Parallel Computing for Empirical Risk Minimization

Shusen Wang
Stevens Institute of Technology

September 30, 2019

Abstract

This lecture note describes synchronous parallel accelerated gradient descent (AGD) for empirical risk minimization ERM. We first describe AGD for solving ERM. We then show how to parallelize AGD; in particular, we assume there is a central parameter server and the data are partitioned among the worker nodes. We finally use Python to write a simulator that mimics synchronous parallel AGD.

1 Optimization for Machine Learning

Empirical Risk Minimization (ERM). Let $\mathbf{x}_1, \cdots, \mathbf{x}_n \in \mathbb{R}^d$ be the training features and $y_1, \cdots, y_n \in \mathbb{R}$ be the training labels. The data are store as a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times d}$ and vector $\mathbf{y} \in \mathbb{R}^d$. Many machine learning problems can be formulated as the empirical risk minimization (ERM):

$$\min_{\mathbf{w} \in \mathbb{R}^d} \quad \left\{ Q(\mathbf{w}; \mathbf{X}, \mathbf{y}) \triangleq \frac{1}{n} \sum_{j=1}^{n} L(\mathbf{w}; \mathbf{x}_j, y_j) + R(\mathbf{w}) \right\}. \tag{1.1}$$

Here, L is a loss function that measures the difference between the prediction and the label, and R is a regularization controlling the model complexity. For example, the loss can be

$$\begin{aligned} \text{least squares:} \quad & L(\mathbf{w}; \mathbf{x}_j, y_j) = \tfrac{1}{2}\big(\mathbf{x}_j^T \mathbf{w} - y_j\big)^2, \\ \text{logistic regression:} \quad & L(\mathbf{w}; \mathbf{x}_j, y_j) = \log\big(1 + \exp(-y_j \mathbf{x}_j^T \mathbf{w})\big). \end{aligned}$$

The regularization can be the ℓ_2-norm $\lambda \|\mathbf{w}\|_2^2$ and the ℓ_1-norm $\|\mathbf{w}\|_1$. ERM is ubiquitous in supervised learning. ERM has a nice property that its loss function can be decomposed to n terms, which makes parallel computing easy.

Accelerated Gradient Descent (AGD). Numerous numerical algorithms have been developed for solving (1.1). We describe the accelerated gradient descent (AGD) for solving (1.1) with differentiable objective function; other first-order and quasi-Newton algorithms can be similarly implemented. Define the gradients:

$$\mathbf{g}(\mathbf{w}) \triangleq \frac{\partial\, Q(\mathbf{w}; \mathbf{X}, \mathbf{y})}{\partial\, \mathbf{w}}. \tag{1.2}$$

In the beginning, $\mathbf{w}_0$ is randomly or all-zero initialized. AGD repeats the three steps: first, compute $\mathbf{g}(\mathbf{w}_t)$, namely the gradient at $\mathbf{w}_t$; second, update the momentum by $\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \mathbf{g}(\mathbf{w}_t)$; and third,

use the momentum direction to update model parameters: $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \mathbf{v}_{t+1}$. Here, $\alpha > 0$ is the learning rate and $\beta \in (0, 1)$ is the momentum parameter.

Computational Costs. Almost all the computation of AGD is spent on computing the gradient $\mathbf{g}(\mathbf{w})$, which can be written as

$$\mathbf{g}(\mathbf{w}) = \frac{1}{n} \sum_{j=1}^{n} \frac{\partial L(\mathbf{w}; \mathbf{x}_j, y_j)}{\partial \mathbf{w}} + \frac{\partial R(\mathbf{w})}{\partial \mathbf{w}}. \tag{1.3}$$

For the regularized least squares regression and logistic regression, computing $\frac{\partial L(\mathbf{w}; \mathbf{x}_j, y_j)}{\partial \mathbf{w}}$ for any j has $\mathcal{O}(d)$ time complexity. Thus, computing $\mathbf{g}(\mathbf{w})$ has at least $\mathcal{O}(nd)$ time complexity. If n and d are both big, then the per-iteration time complexity $\mathcal{O}(nd)$ of AGD can be prohibitive.[1]

2 Synchronous Parallel AGD

If we have tens or hundreds of processors, can we let them work jointly and reduce the wall-clock runtime? In this section, we study how to use parallel computing to make AGD faster (in terms of wall-clock time.)

2.1 Settings

There are numerous parallel optimization models. We focus on the following simple model and derive a parallel algorithm.

