Combining model and data parallelism for Convolutional Neural Networks on multiple GPUs

Vivek Veeriah (vveeriah) Aditya Modi (admodi)

December 30, 2017

Abstract

Deep learning methods have recently furnished the state-of-the-art results in computer vision and natural language understanding problems. A significant portion of these models, especially for vision, comprise of convolutional neural network architectures. Working on very large datasets like Imagenet (Deng et al., 2009) requires huge networks which often don't fit into the GPU memory along with the data. As such, we need to have a way of distributing the training across multiple GPUs and do it in an efficient manner. In this project, we use a GPU-aware MPI implementation of a hybrid parallel model for CNNs with cuda based libraries for implementing convolutional neural networks. Our results show that, the suggested parallelization trick is a feasible and plausible way of training large networks.

Keywords: Convolutional neural networks, multi-GPU programming, GPU-aware MPI, model parallelism, data parallelism.

1. Introduction

We implement a way of parallelizing the training of a convolutional network by distributing the network across multiple GPU nodes and use an SGD based implementation.

2. Modes of parallelism for CNN

As described earlier, we distribute the training of a deep convolutional network over multiple GPUs. Training CNNs is computationally intensive as it operates on large batches of images and uses internal activation layers of very large sizes. Figure 1 represents a previous state-of-the-art network used for winning the Imagenet challenge on image classification. As such, parallelizing the training of CNNs is quite important.

There are two broad ways of parallelizing a deep CNN:

- Model parallelism: We can divide the network itself across different GPUs/processors and different workers in the parallel architecture train different parts of the architecture. This needs communicating back and forth of the activations and the gradients across workers.
- Data parallelism: Different workers train on different training batches which are combined through synchronized or asynchronous gradient updates. This is a prevalent mode of parallelization as stochastic gradient descent enjoys favourable properties which makes the parallelization easier.

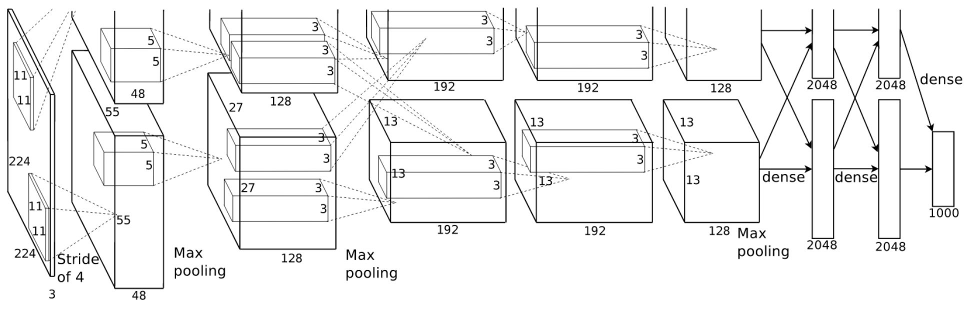

Figure 1: Alexnet architecture

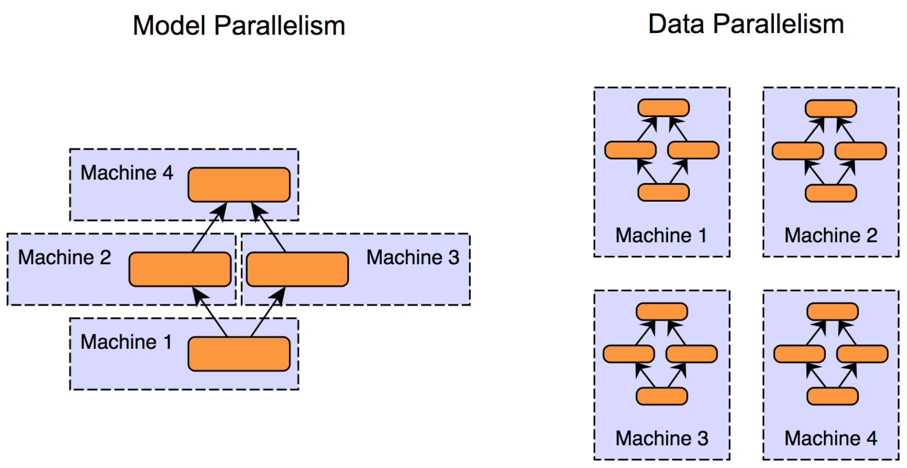

Figure 2: Two modes of parallelizaition of deep networks

There are two important aspects to look at for such parallel training methods:

