang[8] has been already available. For deeper perspective, in order to execute MPI instructions, golang will use cgo[9] package, which leads to many overhead. But comparing to the time consumption of network training, this is a acceptable compromise.

After the codes have been finished, golang can compile our code to binary executable code. By the way, we can directly execute our code with MPI instruction. The number of process is dynamically determined by MPI execution such as following code.

```
1    mpirun -n 4 ./goai
```

4 Project architecture

Before we are going to describe the structure of our project, we want to mention one biggest problem we are currently facing in AI. The training process of a model is expensive costly, while with parallel computation we can significant reduce the duration of training process. For current AI training task, enormous training data are going to be fed into model, but with parallel computation we don't need to feed all data into one model. In our project, there are one main network and many training networks. The training data can be separated according to the number of nodes, and then be fed into each network.

4.1 Data processing

As we now want use multiple nodes to accelerate the training process, we better use a large training dataset, now intel image classification dataset comes into playground. In our project, iris dataset is used for personal computer with MPI, while intel image classification dataset will be handled by the clusters in GWDG[10] with MPI.

To starting our project, the first step is to divie our training data. Each process will be identified by its rank, with its rank the whole training dataset will be equally separated to each process. For general case, a shuffle procedure is implemented before the data segmentation.

4.2 Initial weights synchronization

As our second step is to initialize a model for each process, and randomly initialize all weights and biases[11]. Then in our main network, which has rank of 0, we pack the weights and biases in all layers into an array [12], and then broadcast this array to all other networks[13]. Each training network will receive this broadcasting from main network and synchronize its corresponding values of initial weighs and biases in each layer[14], then each training network train its model with its corresponding training data.

[8] https://pkg.go.dev/github.com/cpmech/gosl/mpi
[9] https://pkg.go.dev/cmd/cgo
[10] Gesellschaft für wissenschaftliche Datenverarbeitung mbH Göttingen (GWDG)
[11] network.Initialization ()
[12] networker.PrepWeightToBoadcast (MPIDATA), MPIDATA is the array we are going to boradcast
[13] newComm.BcastFloat64s (MPIDATA, 0)
[14] networker.ReciveInitialWeight (MPIDATA, rank)

4.3 Non-collective approach

In this subsection we will start with communication between different networks, and using MPI to accelerate the training process. For non-collective approach we use sending and receiving methodology, so the communication will only happen point-to-point.

In this approach our main network will not participate network training process, so we have one process less than the number of launched nodes to train data. But because the data segmentation process is executed by all launched processes, so the training data in main network will not be used at all.

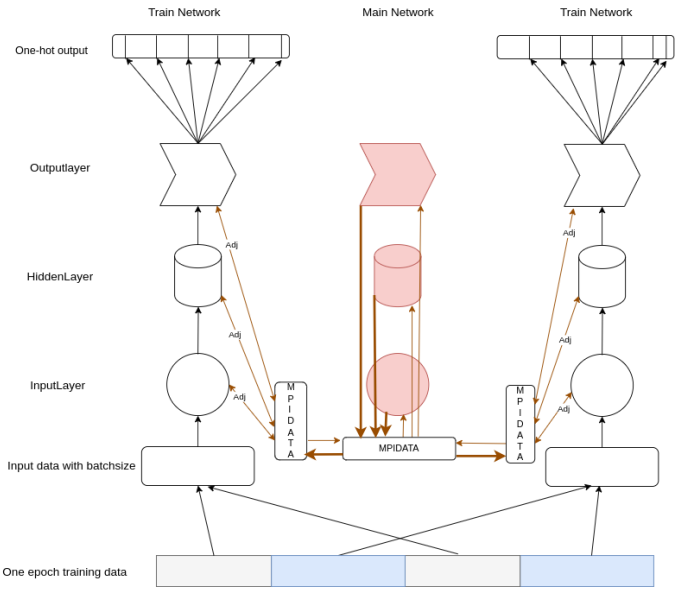

Figure 3: The whole procedure of weights updating with MPI communication using non-collective approach. After each training batch we collect all weights into array (thin yellow arrow to MPIDATA), then such arrays in training network will be accumulated in main network (thin yellow arrow in main network). Nextly we broadcasting the summarization to each training network (thick yellow arrow).

The complete procedure can be found in Figure 3, After data segmentation and initial weights from broadcasting, the next step is to launch the training process[15]. In this step, we only feed one instance to the network for each training process, and one training process contains forward training (forward passing through 3 layers, 3 time standardization, 3 times active function application), loss calculating (comparing one-hot label with network output), error backward propagation (backward passing through 3 layers, 3 times derivative of active function application, the variance of weights scaling with learning rate).

