Data and Model Parallelism in MPI

Yuchen Wang, Junli Cao

December 2022

Summary

We implement data and model parallelism for MLP networks on CPU with Numpy and C++ back-ends. We only use built-in libraries for computation and the MPI protocol for message passing among nodes. For data parallelism, we implement various approaches, such as parameter server and AllReduce, and test them on different model configurations. Moreover, we implement the tensor model parallelism that splits the parameters into several devices. For both parallelism paradigms, we observe massive speedup on various model configurations.

Background

The key data structure in training neural networks on CPU is a multi-dimensional array (Numpy array in Python or nested vector in C++). The main operation is matrix multiplication, which has $O(n^3)$ complexity and can be computationally expensive. Training large-scale deep learning models on a single machine can be challenging due to the model's complexity and the data's growth. The training process can take hundreds of hours, making it necessary to parallelize the training process to improve efficiency at scale. There are several ways to parallelize deep learning models, including model parallelism and data parallelism.

Tensor model parallelism is a specific type of model parallelism that partitions the parameters in each layer and assigns them to different devices. Model parallelism involves splitting the model into subgraphs and assigning each subgraph to a different device. On the other hand, data parallelism divides the input data into smaller batches and feeds each batch to a different device, with the same model on each device.

We are focusing on a simple multi-layer perceptron (MLP) architecture for this project. We have implemented a module-based MLP network that supports automatic differentiation and allows the number of hidden layers and units to be customized. The key challenge in implementing parallel deep learning is efficiently communicating between nodes during the forward and backward passes. We have experimented with three different reduction algorithms: parameter server (centralized communication), naive AllReduce, and ring reduce.

Approach

Automatic Differentiation

Automatic Differentiation (AD) is a powerful technique used in deep learning systems to compute the gradients of a neural network's loss function with respect to its parameters. These gradients are then used to update the network's parameters during training using optimization algorithms such as stochastic gradient descent. There are several approaches to AD, including Backpropagation and Reverse AutoDiff, which both use computational graphs to calculate intermediate values during the forward and backward passes. Reverse AutoDiff is particularly useful because it only requires a single forward-backward pass to differentiate all variables, making it easy to apply gradient-based optimizations in a variety of scenarios.

However, building computational graphs with built-in libraries can be challenging. An alternative approach is a Module-based AD, which does not require an explicit computational graph. This method encapsulates high-level architectural components into modules that correspond to vector-valued nodes in the computational graph. Each module has a forward method and a backward method. The forward method computes the output of a differentiable function f applied to the input matrix X, resulting in an output matrix Y. The module also stores some information during the forward pass to prepare for derivative calculations in the backward pass. The backward method takes in the gradient concerning the output, G_Y, and returns the gradient of the input, G_X. For parametric modules, the backward pass also calculates and stores the gradient with respect to the module's parameters for later optimization.

We have implemented Linear, Sigmoid, and Softmax modules in both Python and C++. The Python implementation is based on one of the assignments in a Machine Learning course, and we use NumPy for matrix multiplication. In the C++ implementation, we use naive three-level for-loops for matrix multiplication. Each Linear module can be initialized with an input and output dimension, and the initial weights are drawn uniformly from the range $[-0.1, 0.1]$. By chaining multiple modules together, we can build an MLP model with an arbitrary number of hidden layers and dimensions. We have also implemented the cross-entropy function and its derivative for classification optimization. To improve numerical stability, we calculate the gradient of the softmax and cross-entropy functions in one step. Our C++ implementation can be found in the file `mlp.h`.

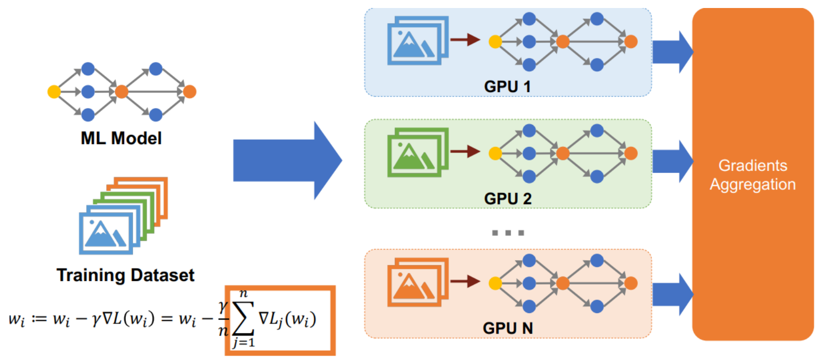
Data Parallelism

Data parallelism is a programming paradigm in which the same operation is performed concurrently on multiple data elements. In distributed machine learning, the data is partitioned into smaller chunks, and each chunk of the data is processed by a separate worker that uses the same ML model. The gradients will be aggregated and the updated weights/biases will be distributed to all workers. In short, each training step performs the following:

- **Each worker performs forward pass**
- **Aggregates the gradients from all workers**
- **Update the parameters to all workers**

Data parallelism can be implemented in various ways. In the project, we experiment the centralized communication pattern, such as parameter servers, and decentralized communication patterns, such as Allreduce.

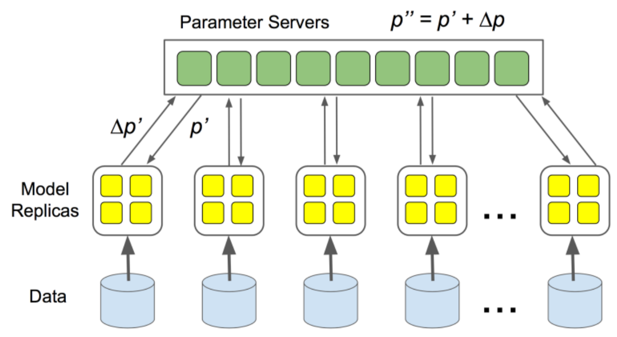
Figure 1: Pipleline of data parallelism


ML Model
Training Dataset
$w_i := w_i - \gamma \nabla L(w_i) = w_i - \frac{\gamma}{n} \boxed{\sum_{j=1}^{n} \nabla L_j(w_i)}$
GPU 1
GPU 2
GPU N
Gradients Aggregation

Parameter Server A parameter server is a centralized system that stores and serves model parameters to multiple worker processes. These worker processes are responsible for training the model on a subset of the data by communicating with the parameter server to retrieve the current model parameters, performing the training, and then sending the model parameter gradients back to the server. The server averages the aggregated gradients and updates the model parameters, allowing the model to be trained in larger effective batch sizes, which can improve the speed of the training process. However, the centralized communication patterns of the parameter server can make it difficult to scale to a large number of worker processes.

Our implementation of the parameter server differs slightly from the traditional approach. In addition to gradient descent, the root node also processes the forward and backward passes of a chunk of data, similar to one of the distributed workers. We have implemented the parameter server in both Python and C++. In the Python implementation, weights and derivatives are stored as Numpy arrays and the MPI4Py package is used to call MPI functions to broadcast weights and gather derivatives in each training step. An MLP model with L linear layers requires $2L$ MPI function calls. In the C++ implementation, we use two-dimensional float vectors to represent weights and derivatives. Before each MPI function call, these vectors are flattened into contiguous one-dimensional arrays, and an array of the same size is created as the receiving buffer. The weights or derivatives are then updated by reading sequentially from the receiving buffer. Our Python and C++ implementations can be found in `train_mnist_param_srv.py` and `train_mnist_param_srv.cpp`, respectively.

Figure 2: Pipleline of parameter server



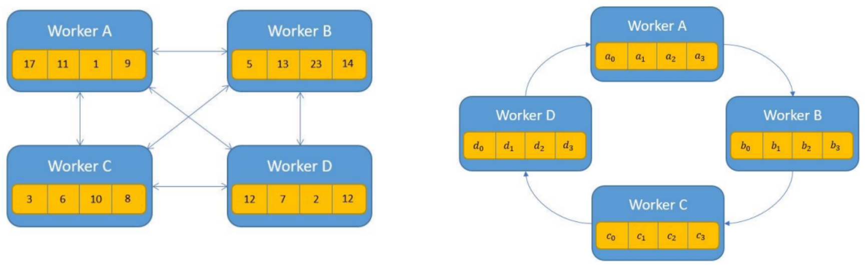
AllReduce Unlike the parameter server, which uses a central node to aggregate gradients and communicates with all computes workers, the Allreduce is a decentralized communication pattern that performs distributed reduction operations across a set of workers and broadcasts the results to all worker processes. In our implementation, we use AllReduce for gradient aggregation and each worker applies the gradients to its own parameters. However, one drawback of the naive AllReduce is the communication cost; with N workers and M parameters, the naive AllReduce requires $O(N^2M)$ communications.