- If we are using model parallelism, in a single forward or backward pass for a given minibatch of data, we will need to synchronize the different neuron activities or backward pass gradients across the workers whenever needed. If the networks are locally and sparsely connected, we can efficiently distribute the set of these activities. However, the case of image being a large matrix with no specific distributed connections poses a big bottleneck.
- In the case of data parallelism, we have to communicate the gradient matrix across the parameter worker configuration. For a big network with large fully connected layers, this can be a big ask affecting the wall time speed of convergence.

One would always wish to exploit all corners of parallelization in training big data models. We want to look at the case of convolutional neural networks which exhibit a specific property:

- In data parallelism, we would want the computation per weight parameter to be higher as we communicate or synchronize these parameters instead of the outputs. In CNNs, the convolutional layers reduce the different channels of images into big layers and therefore, involve a lot of computations. However, the number of actual parameters for a CNN is relatively very less (just the weights of a small filter kernel.) Therefore, the computation of these neuron activity is a computationally intensive task and we can divide the load among different workers, i.e., use data parallelism. But at the same time, synchronizing the weights or the gradients for data parallelism in convolution layers would need communicating these small kernels which is not taxing at all.

- In a fully connected layer, we have to compute a next layers of relatively small sizes (∼200-2000) by a linear combination of the outputs of a previous layer. In CNN, for the connection between a convolution layer output and a fully connected layer, the previous layer is of an image like layer which is large in size (`image_size * image_channels * batch_size`). Therefore, the computation of each of the output in the next layer involves higher computation. However, this computation is alarmingly less than the convolution layers and the number of parameters is huge (`conv_layer_size * fc_layer_size`). Therefore, we would want to use model parallelism in this case. This is because communicating the activations of the network (∼1.3 million values for a batch size of 64) is a lesser ask as compared to the communication required for the large number of parameters (∼44 million parameters for one block of alexnet).

This discrepancy of communication required for the two components of a deep network was brought in light by a position paper of Krizhevsky (2014). Motivated by that work, we implement our own hybrid model of parallelizing a convolutional network in cuda with GPU-aware MPI communication across multiple GPUs. We describe the model in the next section and describe the implementation details in subsequent sections.

3. Hybrid parallelization of CNNs

From the arguments above, we now discuss three approaches where we use data parallelism in the convolutional layers and model parallelism in the fully connected layers. As shown in figure 3, there are K workers which store the same convolution layers and the fully connected layers are divided in the next set of K workers. With this approach, we propose to parallelize the training in the following manner:

3.1 Forward pass for training

In figure 3, we have the following:

1. All K convolution workers receive a different minibatch of same size (say M).

2. All convolution workers compute the activations till the last convolution layer output. Now, there is an interface between the convolutional layer workers and the fully connected (FC) layer workers. The next FC layer is divided among K workers. For computing the activations of the neurons in their corresponding portion, the FC layers need the activation outputs of the convolution layer. After that, we perform a fully connected layer forward pass by doing a matrix multiplication.
   We communicate the convolution layer data in 3 different schemes:

