Transcript:

Our program separates into two stages, data preprocessing and the DPSGD training. Accordingly, the levels of parallelism we are going to implement are big data and distributed parallelization technology. To be more specific, within the data preprocessing stage, we will use Spark to process large amount of data. This is because in our pilot studies, Spark runs much faster than MapReduce. And, for DPSGD training, we will use distributed package of PyTorch with MPI backend to implement the parallelized version of the parameter update iteration, which involves gradient calculation, clipping and noise addition.

We find that there are three popular ways to implement distributed version of stochastic gradient descent: synchronous fashion, asynchronous fashion, and ring all-reduce; we plan to extend these algorithms to design distributed version of DPSGD, either centralized or decentralized. In terms of the infrastructure, since it is hard for AWS to approve our request of more than 5 g3.4xlarge instances, we currently plan to use 4 g3.4xlarge instances to run the distributed version of DPSGD, but it might change later if we are able to request more nodes.

So far we have processed our data and we have already implemented baseline sequential code. we wrote a simple multi-layer perceptron network trained with DPSGD and analyze its execution time and prediction accuracy. This model will be our baseline model and will be compared with distributed parallelled version of code in the future. In this page, we show a snapshot of the code and code profiling results. In our experiment, given specific hyperparameters, we can achieve an accuracy of 0.67. The wall time of execution is about 16 min. The CPU time is about 15 min, and the system time is about 30 seconds. We also test the scalability by changing the size of input data. The right picture presents the profiling of our code, the first column shows the number of function calls and the second column shows the total time of function calls. The most time-consuming functions are highlighted by red box, which is mainly those three nested loops in the code. We decide to implement distributed version of this part of code, start from implementing the synchronous version, and extend to asynchronous and ring all-reduce depending on the time we have. We may focus on these functions and compare the running time of these functions with parallel version of code in the future.

There are two parallelizations involved in our project. One is parallelized data processing, and the other is distributed model training through Pytorch. Both have overheads in the parallelization, and we tend to apply the following techniques to mitigate them. To better optimize spark usage, we use Spark RDD persistence as an optimization technique to save the result of RDD evaluations. We can make persisted RDD through cache() and persist() methods. Using this technique we can save the intermediate result so that we can use it further if required, which is memory efficient. It also reduces the computation overhead, comparing to other distributed computation frameworks such as mapreduce, etc.

To reduce the overhead on torch.distributed, we intend to try the following techniques:

First, to better balance the load among nodes, we will create a load balancer to redistribute the data load and assign batch size for each GPU based on their given runtime.

Second, for data partition during training, we could parallelize the data through PyTorch Distributed Sampler module, which assign each node deterministically its own portion of the data. This avoids the need of communicating the split of data across each node in each iteration, which would largely reduce the communication overhead.

Third, we will use hybrid communication methods to further reduce the overhead. Comparing to point-to-point communication described in class, collectives allow for communication patterns across all processes in a group, and a group is a subset of the existing processes.

After we completed implementing centralized distributed version of the code, we aim to further reduce the communication overhead by implementing a decentralized version of the code based on ring-allreduce algorithm.

In usual CS algorithm, we measure the numerical complexity of a problem as a function of input size, but for a learning algorithm, it does not make sense to define the input size to be the size of training set. Therefore, we give a rough analysis of number of iterations required for distributed version of DPSGD to converge in terms of our target accuracy \epsilon. It turns out that the required number of iterations is inversely linear with the number of workers, and we expect that the theoretical speed-up will be also linear in the number of workers; but since this analysis only accounts for the number of iterations of parameter updates, in practice, we would expect that the communication overhead will play an important role, and we will need to tune the number of worker nodes to achieve maximum speed-up.