To address this issue, we can use the Ring AllReduce, which arranges the workers in a logical ring topology (Fig. 3). Each worker sends its data to the next worker in the ring and receives data from the previous worker at the same time. Once each worker has aggregated the data, they send a slice of the aggregated parameters to the next worker. This reduces the communication cost to $O(NM)$ and is more scalable in scenarios with huge models and many workers.

We have implemented AllReduce data parallelism in both Python and C++. In the Python implementation, we either (1) directly call the AllReduce function provided by OpenMPI or (2) implement our own Ring AllReduce using the Isend and Recv MPI functions. The underlying communication pattern of (1) is unknown as OpenMPI selects the most efficient AllReduce implementation based on the input during runtime. In (2), we call Isend and Recv in each worker $2N$ times per training iteration. Note that we use Isend to send data asynchronously because using both synchronous send and receive leads to a deadlock in a ring dependency graph. We observe that both versions of AllReduce achieve comparable performance in terms of overall training speed. Our Python implementation of AllReduce and Ring AllReduce training can be found in `train_mnist_allreduce.py` and `train_mnist_ring.py`, respectively.

Figure 3: AllReduce

(a) Naive AllReduce (b) Ring AllReduce



In the C++ implementation, we only use the officially-provided AllReduce API for the sake of simplicity. Although we do not need to broadcast weights in every training iteration explicitly, we broadcast the randomly initialized model weights before training to ensure consistency across computation nodes. Our AllReduce C++ training implementation is included in `train_mnist_allreduce.cpp`.

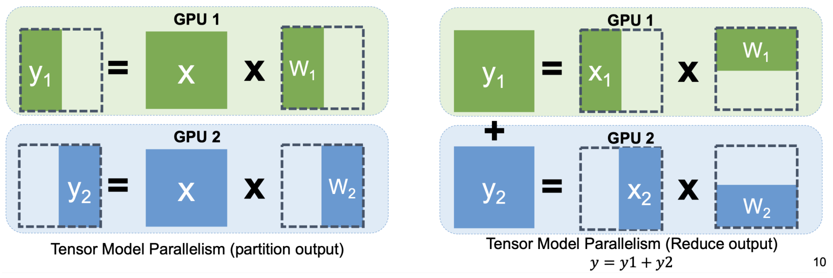
Model Parallelism

Data parallelism requires all devices to hold a copy of the entire model, which can be impractical when the model is large. Model parallelism, on the other hand, partitions the model into smaller slices, or "tensor slices," and assigns each slice to a different device. Tensor model parallelism is a specific type of model parallelism that partitions the model's parameters and gradients within layers and has two main implementations: partitioning the layer output or reducing the layer output.

Our implementation of tensor model parallelism consists of a single hidden layer and a single output layer. For simplicity and efficiency, the first layer uses the "partition output" paradigm, while the second uses the "reduce output" paradigm. This design allows the first layer's output to be directly fed into the second layer without requiring additional synchronization. On the output side, we sum the logits returned by each device and perform backward propagation on all workers.

Specifically, suppose the input and output dimensions are d_{in} and d_{out}, respectively, and the weights of the linear layers in each computational node have shapes $[d_{in}, d_h]$ and $[d_h, d_{out}]$. The overall model has Md_h hidden units evenly distributed across all workers. Our C++ implementation of tensor model parallelism can be found in the file `train_mnist_mpara.cpp`.

Figure 4: Two types of tensor model parallelism



Results

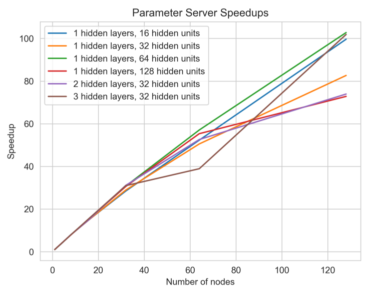
Experiments Setup We benchmark and profile our experiments with MLP networks (with multiple configurations, shown in Tab. 1) on the MNIST dataset that contains 60k training images. We attempt to run our experiments across multiple machines in a cluster. Nevertheless, due to technical issues, we profile all of our experiments on a single PSC machine instead. Moreover, to speed up the benchmarking process, we only run one training epoch for each experiment.