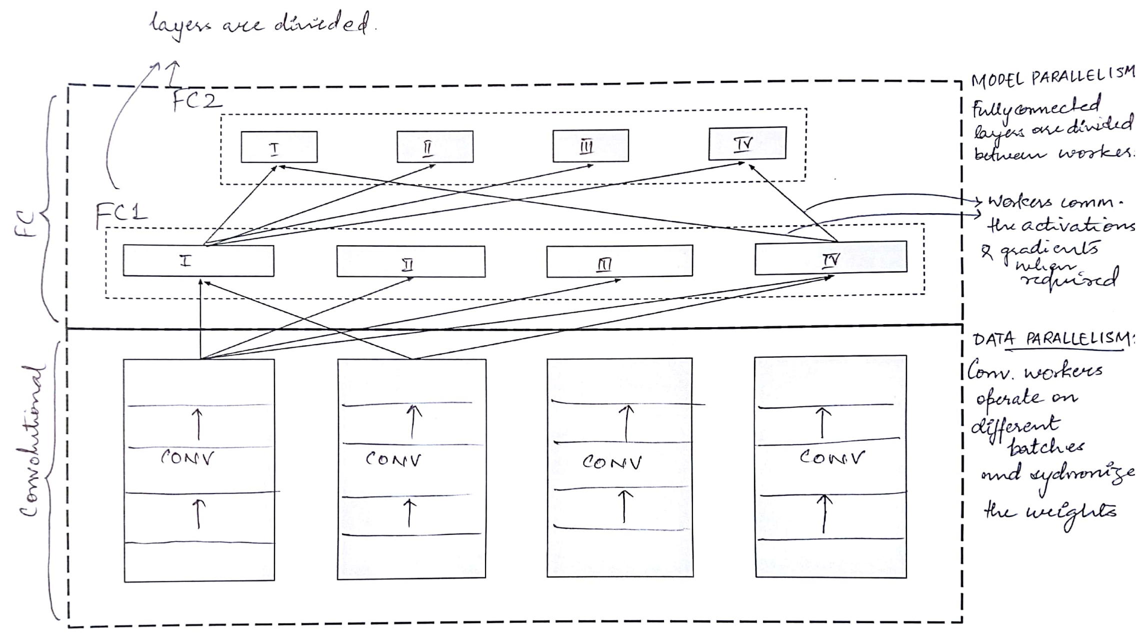
layers are divided.
FC2
I
II
III
IV
FC1
I
II
III
IV
FC
Convolutional
CONV
CONV
CONV
CONV
MODEL PARALLELISM
Fully connected layers are divided between workers.
Workers comm. the activations & gradients when required
DATA PARALLELISM:
Conv. workers operate on different batches and sychronize the weights

Figure 3: An example of the hybrid parallel setup with 4 GPUs.

(a) In this scheme, every convolution worker sends all of its activations to all workers in the fully connected layer. The FC layers would then have a batch of total KM convolution activations.

(b) The convolution workers send their activations turn by turn in a sequence. The FC workers then do the complete forward pass and the backward pass with batch of size M. After that, they take the minibatch of M outputs from the next convolution worker. This communication of the next batch happens **in parallel with** the computation happening for the outputs of previous convolution worker.

(c) This scheme is the same as (b) but changes things in the fully connected layer side.

In scheme (a), the fully connected layers have to wait to assemble the minibatch from all conv workers. As such, the communication is more blocking in this case. The batch size would also matter when we do the model parallelism portion in the fully connected layer. Larger the size of the minibatch, more activations, we will need to communicate between model parallel parts of the FC layers.

In scheme (b) and (c), the main benefit is that every worker (except the first one) sends its activations while the FC workers are computing the forward pass and backprop pass for the convolution outputs of the previous worker. This overlap of computation with communication hides the limited bandwidth and the latency of the network adapter.

The fully connected layers do the forward pass in the following manner:

1. Every FC worker, before computing the outputs for a certain layer, obtains the activations for the previous layer completely from all workers. This involves a merging operation for the

data and enforces a communication block.

$$FC2\_outputs = \sigma(W^\top FC1\_outputs)$$

(Need the complete previous layer, not just the part which I'm responsible for.) where W is the weight matrix.

2. This continues till the loss layer which in this case, we choose to be the multinomial logistic loss which is decomposable across final layer outputs and requires no communication. This reduces the communication required between the workers.

3.2 Backpropagation pass

The fully connected layers start the backpropagation for computing the gradients for the weights of the network. The backpropagation works as usual but there is again a communication which needs to happen. As shown in fig. **??**, every FC worker will compute a portion of a fully connected layer from previous activations. For this it needs the complete previous layer. Therefore, if worker i is computing $fc_k(i)$ for layer k, it uses fc_{k-1}. Therefore, in the backpropagation computation, every worker computes the gradient for fc_{k-1}. As such, the total gradient for the fully connected layer is distributed across workers. Every FC worker sends the values $\partial(loss)/\partial(fc_{k-1})$ to all other workers and then sums up the received $\partial(loss)/\partial(fc_{k-1})[i]$. Afterwards, each worker extracts the derivatives of the activation units for the portion it is responsible for.

