Spring 2019 CS744 Final Project Report
University of Wisconsin-Madison

Transparent Gradient Aggregation in Machine Learning Systems

Group 12 - DM1
Yuan-Ting Hsieh
John Truskowski
Han Wang

Contents

Our code can be found in https://github.com/YuanTingHsieh/horovod

1. Background

In Machine Learning systems such as TensorFlow and PyTorch, one specifies the training computation to be executed in terms of mini-batches, and it is expected that all computation on a mini-batch happens at once. In the distributed setting, if there are N GPUs, then one copy of the script runs on each GPU. If the script specifies the mini-batch size as B, then the global batch size will be NB, and with B data points per GPU. In each iteration, there is a forward and backward pass on the data batch, and the gradient is communicated to the master to be aggregated and applied to the global model. However, issues arise in the case where the scheduler may choose to revoke access to some subset of the GPU resources. For example, if N/2 GPUs are removed. To preserve the semantics of computation, each GPU must now run 2 mini-batches of size B, add up the resulting gradients, and send it to the master node to be aggregated and applied to the global model.

2. Problem Statement

In Horovod [1], a distributed training framework for popular deep learning frameworks such as TensorFlow, Keras, and PyTorch. There are several ways to do gradient aggregation, one is to specify that explicitly by the user [2], another way is to pass backward_passes_per_step to the DistributedOptimizer [3]. Both of these approaches need the user to explicitly specify how to do the aggregation. Our goal is to implement “transparent gradient aggregation” -- to modify Horovod such that changes in resource allocation can be automatically detected and an appropriate gradient aggregation strategy can be applied. This strategy should also be transparent to the underlying deep learning framework and the users, but the users don’t have to figure that out on their own.

Horovod’s architecture is based on well-established MPI concepts such as size, rank, local rank, allreduce, allgather and broadcast [4, 5]. Size would be the total number of processes in this application. And since horovod assume each process run on one GPU, size would be the number of GPUs. When doing operations, horovod launch a background thread to coordinate with all the MPI processes (operations.cc) [1] and it will update its HorovodGlobalState with available resources. We are seeking in here to see if there is an elegant way to detect the resource change.

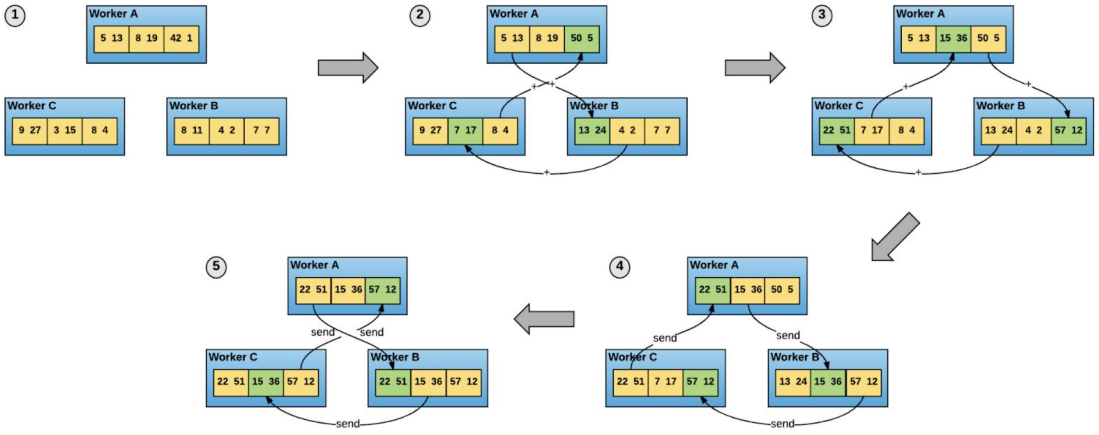
Once we detect the change of resources, we will need to adjust the behavior of gradient aggregation, like changing the batch size per allreduce, accordingly. We will be looking into how the horovod do the gradient updates and some nice pointers are in [3].

We also want this to be transparent to the application and user, so we want to find a way to report this change of behavior. The most straightforward way we are thinking of is using logging to do that. Furthermore, as stated in the paper, there is a profiling tool called Horovod Timeline that provides some graphical interface that users can look at. We are also thinking to signal the behavior change via this tool.