- There is one central server (aka driver) that coordinates m worker nodes. The communication is through message passing.[2] Furthermore, the workers do not communicate with each other. Every worker can communicate with the server by sending and receiving messages.
- Suppose the n samples $(\mathbf{x}_1, y_1), \cdots, (\mathbf{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$ are partitioned among m worker nodes. (The server does not store any data.) This is known as data parallelism.[3]

This model is well-known as parameter server [4].

2.2 Basic Idea

Recall that the bottleneck of AGD is at the computation of $\mathbf{g}(\mathbf{w})$. The data partition can be formally captured by the m disjoint sets: $\mathcal{S}_1, \cdots, \mathcal{S}_m$; if the sample $(\mathbf{x}_j, y_j)$ is in the k-th worker node, then $j \in \mathcal{S}_k$. The gradient $\mathbf{g}(\mathbf{w})$ in (1.3) can be rewritten as

$$\mathbf{g}(\mathbf{w}) = \frac{1}{n} \sum_{k=1}^{m} \underbrace{\sum_{j \in \mathcal{S}_k} \frac{\partial L(\mathbf{w}; \mathbf{x}_j, y_j)}{\partial \mathbf{w}}}_{\tilde{\mathbf{g}}_k(\mathbf{w})} + \frac{\partial R(\mathbf{w})}{\partial \mathbf{w}}. \tag{2.1}$$

[1] If SGD or mini-batch SGD are used instead of AGD, then the per-iteration time complexity will be lower, but the per-epoch time complexity is the same: $\mathcal{O}(nd)$.

[2] Message passing, as opposed to shared memory, means the server and worker nodes exchange data through passing messages to one another.

[3] As opposed to data parallelism, model parallelism means every worker has all the n samples, and every workers has a part of the model parameters.

Note that $\tilde{\mathbf{g}}_k(\mathbf{w}) \in \mathbb{R}^d$ depends only on $\mathbf{w}$ and the k-th worker's data. Thus, upon receiving $\mathbf{w}$, the k-th worker is able to compute $\tilde{\mathbf{g}}(\mathbf{w})$. We can equivalently write (2.1) as

$$\mathbf{g}(\mathbf{w}) \quad = \quad \frac{1}{n}\sum_{k=1}^{m} \tilde{\mathbf{g}}_k(\mathbf{w}) + \frac{\partial\, R(\mathbf{w})}{\partial\, \mathbf{w}}. \tag{2.2}$$

Note also that $\frac{\partial\, R(\mathbf{w})}{\partial\, \mathbf{w}}$ depends only on $\mathbf{w}$, and thus the server can compute it locally. The server aggregates $\tilde{\mathbf{g}}_1(\mathbf{w}), \cdots, \tilde{\mathbf{g}}_m(\mathbf{w})$ and then compute $\mathbf{g}(\mathbf{w})$ according to (2.2).

2.3 Algorithm Description

The AGD algorithm developed in Section 1 can be implemented in the following way to fit the parallel computing framework. Parallel AGD repeats the following four steps till convergence.

Broadcast. The central parameter server stores the latest model parameters, say $\mathbf{w}_t \in \mathbb{R}^d$. The server broadcasts $\mathbf{w}_t$ to the workers so that the workers will have the latest parameters. The time complexity of this step is negligible; the communication complexity is $\mathcal{O}(dm)$ words. (By organizing the workers nodes as a binary tree, the communication cost can be $\mathcal{O}(d \log m)$.)

Workers' local computation. Upon receiving $\mathbf{w}_t$, the k-th worker uses its local data (which are indexed by $\mathcal{S}_k$) to compute the *local gradient* $\tilde{\mathbf{g}}_k(\mathbf{w}_t)$:

$$\tilde{\mathbf{g}}_k(\mathbf{w}_t) \quad = \quad \sum_{j \in \mathcal{S}_k} \frac{\partial\, L(\mathbf{w}; \mathbf{x}_j, y_j)}{\partial\, \mathbf{w}}\bigg|_{\mathbf{w}=\mathbf{w}_t}. \tag{2.3}$$

This step does not need communication. For least squares and logistic regression, the time complexity is $\mathcal{O}(d\,|\mathcal{S}_k|)$.

