Presentation Script

Introduction
The DPSGD is the state-of-art algorithm to train the privacy-preserving deep learning model. However, due to the features of the DPSGD algorithm and the huge datasets, the DPSGD algorithm is in urgent need of combining with big compute and big data technology. Our solution separates into two stages. Within the data preprocessing stage, we use Spark to process large amounts of data. For the model training stage, we specifically designed a new distributed version of DPSGD algorithm with gradient All-Reduce step. We implemented two versions of distributed parallelism of DPSGD, one is based on PyTorch Distributed Data Parallel module and another one is implemented from scratch, where multi-GPU communication is enabled by PyTorch distributed package with NCCL backend. In terms of the infrastructure, we use AWS G3 type instances with Tesla-M60 GPU to run the distributed DPSGD. In this project, we achieved better privacy efficiency trade-off.

Data Preprocessing
Since a typical application of DPSGD is to work with sensitive data, we choose a dataset which contains sensitive information. Our data comes from American Community Survey Public Use Microdata Sample (PUMS) files. It includes useful but somehow sensitive census information such as Sex, Marriage Status, College degree. It has over 1 million records and 10 features in total.
Our objective was to train a deep learning model to predict the unemployment rate based on other demographic information using DPSGD and HPC and HTC tools so that we can both protect privacy and obtain a satisfiable runtime of the algorithm.
Within the data preprocessing stage, we tried both Spark and MapReduce to process the data. The main job is to remove invalid records, check missing value, and impute data.. According to our experiments, Spark has much better performance than MapReduce. We have concluded the following reasons to use Spark. First, Spark processes data in-memory while MapReduce reads and writes data on disk. As a result, Spark runs faster than MapReduce. Second, Spark is easy to implement and has interactive modes. It makes our code extensible and flexible. In addition, Spark suits our dataset because our data fits in the Spark clusters' RAM and we don't need to sort our data which is a necessary stage in MapReduce and may add extra time for processing data.

The time spent on data preprocessing using MapReduce and Spark is shown in this slide. Only when we process a full dataset with more than 2 cores does MapReduce show its power. Otherwise, the sequential data processing code is even more efficient. This is because MapReduce contains many extra steps such as splitting and sorting which is only suitable for large data that is not even able to fit into RAM. Using MapReduce with multiple cores can help us decrease the running time of processing a full dataset, and we can observe a linear speed up with it. Compared to MapReduce, Spark runs much faster. The speedup is more than 20 when we use 2 or 3 cores. Also, the figure implies it is not wise to use more than 2 cores for our dataset because using more cores might get even worse performance due to the excessive parallelization. So the best option for data processing might be Spark with 2 cores.

Distributed Training
Now we are going to talk about how we parallelize DPSGD. We searched many literature about parallel machine learning approaches, and reviewed several design choices. We decided to use **data parallelism** with the AllReduce **approach** and **GPU-accelerated computing** to implement a distributed version of DPSGD. We give the outline of the distributed DPSGD algorithm here. There are two main differences compared with the sequential version of DPSGD: *Data Partition* and *Gradient AllReduce*. For the data partition stage, we divide the dataset into different pieces and assign it to each node. Later in the model training stage, each node will only sample batch data from its own portion of the data. This avoids the need of distributing training data across nodes during the training stage. There is no master GPU, each GPU performs identical tasks. During the forward and backward propagation, each GPU will calculate independently and process the corresponding gradient which involves clipping and noise addition. AllReduce is a combined operation of reduce and broadcast in MPI. In the *gradient AllReduce* step, all of the local gradients are averaged (i.e. reduction) and are used to update model parameters across all of the nodes (i.e. broadcast). This design ensures that the updates to model parameters are identical, therefore eliminating the need for model synchronization at the beginning of each iteration.

We first implemented one version of distributed DPSGD using PyTorch Distributed Data Parallel module with CUDA.
Distributed Data Parallel module provides high-level APIs for multi-GPU distributed training. When we wrap up our model with DistributedDataParallel, the constructor of DistributedDataParallel class will register the additional gradient aggregation functions on all the model parameters at the time of construction, so we do not need to explicitly handle gradient aggregation and parameter updates across the computational nodes during the model training process. Besides that, we use PyTorch Distributed Sampler module to implement a data sampler, which can automatically distribute data batches instead of hand-engineering data partitions.

Although PyTorch Distributed Data Parallel is capable of handling training data partition, gradient aggregation and parameter updates automatically, it also introduces a large setup overhead to accomodate a variety of situations. Since we are dealing with the specific case of DPSGD, we decide to directly implement distributed DPSGD from scratch, where multi-GPU communication is supported by PyTorch distributed library. As discussed before, the two main differences for distributed versions of DPSGD are *data partition* and *gradient AllReduce*. For the data partition stage, we divide the dataset into equal size of pieces and assign each node one of the pieces. For the gradient *AllReduce* stage, each node will send its local gradient to all other nodes, and each node will compute the gradient average and update its own parameter. The message passing figure illustration is given here. We argue that this way of implementing the AllReduce algorithm has several advantages: the computation is completely deterministic, and it's simple to implement, easy to debug and analyze. However, we note that this approach is not ideal. It sends unnecessary messages, which can increase the communication overhead. In our experiment, since we are not able to request more than 4 GPU devices from AWS, the communication overhead is not significant and we found this version of code obtained comparable performance with Code Version 1. We leave the implementation of a more fine-grained AllReduce algorithm such as Tree AllReduce, Round-robin as our future work.

Compared to the original SGD, DPSGD could achieve approximately similar test accuracy in terms of prediction. The runtime of DPSGD is significantly slower than the original SGD, and the bottleneck of the algorithm is mainly the gradient clipping and noise addition in backpropagation, as shown in the code profiling here. However, the upside of using this optimizer is that we could better protect the privacy of data. Here we have demonstrated that by using DPSGD, the model can better protect the privacy of sensitive training data from model inversion attack, which is a benchmark for measuring the privacy level of the machine learning model.

We used strong scaling to calculate GPU speedup. While keeping the overall batch size constant, each GPU will handle a smaller portion of data as the number of GPU increases. Here we can see that the time of each epoch decreases is almost proportional to the number of GPU increases. However, we haven't achieved ideally linear speedup because of data partition and communication overheads.

We also tried out different distributions of GPU clusters with 4 GPUs within each. Each cluster achieves approximately the same speedup, while the node with 4 GPU builtin achieves the best performance. By examining the code profiling, we discovered that "average_gradients" function takes the most of computation time, and the function basically perform gradient all-reduce as introduced before: Hence, we think the reason why large instance is faster is it has only intra-node communication while backpropagating, which is lower than internode communication overhead.

Money-time tradeoff is also a factor we consider when doing distributed training: g3.4 has one GPU and is the cheapest instance among all, but it is also pretty slow compared to multi-GPU clusters. In our experiment, a single node g3.16 with 4 GPUs are usually best money for value given the lower overhead involved.

Discussion:

In this project, we successfully achieved large speed up for both the data processing stage and distributed computing stage. However, as we said earlier, there is still space for improving the scalability by implementing a more fine-grained AllReduce algorithm. For the future work, we plan to implement some advanced AllReduce algorithms to further improve the performance, especially ring Allreduce, which is the current state-of-art. This change will be significant when we have access to a larger number of computational nodes.