Jake Skinner
MPCS 52060 – Parallel Programming

Parallel and Distributed Deep Learning in Go

0. Key Terminology

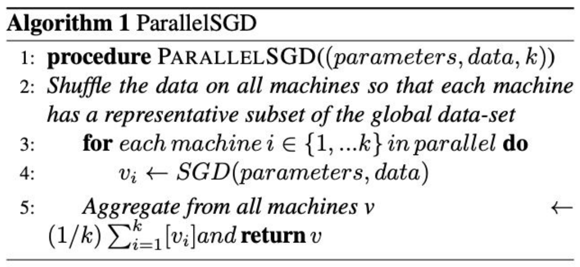
- *SGD = Stochastic Gradient Descent*
- *N = Number of Workers Performing SGD*
- *P = Number of Samples fed to Algorithm*
- *S^{p}_{n} = SpeedUp over sequential for P operations index and N number for threads*

1. Description

My project is a proof of concept and working prototype of a highly parallel and distributed architecture for training deep learning models and is implemented entirely in Go. At a high level the project takes advantage of the independent nature of Stochastic Gradient Descent to distribute work over N worker goroutines, while utilizing a centralized "broker" routine to aggregate results and apply adjustments to the parent model weights.The model is data-parallel architecture and employs BSP strategy for synchronization. It utilizes a circular barrier to ensure that no worker enters the next iteration without all N-1 workers having completed their work which ensures that after each iteration all worker values receive updated bais weight values. In addition to this the neural net that backs the underlying model is entirely bespoke and built from the ground up utilizing matrix operations to enhance both the forward and backward propagation algorithms. The idea for this project is loosely based on the following paper:

Hegde, Vishakh, and Sheema Usmani. "Parallel and distributed deep learning." In Tech. report, Stanford University. 2016

Here is the rough algorithm used to handle updating the model weights and bias as well as the conceptual ideas around communication of worker results:



Algorithm 1 ParallelSGD

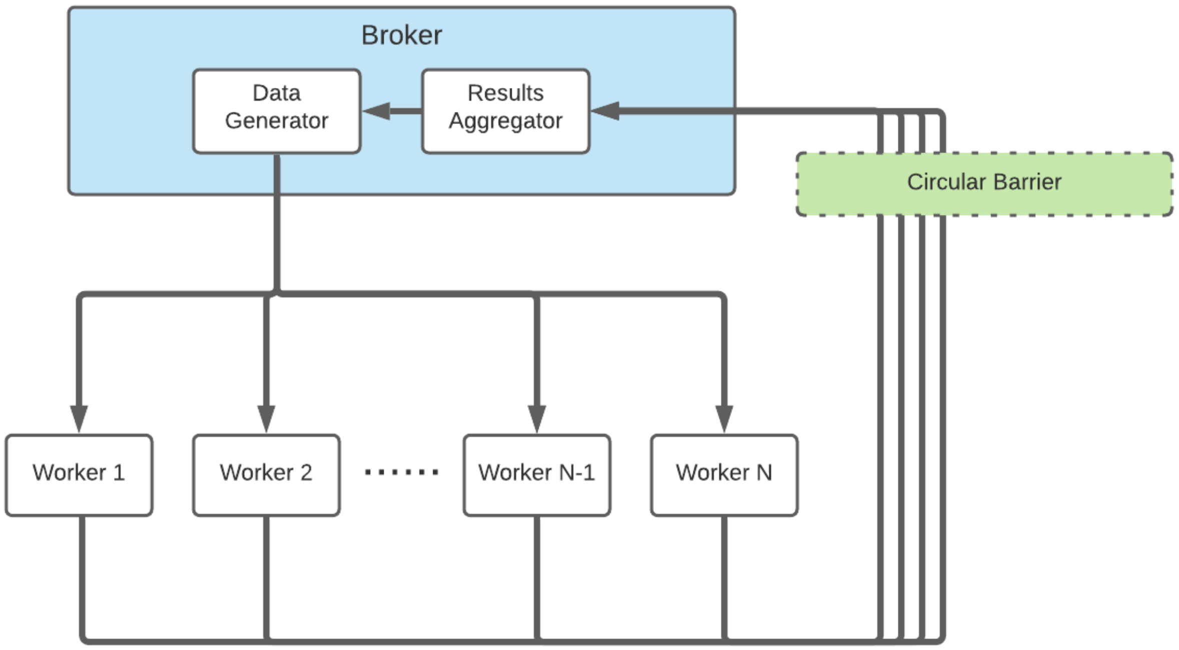
```
1: procedure ParallelSGD($(parameters, data, k)$)
2: Shuffle the data on all machines so that each machine
   has a representative subset of the global data-set
3:     for each machine $i \in \{1, ...k\}$ in parallel do
4:         $v_i \leftarrow SGD(parameters, data)$
5:     Aggregate from all machines v                    $\leftarrow$
   $(1/k)\sum_{i=1}^{k}[v_i]$ and return $v$
```