3. Related Work

3.1 Gradient Aggregation in Distributed Training

Ring-AllReduce (RAR) [8] design has been proposed to efficiently average gradients at any particular node and device, its mechanism is shown in the figure below. However, this bandwidth-optimal algorithm only utilizes the network in an optimal way if the tensors are large enough. Horovod [1] is built upon RAR but introduced a new gradient fusion strategy that also adopts centralized deployment mode to reduce tensor fragmentation and improve bandwidth utilization. CodedReduce [6], in advance, is proposed to develop efficient and robust gradient aggregation strategies that overcome communication bandwidth and stragglers' delays. The below figure is an example showing how RAR algorithm works.



3.2 Fault Tolerance for Distributed Learning

MPI has been criticized for its lack of support for fault tolerance. There are a series of research focusing on developing fault-aware MPI. First, a good number of rollback-recovery techniques for distributed systems have been developed so far, such as DejaVu [9], DMTCP [10]. The rollback-recovery techniques let the failed tasks can restart from a previously saved state to avoid the waste of resources. Second, User-Level Fault Mitigation (ULFM) is introduced to enable libraries and application to survive the increasing number of failures, and, subsequently, to repair the state of the MPIs world [11]. Our system adopts several useful concepts from the above work when the scheduler revokes resources from clusters and then developed a transparent rollback-recovery technique to do the resource allocation.

Another interesting research area in the failure resilience of distributed machine-learning systems is the Byzantine failures, the most general failure model. Byzantine failure may be caused by hardware issues, software bugs, network asynchrony, or other reasons. Some worker machines may behave completely arbitrarily and can send any message to the master machine that maintains and updates an estimate of the parameter vector to be learned. The Krum algorithm [7] is proposed to tolerate f Byzantine participants out of n workers for distributed SGD. While this is related to the revocation of resources by a scheduler, it is outside the scope of this problem - as Byzantine workers are not unavailable for use, but may actively cause harm by providing bad updates due to a stale model.