In this way, the fully connected workers, each compute the last-stage convolutional layer activity gradients for the current minibatch. Now, depending upon how the minibatch was assembled, we need different communication for gradients:

(a) In scheme (a), every convolution worker had sent its activation outputs to the fully connected layers. Therefore, each FC worker, sends the correct portion of last-stage convolutional layer activity gradients to the correct convolution worker. After this, the FC workers update their parameters with the gradient and the complete iteration with batch size of MK completes. We will describe how the convolution workers update their parameters in a while.

(b) In scheme (b), the convolution workers send their last-stage convolutional layer activity turn by turn. As such, I send the complete last-stage convolutional layer activity gradients to the worker which I received the batch of size M from. After that, the FC workers update the parameters and then start on the minibatch of the next convolution worker.
Note: This is not gradient descent exactly as the convolution layer outputs use the convolution kernel parameters after last update from the previous iteration and the parameters of the FC layers are updated within one iteration. This introduces some asynchrony but we observe convergence of training loss in this case too.

(c) In this case, everything remains the same apart from the update. In this case, I store the gradients of the parameters and after doing the backprop for minibatches of all convolution workers, we sum the gradients up and update the parameters.

The convolution workers each do the backpropagation and compute the gradients of the parameters.

3.3 Synchronizing the gradients of convolution layers

After completing the backprop for the convolution layers, each convolution layer has a set of gradients. As we are using data parallelism in convolution layers, we need to synchronize the gradients. We do this simply by sending the gradients from each convolution worker to every other one and then average the gradients. In this manner, every worker makes the same update.

3.4 Challenges and difficulties

The communication setup here seems to be pretty straightforward as we need to do the right send and receive operations for achieving the correct communications. Implementing this in a CPU only manner will need less effort and might pose interesting issues for analysis. However, what we have done is, implement the same parallel procedure across multiple GPUs. In order to do this implementation in cuda across multiple GPUs, we had to study the documentation of two libraries for deep learning based on cuda implementations (C++) and establish ways of doing inter-GPU communication efficiently. We studied the method of doing GPU communication via MPI which we describe in the next section. Identifying the bugs and issues while communicating across GPUs was challenging and at the same time, quite interesting.

4. Implementation details

For implementing the network, we use GPU based cuda libraries for network components. We use the NVIDIA deep learning SDK and specifically, use cudnn to implement portions of the network. As certain computations are not supported by the cudnn implementation, we use cuBlas for computing the forward pass activations and backpropagation gradients for fully connected layers.[1]

The key step in implementing this parallel network is establishing communication between the GPUs. In the initial phases of the implementation, we used intensive device-to-host communication to pass values from one GPU to another. The pipeline for this transfer is (device1 - host1 - host2 - device2). This costs to separate communication bottlenecks between host and device. However, since 2015, MPI supports GPU-aware message passing. In this method, the MPI environment combines the address space for both device and the host memory. For implementing the parallelism across K GPUs, we spawn $2K$ threads where every GPU handles a full convolutional worker and $1/K$-th of the fully connected layer. These $2K$ threads communicate by using the GPU-aware MPI environment. A key benefit of using the GPU-aware method is that it pipelines the data transfer from host to device communication via PCIe-DMA transfers, host copies and host to host transfer using network adapter which hides the execution time for a significant portion of the code (fig. 4). This goes in similar to the kind of pipelining we have tried to implement in scheme (b) and (c). In addition to this communication, we implement a number of cuda kernels to merge/distribute a given vector and send appropriate sections to other GPUs.

5. Experiments and results

The network that we used for implementing our approach consisted of two convolutional layers, each of them subsequently followed by a max-pooling layer. Each of these convolutional layer consisted of a filter of size 5×5. The first convolutional layer consisted of 20 filters and the second

1. We had a hard time debugging this code as cuBlas uses column major notation and the data structure of cudnn is not very transparent.