Note that that my implementation does NOT use batching and thus earac worker only holds a single data sample, thus there is no need to shuffle the data prior to sending to workers. That being said, data samples are randomly assigned each iteration to new workers using the Broker.ShuffleSlice array.

2. Architecture

At a high level the program can be decomposed into the following objects:

- **Broker:** Has a 1 to many relationship with all of the workers, it handles the main thread and is in charge of spinning off the Data Generator routine as well as the Results Aggregator routine. It is also in charge of marshalling the data generator and updating all worker model parameters. Has parent model which is the source of truth and the output of the program
- **Data Generator:** Is an independent subroutine within the Broker and handles the digestion of the data from the source input, utilizes a buffered channel for outbound communication
- **Results Aggregator:** Is an independent subroutine within the Broker, handles all inbound results on a results channel, performs linear algebra and produces new model weights and bias
- **Worker:** Is a consumer on the updates channel produced by broker, performs forward and backward propagation on one sample and then sends results to broker (results aggregator)
- **Circular Barrier:** Prevents any one worker from updating weights twice in a row without receiving new weights from the broker, enforces BSP



3. Execution Instructions

The underlying neural network was implemented by me personally, because of that fact it is a bit brittle in its use case and requires a large number of cmd line parameters. It is designed to run the MNIST which can be downloaded from here, it should be placed inside the proj3/inputs folder. Note that since Go is lacking in a number of statistical libraries i have had to do a good deal of hard coding standardization values for pixel values. If you would like to change the number of hidden layers in the model you can do so by adjusting the config setting in main.go.