Table 1: Parallel Paradigms and Model Configurations. PS: parameter server, AR: AllReduce, TMP: tesor model parallelism

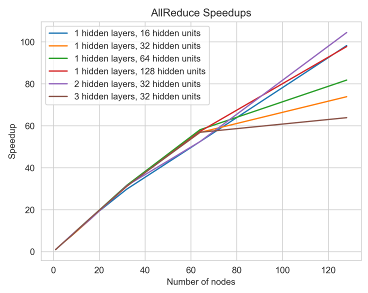
Number of Layers	Hidden Units	Strategy
1	16	PS, AR
1	32	PS, AR
1	64	PS, AR
1	128	PS, AR, TMP
1	256	TMP
1	512	TMP
2	32	PS, AR
3	32	PS, AR

We benchmark and measure our results speedup. The speedup measures how fast in time the distributed training strategies run one epoch compared to the model running with a single compute node. Moreover, we will briefly discuss the effectiveness of parallelization in deep learning. The effectiveness measures how effective in loss decrease our parallel implementations are for this classification task compared to the sequential version. The effectiveness is crucial and necessary as the convergence of deep learning models heavily relies on gradient updates. With too many nodes in the data parallelism paradigm, the model may converge slower, even with a higher speedup, due to fewer gradient updates in one epoch.

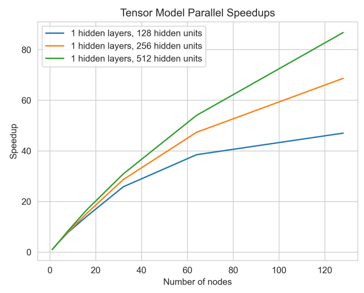
Measurement: Speed-up Fig. 5 shows the speedup of data parallelism (parameter server and AllReduce) and the tensor model parallelism. All three parallel strategies show significant speedup compared to the model with a single node on various model configurations. Specifically, the parameter server and AllReduce achieve almost linear speedup on most of the model configurations. One issue we observe in the plot is the speedup inconsistency across different models. For example, the more complex model with three layers and 32 hidden units in each has a lower speedup than the simpler model with one layer and 32 hidden units. This inconsistency could be explained by: although some models are more complicated than others, all of the models are very simple such that models with 128 workers can finish one epoch running in less than 1 second. A small perturbation during the training could result in a colossal speedup difference. This explains the abnormality in the plot.



(a) Parameter Server



(b) AllReduce

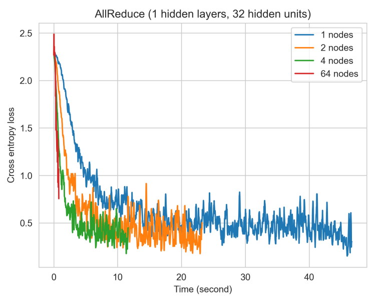


(c) Tensor Model Parallelism

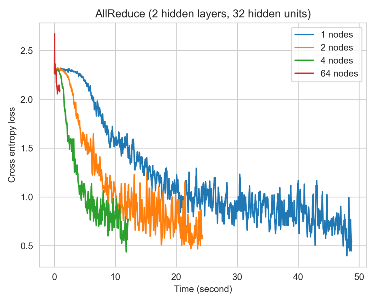
Figure 5: Speed-up

One interesting thing is that we observe an on-par speedup for the parameter server and AllReduce. For AllReduce, we observe a similar trend as the parameter server implementation: it shows linear speedup for most of the model configurations, though there is also speedup inconsistency. This observation disagrees with the theory that AllReduce should have much small communication costs than the parameter server, thus faster in general. However, if we take the fact into consideration that all experiments run on a single machine instead of a cluster so that the bandwidth between workers are not as limited as the one across clusters, the on-par performance makes sense as the communication costs are not bottlenecks in these experiments.

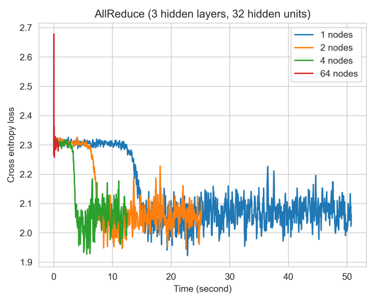
We run three models with tensor model parallelism: 128 hidden units, 256 hidden units, and 512 hidden units. This parallelization scheme shows great speedup as well. In this scheme, more complex models benefit more from parallelization. However, tensor model parallelism shows a lower speedup when using more workers on the same model (1 layer + 128 hidden units) than both parameter server and AllReduce. For example, with 128 workers, tensor model parallelism gives around 50x speedup, whereas the parameter server and AllReduce show more than 70x. However, this is not a fair comparison, nor indicating tensor model parallelism is worse than data parallelism, as tensor model parallelism can be used to train huge models that could not fit into a single GPU.