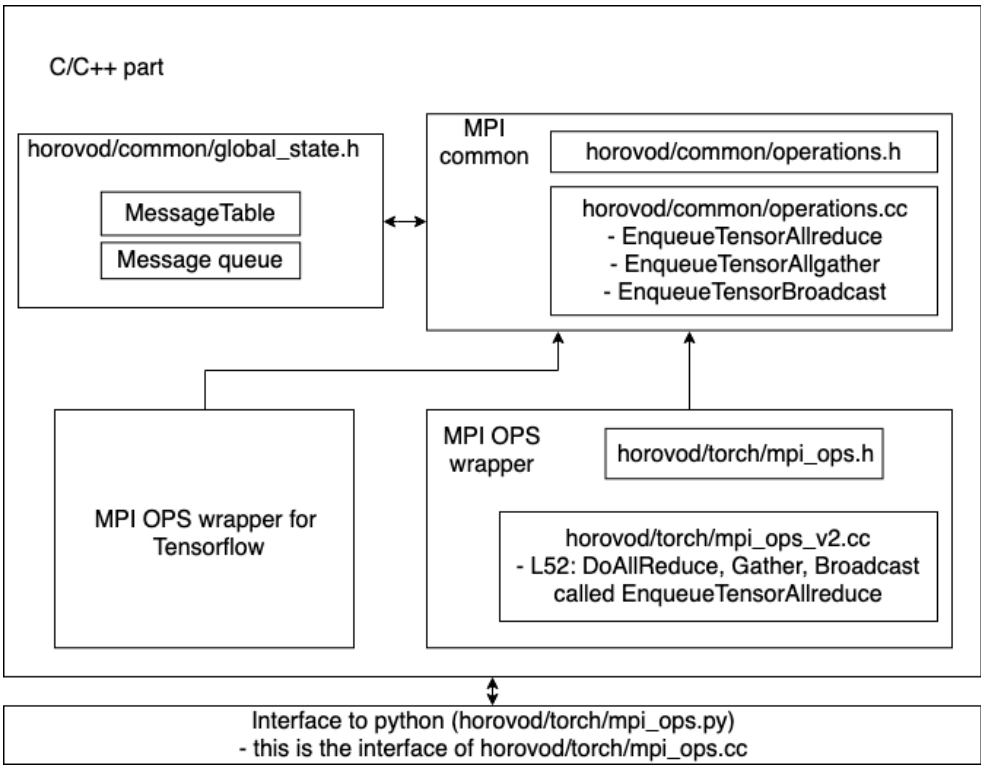
4. Design aspects

4.1 Resource change detection

To achieve transparent aggregation, first, we need to detect resource change. There are two ways to do that, one is polling at each period and the other is via an interrupt. But we think it is not suitable to do the detection by active polling on each node. Because that would require a thread to keep interacts with each of the nodes in our cluster to get the change. We think that it is fair to assume that there is a resource scheduler and that it would decide to revoke the resource based on its need. Because we don't have a resource scheduler set up here, we just assume that one of the nodes in the cluster will send a request saying that how many computing devices (GPU/CPU) need to be revoked. And we assume that each of the nodes uses the same number of computing devices to do training.

4.2 Gradient aggregation in Horovod

To implement gradient aggregation in Horovod, first, we have to understand its execution model. Below is the figure showing the important C++ classes in horovod. The core logic of Horovod is all in common/operations.cc and there are framework-specific C++ and Python wrappers.



When we run horovodrun, the running procedure is as follows:

1. run/run.py: make sure all tasks server can be ssh into and then use mpirun to run user code on all workers (which in our case is "example/pytorch_mnist.py")

2. Inside each user code, horovod.init() is called which can trace back to “common/operations.cc” . Then it called InitializeHorovodOnce() which would start a BackgroundThreadLoop() that keep calling RunLoopOnce() until the server halts. Inside RunLoopOnce() the following is performed
    a. Get message queue from global to local
    b. Master thread put names into the ready_to_reduce queue
    c. Called PerformOperation() which would invoke op_manager-> ExecuteOperation()
    d. common/ops/operation_manager.cc: use for loop to iterate through and call op->Execute() which depends on the type of OP would go corresponding function
    e. common/ops/mpi_operations.cc: All reduce Execute will call MPI_Allreduce. Finally, we arrived at our familiar MPI operations
3. The server stops

And the training procedure is as follows:

1. torch/__init__.py has DistributedOptimizer where its step method is changed to _allreduce_grad_async which is the function defined in torch/mpi_ops.py which gets actual function definition from torch/mpi_ops.cc
2. /torch/mpi_ops_v2.cc or mpi_ops.cc: called DoAllReduce, Gather, Broadcast which will call EnqueueTensorAllreduce/Gather/Broadcast
3. horovod/common/operations.cc: line 1654 EnqueueTensorAllreduce, just push tensor and message into global structure tensor_table and message queue
4. At each tick, RunLoopOnce will take what's inside the global structure and perform corresponding operations then put back the response

Once we understand the whole execution of Horovod, we realize that it is too complicated to modify the C++ implementation as it is carefully designed. So instead, we decided to modify the part of the Python wrapper of DistributedOptimizer. We adopt the changes in [3], that is we introduce a parameter update delay in the optimizer. And only when the value of delay is 0 then we call allreduce operation on gradients to get aggregated gradients from all workers, otherwise, we just decrease the delay by 1.

5. Implementation

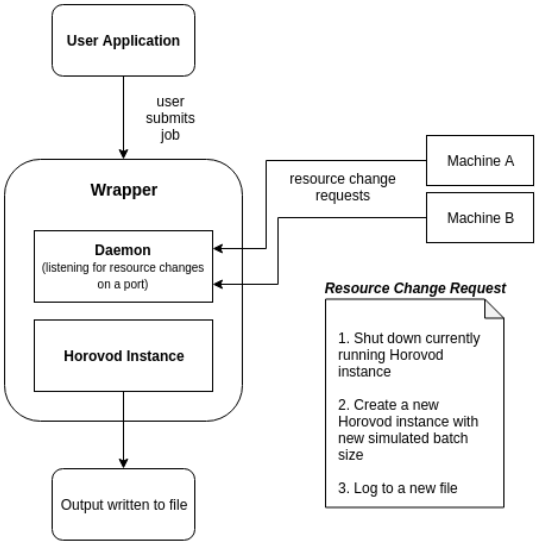


Figure 1: Project Architecture

We have implemented transparent gradient aggregation as a Python wrapper around Horovod. Users submit jobs to the wrapper, which runs them in Horovod. Detection of resource changes and application of an appropriate strategy is handled by the wrapper and requires virtually no modifications to the user application's code. (A couple required modifications are discussed in Section 5.3) The architecture of the system is shown in Figure 1.