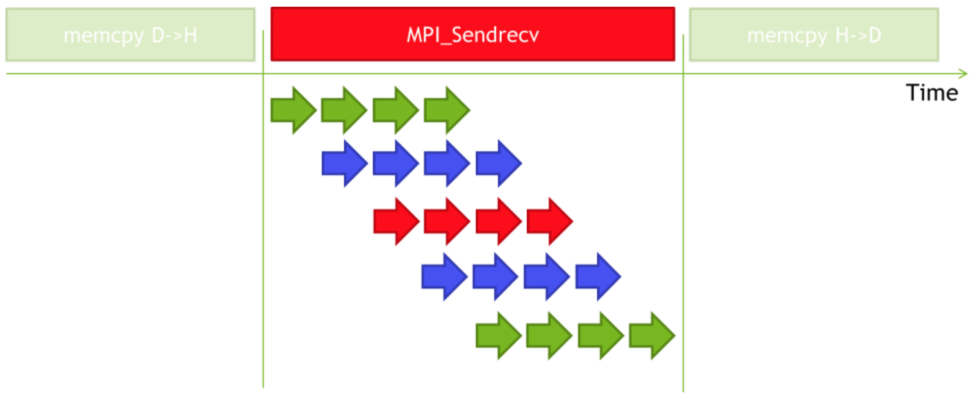

Figure 4: GPU-aware MPI

one consisted of 50 filters. Each of the max-pooling layer reduced the resulting activations from the convolutional layer by a factor of 2. After the convolutional and max-pooling layers, the neural network consisted of a fully connected layer of size 300 for the MNIST dataset and of size 600 for the CIFAR-10 dataset. A output multinomial logistic layer immediately followed this fully connected layer and had a size of 10 as there were 10 classes in the dataset.

The computing machine that we used to run our experiments consisted of 4 GPUs and 2 processors with 12 threads each. Each thread that operated the convolutional layer consisted of its own portion of the training dataset as it is operating under the principles of data parallelism. Similarly, the partitions of the fully connected layers were operated by different threads and used the available portions of the GPU in the system. Overall, each convolutional data parallel worker required a thread and significant amount of GPU memory. There were K such workers. This implied that the fully connected layers and the subsequent output layers were also partitioned into K fragments, each fragment being controlled by a unique thread. One problem which this might give way to is that for MPI the environment is using the same memory bus and only one thread is allowed access to the memory bus. As such, MPI with increasing processors takes more time to send data from one place to another than expected.

To evaluate the effectiveness of our parallelization approaches, we chose two standard image classification datasets. The first one is the MNIST dataset LeCun et al. (1998), which is considered to be standard benchmark in supervised learning literature. The MNIST dataset consists of $60,000$ hand-written digits, which are used as training examples. The hand-written images are have been size-normalized and centered in a fixed-size image. The supervised labels for these images range from 0 to 9.

The second dataset we used for evaluating our approaches is the CIFAR-10 dataset. It consists of $50,000$ training images. These images were collected from real-world objects, thereby making it a difficult image classification datasets. The supervised labels for these images are based on the category of the object present in the image. Specifically, the labels for these images take one of the following values: { airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck }.

Further, to take a look at the training curve, we see the following pattern:

As we can see, the training loss with increasing number of processors for each scheme **increases** takes more time to converge. However, for scheme b and scheme c, we see that the loss converges