After the variance of each weight is derived, we do not work the update procudure. As we set batch size to be 4, such training process will be continually executed 4 times, and the variance of this 4 times will be accumulated together, the next step is to collect all accumulated value from all layers and pack them into an array[16]. This array will be passed

[15] networker.TrainWithEpochMPI

[16] networker.PrepSendAdj (MPIDATA)

to main network thought MPI[17]. On the other hand, main network recives such arrays from all train networks[18], and accumulate them all to another vairable [19], and we call it as Adj. After the addition of all training network have been finished, The last step is to send the Adj variable back to each train network again with array[20]. Since the train networks receive the new variance of weights and bias[21], which is from Adj variable, the variation will be added to corresponding weighs and biases in each layer[22]. We call all above mentioned procedure together as one training batch. The next training batch can be started again after the addition of weightes and biases of each layer is done.

As we determine batch size to be 4, how many training batches contains in every epoch is depends on the number of training instance of training dataset. Here we have two important factors to mention, one is that the batch size is also a super-parameter for model training, if we change it, the model performance can be very variational. But at same time, as you can see, batch size controlls the MPI communication frequency. This means, high batch size can reduce the MPI communication times. The second factor needed to be mention is in this approach, we manually lead to synchronization of each training batch, this can be a huge impact for project performance.

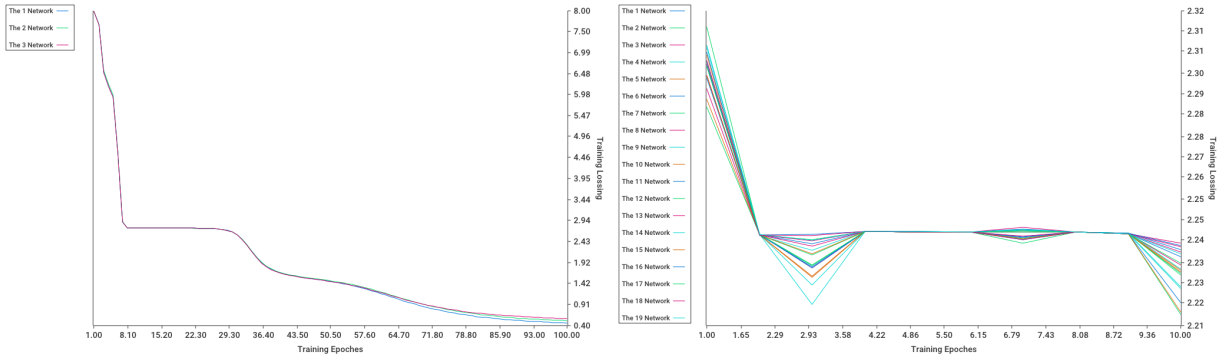

Figure 4: In this figure we present the loss decreasing of 2 datasets (iris for left side, and intel image classification for right side) for non-collective MPI communication approach. The number of training network is one less than nodes we launched, and on the right we only launch 10 epochs but we can still see the loss decreasing and weights synchronization.

After all training batches of each epoch has been done, we collect the accuracy and loss of our model. All the train network send its accuracy and loss to main network with MPI [23], and after main network receive the loss and accuracy for all epochs, [24], we plot it as png file as in Figure 4 [25]. The left side is from iris dataset for 4 processes, only 3 training

[17] newComm.SendFloat64s (MPIDATA, 0, rank)
[18] newComm.RecvFloat64s ()
[19] networker.UpdateWeightsMain (AdjTmp, MPIDATA)
[20] newComm.SendFloat64s (MPIDATA, mark, mark)
[21] newComm.RecvFloat64s (0, rank)
[22] networker.ReciveInitialWeight (MPIDATA, rank)
[23] newComm.SendFloat64(loss, 0, rank) and newComm.SendFloat64(accuracy, 0, rank+1)
[24] newComm.RecvFloat64(mark, mark), and newComm.RecvFloat64(mark, mark+1)
[25] DrowLoss (Loss, numEpochsenv, parallelism), and DrowAccuracy (Accuracy, numEpochsenv, parallelism)

networks are presented, their loss decreased well as single node before. After 100 epochs, the loss is about 0.3, which is significant better than single node for 0.5. While on the right side we show that 20 processes for intel image classification dataset were launched in GWDG cluster, and only 19 training processes worked the training procedure. We can also clear see that, loss are decreased, that means our model works. Because this is a image classification task, without convolutional network, reside network and adjustable learning rate, it hard to performance very well.