The wrapper runs two subprocesses:

1. A daemon that binds to a socket and listens for resource change messages from machines
2. A Horovod instance running in the background and writing output to a file

5.1 Listener Daemon

This thread binds to a socket and listens for resource change requests from machines in the cluster on a port. The messages consist of a single integer with the number of compute units (CPUs in our case) that the machine wants to revoke. If the resulting resources for a given machine is <= 0, it is removed from the cluster.

In order to preserve semantics, the new number of computing resources must be a factor of the old resources (so that each machine can aggregate gradients for a whole number of steps before updating the model). Thus, if the resulting allocation is not evenly-divisible, the daemon will randomly remove resources from the other machines until either a factor has been reached or there are no resources remaining. For example, if we start with the following allocation: node0:6, node1:6, node2:6 and node1 requests to remove all 6 of its resources, an additional 3 resources must be removed from the other

machines so that a new total of 9 is achieved (18 % 9 = 0). One possible resulting allocation is: node0:5, node2:4

5.2 Horovod Instance

Upon receiving a user job, the wrapper creates a subprocess that runs Horovod with the user's application as input. After a resource change request is received and processed, the wrapper kills the Horovod subprocess and spawns a new one with the updated resource profile and gradient aggregation strategy. The Horovod instance writes to a new file for each resource update. For example, if the wrapper receives 2 resource update requests during training, the files `horovodrun_0.out, horovodrun_1.out` and `horovodrun_2.out` will be produced, where `horovodrun_0.out` contains output before any resource request, and the other two contain the data produced after the first and second resource update respectively. Standard out shows the user what commands are being used to run Horovod and prints resource changes. The output to the user from the example in Section 5.1 is shown below:

```
$ python3 submitjob.py --hosts node0 node1 node2 --cpus 6 6 6 --jobfile
./horovod/examples/pytorch_mnist.py --epochs 15

Starting daemon
Starting horovod with command: ./horovod/bin/horovodrun -np 18 -H
node0:6,node1:6,node2:6 --verbose 2 python3 ./horovod/examples/pytorch_mnist.py
--loadcp --epochs 15 --batches 1 > horovodrun_0.out
Listening for messages...
INFO: Had to revoke additional resources to satisfy the request from client.
INFO: Resource update request received from 10.10.1.2
       CPUs reduced from 18->9
       Updated batches-per-allreduce: 2
Starting horovod with command: ./horovod/bin/horovodrun -np 9 -H node0:4,node2:5
--verbose 2 python3 ./horovod/examples/pytorch_mnist.py --loadcp --epochs 15 --batches
2 > horovodrun_1.out
```

5.3 Changes Required to Application Code

A primary goal of our project was to provide an interface to enable transparent gradient aggregation with minimal changes to user application code. However, the user is required to make a few minor modifications to their code.

1. *Checkpointing:* The user should implement checkpointing in their code. But this is usually the case as the user wants to save their model along the training process. The primary benefit of transparent gradient aggregation is being able to load a checkpoint with a new resource profile rather than starting the computation over from the beginning.
2. *Batches per AllReduce:* The application must implement an argument that represents how many batches should be aggregated locally by the workers before performing the AllReduce operation (default = 1). The learning rate should be scaled by this parameter, and it should also be passed as the `backward_passes_per_step` argument to the hvd.DistributedOptimizer. This is necessary so that after a resource change, the wrapper can tell the optimizer about the new aggregation strategy.

6. Evaluation

6.1 Experiment setup

To evaluate the system, we set up a three-node cluster using Cloudlab. Each of the nodes has 4 Intel Xeon Silver 4114 CPU @ 2.20GHz, 187G of memory, and running on Ubuntu 16.04.1. (We are using the profile of assignment 2). We install horovod (0.16.1) and build it from source to use the PyTorch version. For the MPI version, we are using OpenMPI 4.0.1. For the data, we are using MNIST dataset, which consists of 70,000 images of handwritten digits (28 by 28). It is a classification problem, which has 10 classes (digit 0 to digit 9).