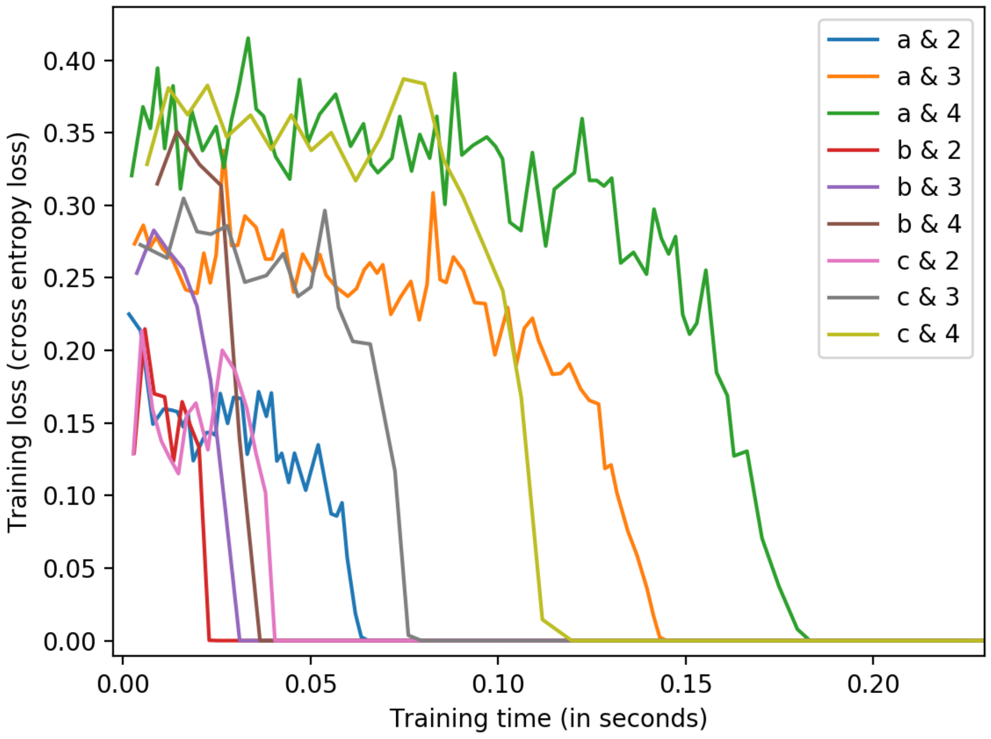

Figure 5: Training loss vs training time (MNIST)

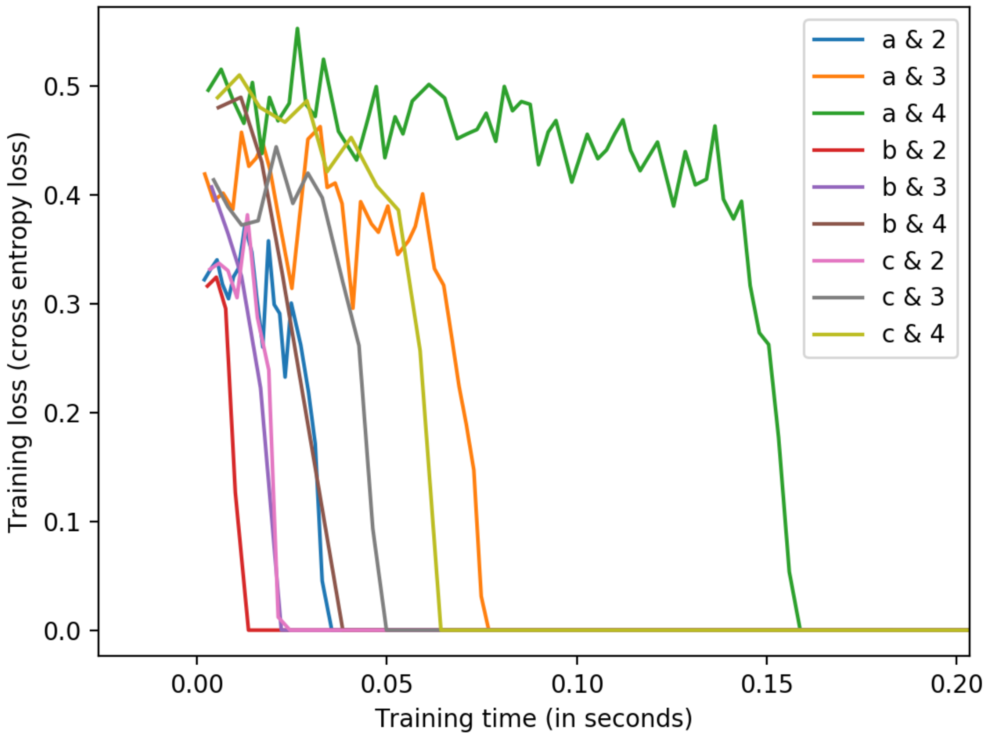

Figure 6: Training loss vs training time (CIFAR)