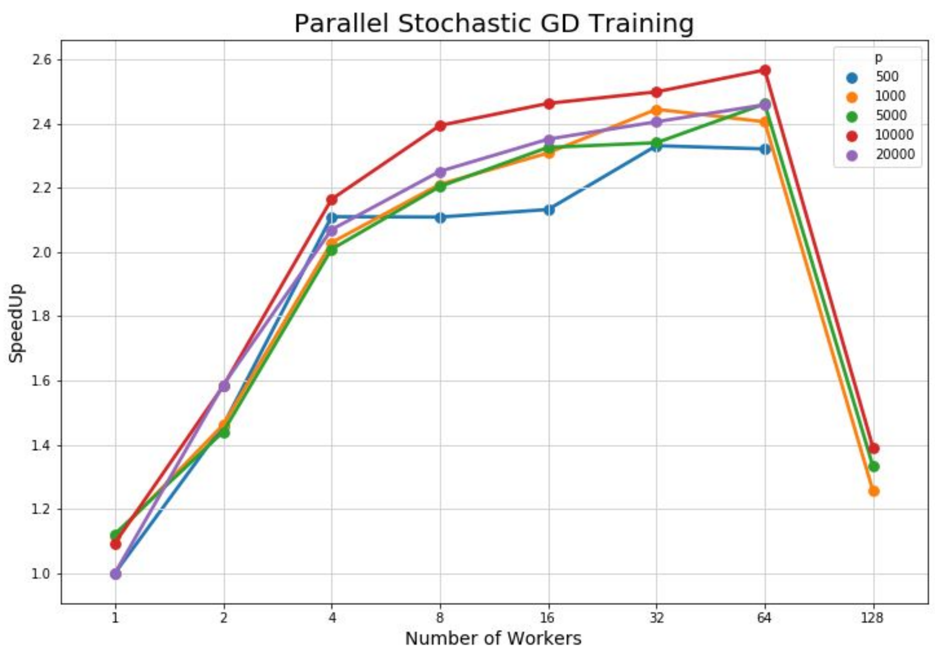
Testing Mode

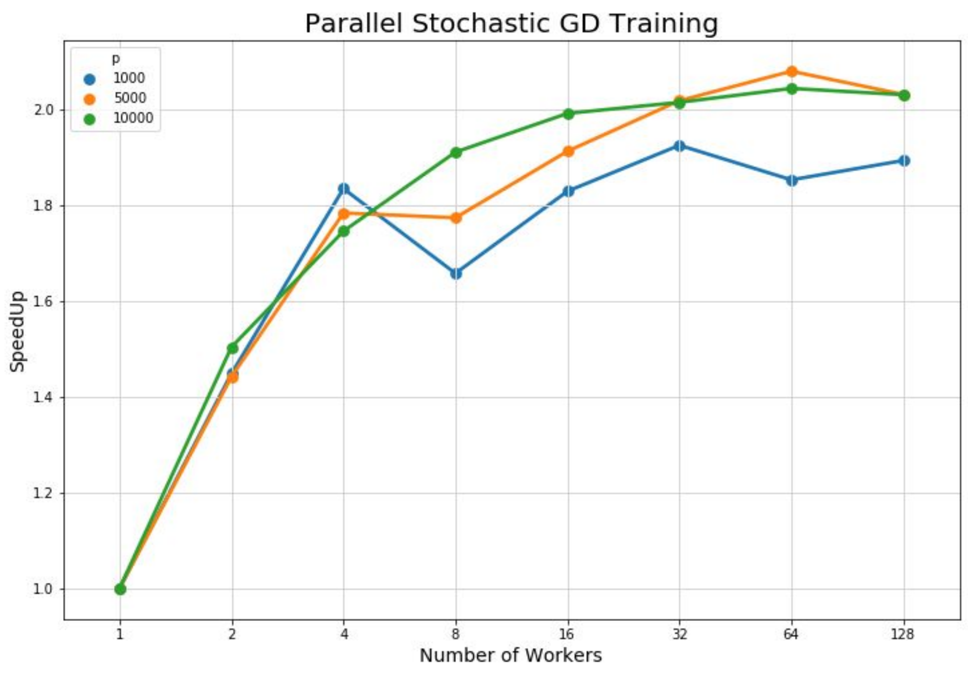
```
    $ cd proj3/testing/python
    $ python3 testing_script.py
```

Default Mode

```
    $ cd proj3/main
    $ go run main.go #input_nodes #hidden_nodes #output_nodes
#max_workers #max_trainsize #input_filepath #output_filepath
```

4. Results Discussion





For all testing runs the defaults settings were:

- Input Nodes = 784

- Hidden Nodes = 200
- Ouptut Nodes = 10
- Epochs = 5
- Learning Rate = 0.1
- Note that my kernel died on a number of the later examples

Overall the results of the Proj3 experiment look as expected, with execution time (and speedup) increasing nearly logarithmically in proportion to the number of workers. This conclusion holds for all P up until N > 64 where we see a large decline in performance due to massive thread overhead. It appears that after 16 workers (notably the number of threads on my machine) we no longer see large gains in speed up by applying additional workers, this holds true for all cases but appears to be less applicable to extremely large datasets where P > 10000.