Data and Model Parallelism in MPI

Junli Cao Yuchen Wang
{junlicao, ywang7}@andrew.cmu.edu
Carnegie Mellon University

Introduction

For our final project we are implementing distributed training in machine learning. Specifically, we implement data parallelism and tensor model parallelism for MLP networks. For the data parallelism, we implement the Parameter Server and AllReduce using MPI.

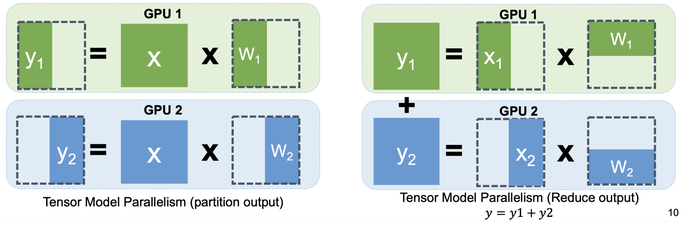
Background

- Neural networks can be computationally expensive to train, requiring the use of parallelization techniques
- Two main types of parallelization are model and data parallelism
- Model parallelism involves dividing the model into subgraphs, and data parallelism involves dividing the input data into smaller batches
- Efficient communication between nodes is necessary for implementing parallel deep learning, and different reduction algorithms can be used for this purpose.

Approach

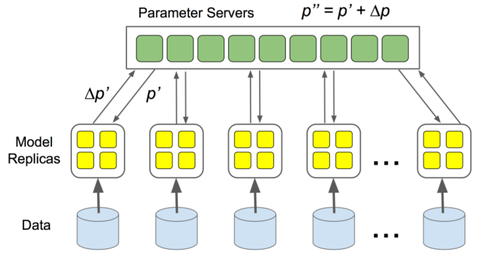
Automatic Differentiation

- There are several approaches to AD, including Backpropagation and Reverse AutoDiff, which use computational graphs to calculate intermediate values
- Module-based AD is an alternative approach that does not require an explicit computational graph and encapsulates high-level architectural components into modules with forward and backward methods
- We have implemented Linear, Sigmoid, and Softmax modules in Python and C++, and have used them to build a customizable MLP model with an arbitrary number of hidden layers



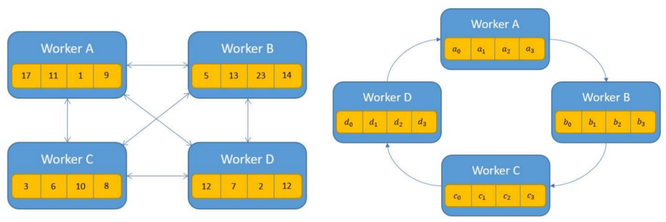
Approach

Parameter Server



- In our 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 our 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.

AllReduce

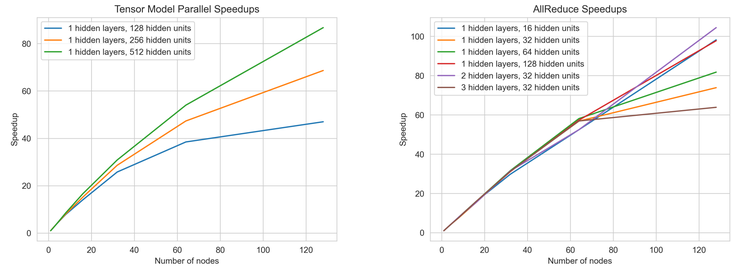


- The Python implementation uses the AllReduce function provided by OpenMPI or implements a Ring AllReduce using Isend and Recv MPI functions
- Both versions of AllReduce achieve comparable performance in terms of overall training speed

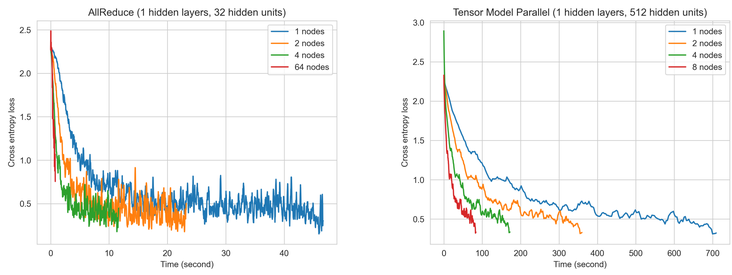
Tensor Model Parallel

- Our implementation of tensor model parallelism consists of a single hidden layer and a single output layer
- The first layer uses the "partition output" paradigm and the second layer uses the "reduce output" paradigm

Results



- Both parallelism show significant speedup compared to a model with a single node on various model configurations.
- AllReduce achieves almost linear speedup on most model configurations.
- AllReduce shows 100x speed-up on 128 hidden units model with 128 nodes, whereas tensor model parallelism shows 70x.
- The inconsistency could be due to the simplicity of the models and the possibility of small perturbations during training resulting in large speedup differences.



- As the number of workers increases, the time to finish running one epoch decreases.
- For 64 workers, the experiment finishes one epoch in under one second, but the loss only decreases a small amount compared to using fewer workers.
- The loss decreases to the same level for all choices of workers in tensor model parallelism, because the number of gradient updates is independent of the number of workers.

Reference

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