Configuration	Mean (mnist)	Std. Dev. (mnist)	Mean (cifar)	Std. Dev. (cifar)
Scheme a * 2	0.00184	0.00057	0.00183	0.00047
Scheme a * 3	0.00224	0.00051	0.00231	0.00058
Scheme a * 4	0.00576	0.00090	0.00586	0.00126
Scheme b * 2	0.00256	0.00052	0.00282	0.00046
Scheme b * 3	0.00425	0.00087	0.00407	0.00059
Scheme b * 4	0.00587	0.00107	0.00589	0.00096
Scheme c * 2	0.00257	0.0006	0.00284	0.00047
Scheme c * 3	0.00422	0.00069	0.00416	0.00056
Scheme c * 4	0.00592	0.00112	0.00583	0.00096

Table 1: Time for each iteration as measured from convolutional workers

Configuration	Mean (mnist)	Std. Dev. (mnist)	Mean (cifar)	Std. Dev. (cifar)
Scheme a * 2	0.00180	0.00053	0.00186	0.00052
Scheme a * 3	0.00228	0.00063	0.00231	0.00077
Scheme a * 4	0.00544	0.00075	0.00579	0.00077
Scheme b * 2	0.00253	0.00049	0.00278	0.00046
Scheme b * 3	0.00422	0.00081	0.00410	0.00059
Scheme b * 4	0.00591	0.00089	0.00581	0.00079
Scheme c * 2	0.00255	0.00060	0.00283	0.00052
Scheme c * 3	0.00418	0.00059	0.00416	0.00067
Scheme c * 4	0.00595	0.00114	0.00576	0.00087

Table 2: Time taken in one fully connected pass iteration (model parallelism)

faster than scheme a. We expect some speedup and not increase in training time with thi setup. To look closely into why this is happening, we see the various time taken for different components.

Iteration time: (Table 1) We can clearly see that the iteration time for all the schemes increases when we increase the workers. This suggests that there is some component which is increasing more than expected.

Fully connected layer: (Model parallelism) Here (table 2), while testing the average time taken for the fully connected pass over the layer shows that it is the fully connected computation which is increasing the training time.

Receiving pooling activations: (Table 3) Fully connected layers receive the pool layers from the MPI platform and there might be a longer wait time for the MPI call to complete. The calls are asynchronous send operations at all places with corresponding request variable being used to call `MPI_Wait`. However, the time taken for pool communication is surprisingly not the issue.

Combining the partial activations of fully connected layer 1: In this case (table 4), every fully connected layer worker, combines its own output of the FC activations with the portion of other workers. After doing this merge, the method calls forward pass over the data. This operation's time is increasing significantly over increasing the number of workers.

Dividing the gradient of the activity in FC layer: In this case, we see that the time again increases with the increase in number of workers.

Configuration	Mean (mnist)	Std. Dev. (mnist)	Mean (cifar)	Std. Dev. (cifar)
Scheme a * 2	0.00124	0.00033	0.00123	0.00033
Scheme a * 3	0.00144	0.00049	0.00131	0.00061
Scheme a * 4	0.00131	0.00032	0.00134	0.00037
Scheme b * 2	0.00091	0.00017	0.00103	0.00017
Scheme b * 3	0.00134	0.00028	0.00127	0.00026
Scheme b * 4	0.00133	0.00029	0.00135	0.00036
Scheme c * 2	0.00088	0.00018	0.00104	0.00017
Scheme c * 3	0.00133	0.00026	0.00125	0.00033
Scheme c * 4	0.00137	0.00041	0.00128	0.00029

Table 3: Time for which FC workers wait for receiving pool activations (Interface between model and data parallelism)