6.2 Results

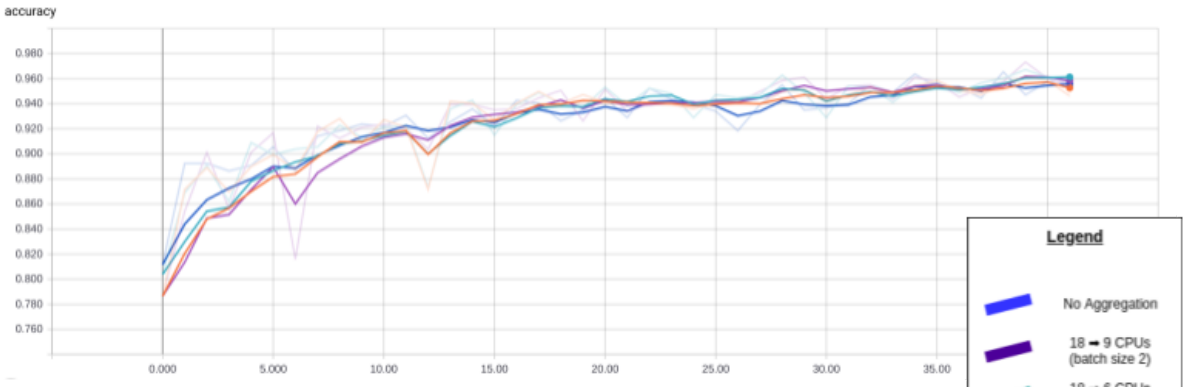


Figure 2: Accuracy comparison of simulated batch sizes

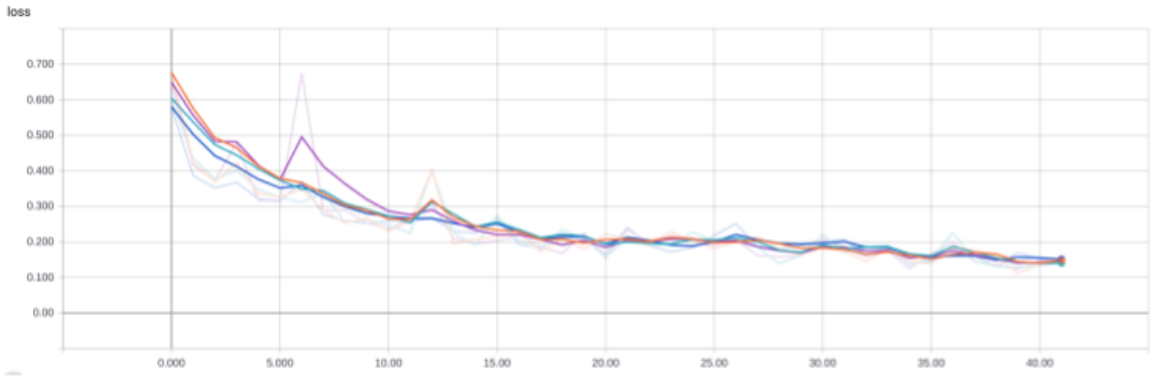


Figure 3: Loss comparison of simulated batch sizes

We measure both the accuracy and loss over 15 epochs, recorded every 18 training steps to show that convergence is the same with updated simulated batch size - demonstrating semantic equivalence. Figures 2 and 3 show the results with a batch size of 64 after resuming from a checkpoint after epoch 1.

In addition, we measure wall time. As expected, although the allocation with fewer resources eventually converges correctly, it takes longer. Figure 4 compares the wall time of accuracy and loss of the same benchmark tested above. In our tests, we experienced a minor increase in wall-clock time. Assuming no extra communication overheads, reducing the number of CPUs by factors of 2, 3, and 6, we would expect to see increases of 2x, 3x, and 6x. But in reality, the wall-clock time increased by 1.5x, 2x, and 3.5x respectively. We think it is the communication overheads that cause this difference.