4.4 Collective approach

In this approach we still call the process to be main network, which has rank of 0, and other networks are called training network. While all network, both main and training network, will train their model with their segmented training data.

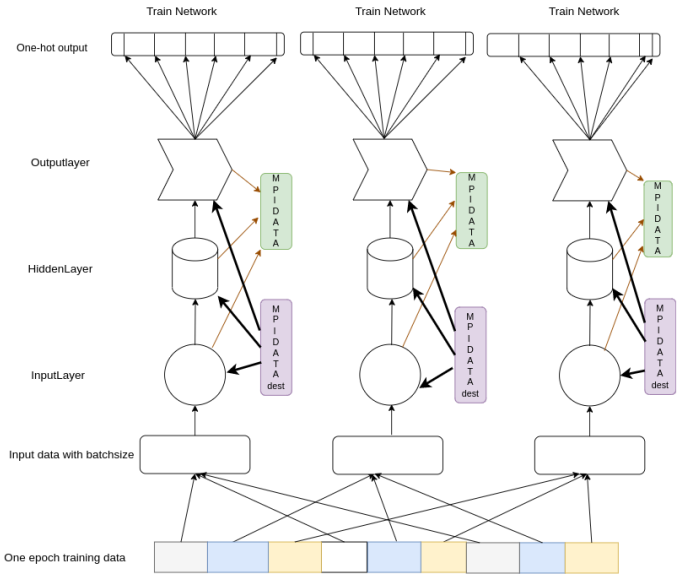

Figure 5: The whole procedure of weights updating with MPI communication using collective approach. After each training batch we collect all weights into array (MPIDATA). With allreduce MPI instructions we need another array (MPIDATAdest) to receive the summarization of variance for each weight, and distribute it to each layer for next training batch.

In this subsection we introduce a collective approach. Comparing to non-collective approach, as we can see in Figure 5, the data segmentation and weights initialization are the same. Also the procedure of 4 times accumulation of variance for each weights at 3 layers are the same, and collection for array (MPIDATA). The different in collective approach is that we use allreduce MPI instruction to recive the new updating value for each weight. By this way an array, which is called MPIDATAdest, is constructed for the destination of allreduce. The last step is to update weights and bias with MPIDATAdest after allreduce instruction is complete in each train batch.

For collective approach, batch size behaves just like in non-collective approach, but comparing to sending and receiving, allreduce has the computation resource of main network, also benefit from non synchronization in main network for array construction.

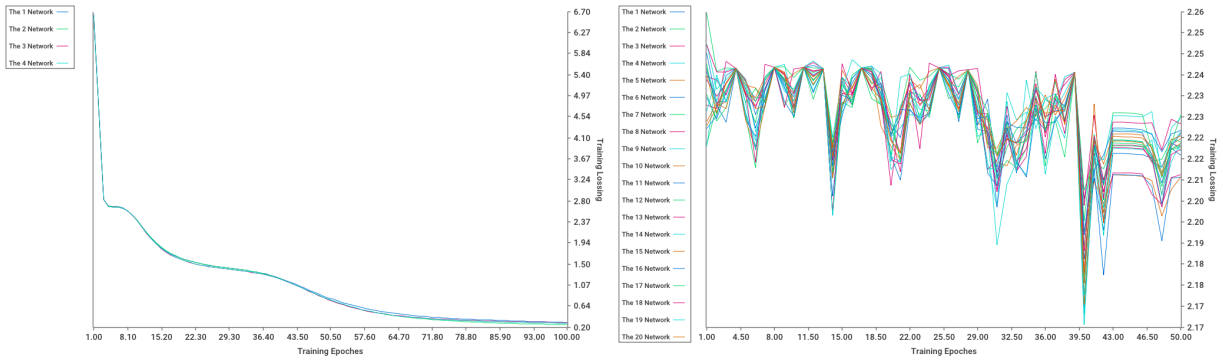

Figure 6: In this figure we present the loss decreasing of 2 datasets (iris for left side, and intel image classification for right side) for collective MPI communication approach. The number of training network is the same as the nodes we launched.

For collective approach we also plot the loss in Figure 6, here we launched also 4 processes for iris dataset in personal computer, and 20 processes for intel image classification dataset in GWDG cluster. All networks are presented as in lengend, and the decreasing loss performances are very likely as non-collective approach.