Configuration	Mean (mnist)	Std. Dev. (mnist)	Mean (cifar)	Std. Dev. (cifar)
Scheme a * 2	0.00016	0.00015	0.00019	0.00012
Scheme a * 3	0.00017	0.00001	0.00019	0.00009
Scheme a * 4	0.0019	0.00034	0.00199	0.00037
Scheme b * 2	0.00044	0.00026	0.00050	0.00029
Scheme b * 3	0.00137	0.00045	0.00080	0.00036
Scheme b * 4	0.0021	0.00047	0.00163	0.00035
Scheme c * 2	0.00045	0.00029	0.00048	0.00034
Scheme c * 3	0.00127	0.00041	0.00077	0.00036
Scheme c * 4	0.00153	0.00036	0.00189	0.00034

Table 4: Time FC workers take to receive and merge outputs of first fully connected layer (Model parallelism)

Configuration	Mean (mnist)	Std. Dev. (mnist)	Mean (cifar)	Std. Dev. (cifar)
Scheme a * 2	0.00010	0.00007	0.00012	0.00007
Scheme a * 3	0.00014	0.00005	0.00020	0.00005
Scheme a * 4	0.00184	0.00023	0.00188	0.00037
Scheme b * 2	0.00050	0.00016	0.00061	0.00018
Scheme b * 3	0.00066	0.00024	0.00109	0.00018
Scheme b * 4	0.00186	0.00033	0.00183	0.00030
Scheme c * 2	0.00049	0.00017	0.00064	0.00016
Scheme c * 3	0.00073	0.00023	0.00113	0.00018
Scheme c * 4	0.00183	0.00030	0.00181	0.00031

Table 5: Time FC workers take to reduce and extract derivatives of first fully connected layer activations (Model parallelism)

6. Discussion

From the results, we see that the network is not speeding up due to the model parallelism component. The idea behind using this hybrid parallelism is that there are lots of datasets which have large size and at the same time require large number of parameters. As such, (as was the case initially in early 2010s with GPUs), the network and data need to be segmented across machines. In this setting, it is important to look carefully at the specific design of the network and see what components of data are useful for distribution and what components of the network are good for division. There can be a large number of reasons because of which the results show a negative curve:

- Our dataset which we used are small against the motivation we use and have really fast converging rates. In case of CIFAR-1000, it has been shown that model parallelism is useful to divide the network in an efficient manner and is many a times necessary (Dean et al., 2012; Yadan et al., 2013).

- The datasets need to have sufficient variation in their examples, so that the effective batchsize can play a big role in learning. When we increase the number of workers, the number and variability of examples which we see increases and helps in learning faster. In our two datasets we considered, we mainly have two networks of slightly different sizes but with toyish nature of the data. In this case, the computation time is overshadowed by the communication time while using model parallelism.

- The idea of model parallelism is quite significant and is especially useful when we have locally connected networks which are rising up in the computer vision community. In Yadan et al. (2013), the authors train using a hybrid parallel model but use local connections to reduce communication. They show that there is speedup in CIFAR-1000 dataset.

- Training environment: In our case, due to lack of resources (training time and computational resources), we could not test out approach in the exact domain of big data where it applies. Therefore, this does remain a significant question for us: ”How can we analyse the timings/speedups (if any), obtained by using model parallelism?”. In case there is a speedup, our individual timing components suggest that this hybrid parallelism should work very well for such convolutional networks.

7. Conclusion

In this project, we attempted to implement a hybrid parallel model of deep convolutional network where the convolutional layers use data parallelism and the fully connected layers use model parallelism. We observed that the model parallel component of the network is playing a big role in the parallelization and hinders any speedup in case of smaller datasets. In totality, we did get this thing working on a multi-GPU system using GPU-aware MPI and using this in another place as an experience would be a great learning curve for us. Apart from this, we still wish to explore this hybrid parallelism idea as it has been shown to have some merit in case of big datasets. Understanding if this is a property of the gradient descent method or there is indeed a lot of room for model parallelism is something we wish to pursue.

References

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231, 2012.

J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Omry Yadan, Keith Adams, Yaniv Taigman, and Marc'Aurelio Ranzato. Multi-gpu training of convnets. *arXiv preprint arXiv:1312.5853*, 2013.