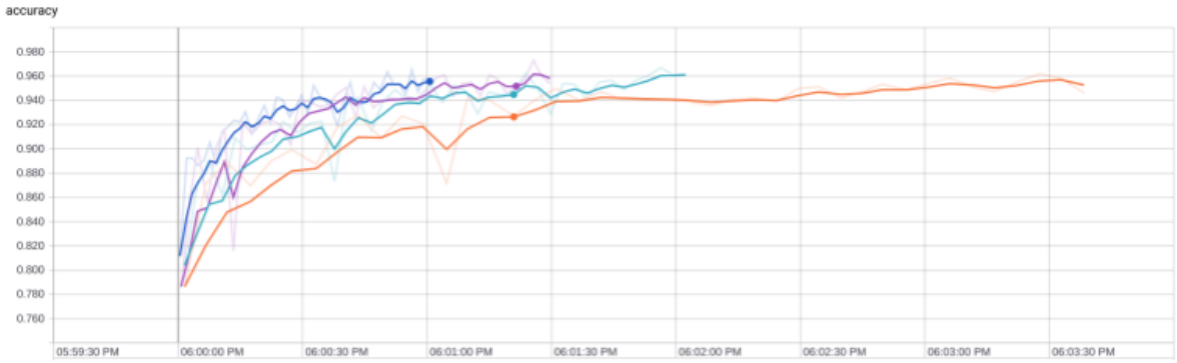


Figure 4: Wall-clock time comparison for simulated batch sizes

7. Discussion & Future Work

7.1 When to use gradient aggregation?

Gradient aggregation may not always be the best strategy when handling changing resource requirements. Currently, our implementation requires that after a request is made, the old total resources must be evenly-divisible by the new total resources. In some cases, this may result in many more resources being removed than necessary. For example, if we have a total of 20 CPUs and a machine wants to revoke access to 1 CPU, we will need to remove an additional 9 CPUs (nearly half of our total resources!) in order to produce a semantically-equivalent model. In cases like this, especially if the total runtime is relatively small, it is better to restart the computation from scratch with a new batch size that can use all available resources.

Long-running jobs are best-suited for gradient aggregation. If we have a job running for hours that receives a resource-change request after it is 95% converged, we don't want to restart the whole computation. In this case, using gradient aggregation to finish the job with a subset of the resources will likely be faster than restarting the job with better resources usage.

7.2 Future Work

We chose to focus on providing a framework for PyTorch applications since Horovod already has support for local gradient accumulation in PyTorch [3]. In the future, we would like to extend this to work with more of the frameworks supported by Horovod (Tensorflow in particular). Modifying our wrapper to support this is trivial -- all that is required is to add support for gradient aggregation for each individual framework.

8. Conclusion

In this project, we have understood Horovod in great depth in its execution model and we are able to set up a three-node cluster to run experiments. We also implement transparent gradient aggregation using a Python script wrapping around Horovod. Using our modified example user code, we are able to verify the correctness of our program.

References

[1] “Horovod”, Uber. Accessed May 8, 2019. https://github.com/horovod/horovod
[2] Hotfix of gradient aggregation on Horovod. Accessed May 8, 2019. https://github.com/aaron276h/horovod/commit/1e0c7608eccfdcceb906a6e2a0b0f48f211cfb8c
[3] PR of Horovod gradient aggregation. Accessed May 8, 2019. https://github.com/horovod/horovod/pull/546
[4] MPI API documentation. Accessed May 8, 2019. https://www.mpich.org/static/docs/v3.2/www3/
[5] MPI concepts. Accessed May 8, 2019. https://github.com/horovod/horovod/blob/master/docs/concepts.md
[6] Reisizadeh, Amirhossein, Saurav Prakash, Ramtin Pedarsani, and Amir Salman Avestimehr. "CodedReduce: A Fast and Robust Framework for Gradient Aggregation in Distributed Learning." arXiv preprint arXiv:1902.01981 (2019).
[7] Blanchard, Peva, Rachid Guerraoui, and Julien Stainer. "Machine learning with adversaries: Byzantine tolerant gradient descent." In Advances in Neural Information Processing Systems, pp. 119-129. 2017.
[8] “Bringing HPC Techniques to Deep Learning”. Accessed May 8, 2019. http://andrew.gibiansky.com
[9] Ruscio, Joseph F., Michael A. Heffner, and Srinidhi Varadarajan. "Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems." 2007 IEEE International Parallel and Distributed Processing Symposium. IEEE, 2007.
[10]Ansel, Jason, Kapil Arya, and Gene Cooperman. "DMTCP: Transparent checkpointing for cluster computations and the desktop." 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, 2009.
[11] Vishnu, Abhinav, Charles Siegel, and Jeffrey Daily. "Distributed tensorflow with MPI." arXiv preprint arXiv:1603.02339 (2016).