5 Project performance

In this section we are going to present the acceleration results because of the application of MPI. The settings of model can be found in Appendix C, as our project has two training dataset, we show the results of iris dataset at first. By the way, the performance of network is significant depends on the initialization, so the results of iris is from the average of 5 times measurements.

In Figure 7, we show the time consuming in minutes from 1 node to 8 nodes. Because for non-collective approach we need at last 2 nodes, there is no value for non-collective approach for one node.

Up to 6 nodes, we can see that the average time comsumpation is continually decreasing, both for collective and non-collective approach. In this region we benefit from the increaseing of nodes. After that, because in my personal computer has only 8 cores, I can only maximal have 8 nodes, while the time comsumpation is not decreasing, rather increaseing. This indicate that the new MPI communication time between nodes is larger than we benefit time comsumpation from increaing nodes.

While for intel image classification dataset we can only run it in GWDG cluster, this give us a bigger parameter space of nodes. For non-collective and collective approach we launched the model for multiple time with different nodes [26]. As in Figure 8 we see the time comsumpation curve for non-collective (with sending and receiving) and collective (with allreduce) approach.

For collective approach the time comsumpation decrease all the time, from about 90 minutes to 40 minutes when the number of nodes from 7 increase to 30. Up to 30 nodes we

[26] nodes number are 7, 8, 10, 12, 16, 20, 25, 30

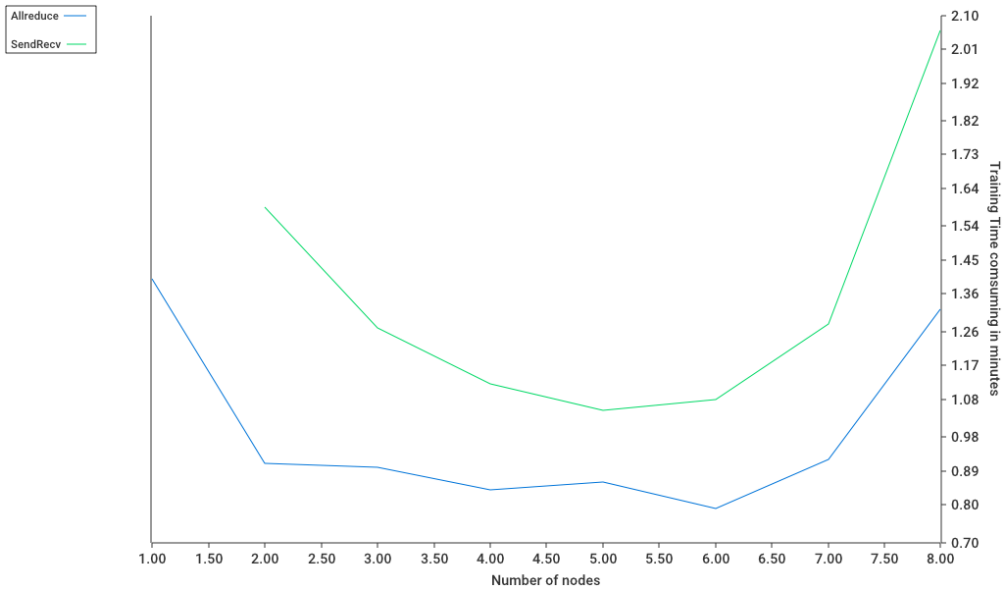

Figure 7: The Illustration of speedup diagram for iris dataset, non-collective approach (upper) starts with at least 2 nodes, while collective approach (lower) can starts with only one node. Both approaches indicate that time comsumpation is decreasing at begin and turns to increase for more than 6 nodes.

do not see time comsumpation increasing scenario, which means our model benefit always with the nodes increaing.

For the other approach, namely non-collective, we see the time comsumpation is generally 2 to 3 times bigger than collective approach. This is because in this approach we have complicated array collection and distributing process. Also each MPI communication works as a synchronization for all nodes. At the begin, up to 16 nodes, we see also the significant benefit from task parallelism. This acceleration is almost linear, but after that, it indicate slowly continually increaing of time comsumpation with the increaing of nodes up to 30. Therefore we can identify that with the node increaing more than 16, we can not decrease the time comsumpation of our model by add new nodes. Here the MPI communication causes more overhead than benefit from task parallelism by adding more nodes.

6 Conclusion and outlook

In this report we review the process of build a deep learning model from scratch with golang, and show its performance as single node network for iris dataset. After that we start to discuss MPI communication, here we introduce two methodologies, collective and non-collective. Because of the compatibility of golang to C language, we can use mpi to build a distributed deep learning model.