(a) 1 Layer MLP



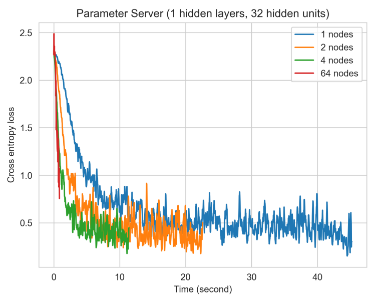
(b) 2 Layers MLP



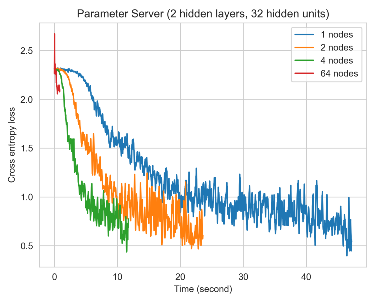
(c) 3 Layers MLP

Figure 6: Loss Plot for AllReduce

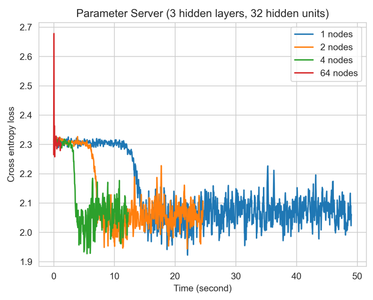
Measurement: Effectiveness We demonstrate the speedup gains when employing distributed training on various model configurations. However, running fast in one epoch does not necessarily lead to faster convergence, as more workers indicate fewer gradient updates within one epoch. Therefore, we might want to carefully choose the number of workers for our applications when using data parallelism.



(a) 1 Layer MLP



(b) 2 Layer MLP



(c) 3 Layer MLP

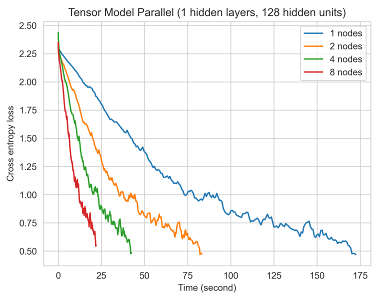
Figure 7: Loss Plot for Parameter Server

Fig. 6 and Fig. 7 show the loss decrease vs. time in seconds for various numbers of workers for parameter server and AllReduce. We can see that as the number of workers increases, the time to finish running one epoch decreases accordingly. For 64 workers, the experiment finishes one epoch in under one second. However, the loss only decreases a small amount compared to the 1, 2, and 4 workers. Therefore, we might need to run more epochs when we use data parallelism with a large number of workers in order to get a converged model.

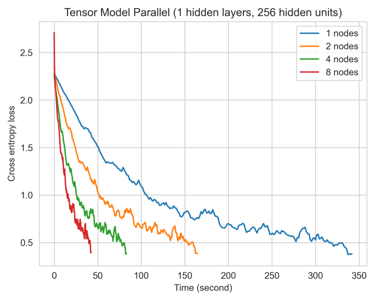
Different from data parallelism, the tensor model parallelism does not have the loss decrease issue. Fig. 8 shows the loss decrease vs. time for the tensor model parallelism. Since the number of gradient updates is independent of the number of workers in tensor model parallelism, the losses decrease to the same level for all choices of workers.

Reference

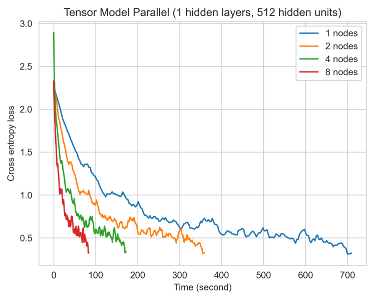
(1) 15-618 lecture slides. (2) 10-601 assignment five. (3) OpenMPI documentation. (4) MPI4Py documentation.



(a) 128 Hidden Units



(b) 256 Hidden Units



(c) 512 Hidden Units

Figure 8: Loss Plot for Tensor Model Parallelism

Work Distribution

- Yuchen: 50% - research module-based AD, research C++ implementation, research OpenMPI, benchmark the results.
- Junli: 50% - research computational graph, research Python implementation, research MPI4Py, analyze the results.