In the next part of our report we focused on the implementations of two approaches, which are collective and non-collective. Because of the different communication mechanism of non-collective and collective, we have to prepare difference arrays for two sides. For the two approaches, we see both loss decreasing for iris dataset and intel image classification

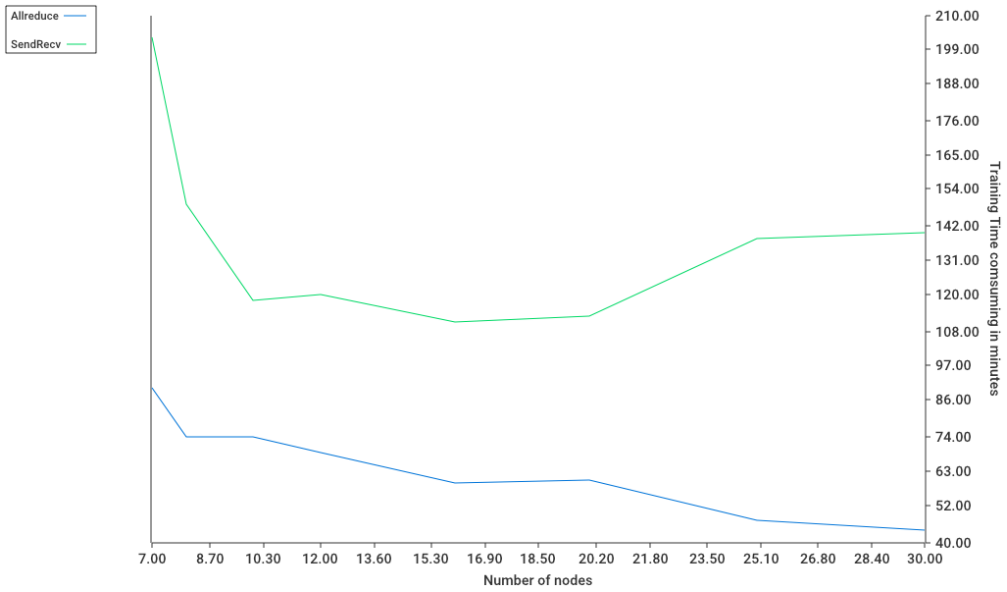

Figure 8: The Illustration of speedup diagram for intel image classification dataset for non-collective approach (upper) and collective approach (lower). Both approaches indicate that time comsumpation is decreasing at begin. After the number of nodes is more than 16, we can see the overhead of MPI communication between nodes suppress the benefit from nodes increaing.

dataset. But comparing to other famous image classification platform, our implementation has very poor performance. Here we also mentioned batch size can change the frequency of MPI communication also has a big influence on model training process.

As for the last part of this report we illustrated the acceleration of our model while training with MPI. Our training data is equally divied for launched nodes, each node starts a process to train its model. For small iris dataset, both collective and non-collective we can clear see the benefit from task parallelism, and how it is suppressed by the overhead of too many nodes. While for bigger intel image classification dataset we only see the same scene like iris dataset for non-collective communication. Up to 30 nodes the time comsumpation of collective communication benefit always from increaing the number of nodes.

For this report we need to mention, the topic belongs to distributed learning, which is also a current worthy research topic. About our implementation there are many aspects can be improved. Such as the implementation can contain convolutional network, reside network, dropout, learning rate decreasing. Also the accumulation of all nodes implys a synchronization, which can also be optimized.

References

[1] Lucas Beyer Alexander Kolesnikov, etc. Big transfer (bit): General visual representation learning. *arXiv:1912.11370*, 2020.

[2] Lucas Beyer Alexey Dosovitskiy, etc. An image is worth 16 x 16 words :transformers for image recognition at scale. *ICLR 2021*, 2021.

[3] Kozlov V. Y. Garcia, A. L. A container-based workflow for distributed training of deep learning algorithms in hpc clusters. *arXiv*, 2022.

[4] Javeed A. Devine K. Pearson, C. Machine learning for cuda+mpi design rules. *arXiv*, 2022.

[5] Gracia J. Schneider R. Zhou, H. Mpi collectives for multi-core clusters: Optimized performance of the hybrid mpi+mpi parallel codes. *arXiv*, 2020.

A Networks implementation with variables name

nn.InitData for data initialization:
X, Inputdata
y, outputlabels
nn.FInputData2InputLayer for data training in inputlayer

$$y_1 = W_1 * X + b_1 \tag{5}$$

W_1, wInput2InputLayer
b_1, bInput2InputLayer
y_1,dInputLayerTmp
nn.ActiveFuncInputLayer applying active function in inputlayer

$$Y_1 = sigmoid(y_1) \tag{6}$$

Y_1,dInputLayer
nn.FInputLayer2HiddenLayer for data training in hiddenlayer

$$y_2 = W_2 * Y_1 + b_2 \tag{7}$$

W_2, wInputLayer2Hiddenlayer
b_2, bInputLayer2Hiddenlayer
y_2, dHiddenlayerTmp
nn.ActiveFuncHiddenLayer applying active function in hiddenlayer

$$Y_2 = sigmoid(y_2) \tag{8}$$

Y_2, dInputLayer
nn.FHiddenLayer2OutputLayer for data training in outputlayer

$$y_3 = W_3 * Y_2 + b_3 \tag{9}$$

W_3, wHiddenlayer2Outlayer
b_3, bHiddenlayer2Outlayer
y_3, dOutputlayerTmp
nn.ActiveFuncOutputLayer applying active function and softmax function in outputlayer

$$Y_3 = sigmoid(y_3) \tag{10}$$
$$Y_3 = softmax(Y_3) \tag{11}$$

Y_3, dOutputLayer
nn.Output calculating the L2 loss function of outputs(Y_3) and labels(y),

$$L = \frac{1}{2} \sum_{i=1}^{batchsize} (Y_3^i - y^i)^2 \tag{12}$$

nn.BOutput2OutputLayer update the weights and bias in ouputlayer

$$\frac{\partial L}{\partial W_3} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot Y_2 \tag{13}$$

$$\frac{\partial L}{\partial b_3} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \tag{14}$$

$Y_3 - y$: dOutputLayerErr
$y_3(1 - y_3)$: dOutputlayerSlope
$(Y_3 - y)y_3(1 - y_3)$:dOutputLayerAdj
$(Y_3 - y)y_3(1 - y_3)Y_2$:wHiddenLayer2OutLayerAdj

$$W_3 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot Y_2 + W_3 \tag{15}$$

$$b_3 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) + b_3 \tag{16}$$

nn. BOutputLayer2HiddenLayer update the weights and bias in hiddenlayer

$$\frac{\partial L}{\partial W_2} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot Y_1 \tag{17}$$

$$\frac{\partial L}{\partial b_2} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \tag{18}$$

$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3$: dHiddenLayerErr
$y_2(1 - y_2)$: dHiddenLayerSlope
$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2)$: dHiddenLayerAdj
$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot Y_1$: wInputLayer2HiddenLayerAdj

$$W_2 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot Y_1 + W_2 \tag{19}$$

$$b_2 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) + b_2 \tag{20}$$

nn. BHiddenLayer2InputLayer update the weights and bias in inputlayer

$$\frac{\partial L}{\partial W_1} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1) \cdot X \tag{21}$$

$$\frac{\partial L}{\partial b_1} = (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1) \tag{22}$$

$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2$: dInputLayerErr
$y_1(1 - y_1)$: dInputLayerSlope
$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1)$: dInputLayerAdj
$(Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1) \cdot X$: wInput2InputLayerAdj

$$W_1 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1) \cdot X + W_1 \tag{23}$$

$$b_1 \leftarrow \eta \cdot (Y_3 - y) \cdot y_3 \cdot (1 - y_3) \cdot W_3 \cdot y_2 \cdot (1 - y_2) \cdot W_2 \cdot y_1 \cdot (1 - y_1) + b_1 \tag{24}$$

B Slurm sbatch script in cluster

```
#!/bin/bash
#SBATCH --job-name mpi-go-neural-network
#SBATCH -N 1
#SBATCH -p fat
#SBATCH -n 20
#SBATCH --time=01:30:00

module purge
module load openmpi

mpirun -n 20 ./goai
```

C Configuration file

For iris dataset

```
inputdataDims=4
inputLayerNeurons=30
hiddenLayerNeurons=20
outputLayerNeurons=3
labelOnehotDims=3
numEpochs=100
learningRate=0.01
batchSize=4
```

For iris intel image classification dataset

```
inputdataDims=67500
inputLayerNeurons=40
hiddenLayerNeurons=10
outputLayerNeurons=6
labelOnehotDims=6
numEpochs=10
learningRate=0.004
batchSize=4
```