Aggregate the local gradients. The workers send their local gradients, $\tilde{\mathbf{g}}_1(\mathbf{w}_t), \cdots, \tilde{\mathbf{g}}_m(\mathbf{w}_t)$ to the server. The server aggregates the local gradients by $\frac{1}{n}\sum_{k=1}^{m} \tilde{\mathbf{g}}_k(\mathbf{w}_t)$ and then compute the gradient $\mathbf{g}(\mathbf{w}_t)$ according to (2.2). The server needs to compute the differential $\frac{\partial\, R(\mathbf{w})}{\partial\, \mathbf{w}}\big|_{\mathbf{w}=\mathbf{w}_t}$ and sum the local gradients, which are inexpensive operations. The aggregation requires all-to-one communication of the local gradients which has $\mathcal{O}(dm)$ or $\mathcal{O}(d \log m)$ time complexity, depending on the computer network structure.

Server updates the model parameters. With $\mathbf{g}(\mathbf{w}_t)$ at hand, the server updates the momentum by $\mathbf{v}_{t+1} = \beta \mathbf{v}_t + \mathbf{g}(\mathbf{w}_t)$ and the model parameters by $\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \mathbf{v}_{t+1}$. This step does not have communication. These operations have merely $\mathcal{O}(d)$ time complexity.

2.4 Analyzing the Costs

Computational costs. As aforementioned, the computation of AGD is mostly in computing the gradient. In parallel AGD, the gradient is computated by m workers in parallel. If the work load is balanced, then every worker has a per-iteration time complexity of $\mathcal{O}(nd/m)$. The server performs inexpensive vector operations, and its computational cost is negligible.

Communication costs. Every iteration of the parallel AGD algorithm has two communications: one-to-all broadcast and all-to-one aggregation. The communication complexities is $\mathcal{O}(dm)$ words if the connections between the server and workers form a "star graph"; it is $\mathcal{O}(d \log m)$ if the network connection is a binary tree structure.

What is the time cost of one communication? It mainly depends on the bandwidth (denote B) and latency[4] (denote L) of the network and the communication complexity (denote C) of the algorithm. Then the time cost is approximately $\frac{C}{B} + L$.

Synchronization costs. This algorithm is bulk synchronous, the server will wait for all the workers to complete before starting the next iteration. An obvious shortcoming is that everyone must wait for the slowest worker to complete; see the illustration in Figure 1. If a worker fails (due to hardware or software failure) and restarts, then the time cost of this iteration will double. This worker is called straggler, and this phenomenon is known as the straggler effect. It is exacerbated as the number of workers increases. If a worker fails with probability 1% and there are 50 workers, then with probability 60%, there will be at least one node fail. Asynchronous algorithms, e.g., [8], are typically faster than the synchronous.

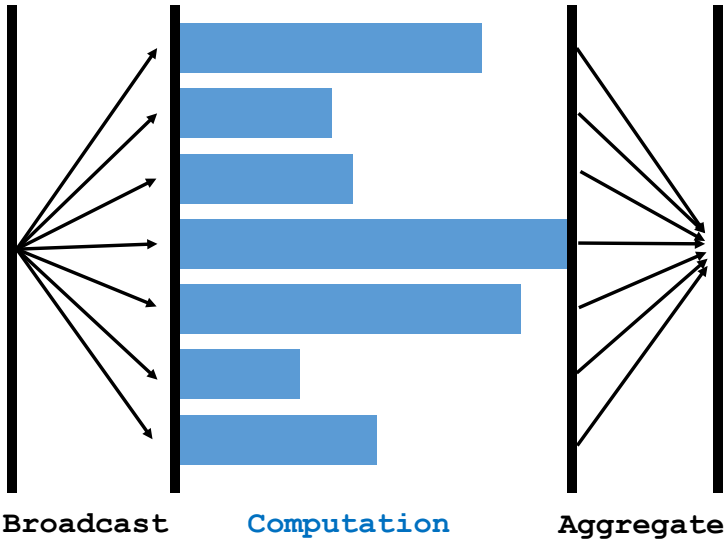

Figure 1: Illustration of synchronization.

Speedup ratio. Ideally, since the computation of gradient is performed by m workers in parallel, the wall-clock runtime can be $\frac{1}{m}$ of the single-machine AGD. If it is the case, then we say the speedup ratio is m. Unfortunately, due to the other overhead, the speedup ratio is always smaller than m; when m is large, the growth rate of speedup ratio is slow. See Figure 2 for the illustration.

[4]When the server sends a message to a worker, the work does not receive the message immediately, even if the package has only one bit. Network latency is an expression of how much time it takes for a packet of data to get from one designated point to another.

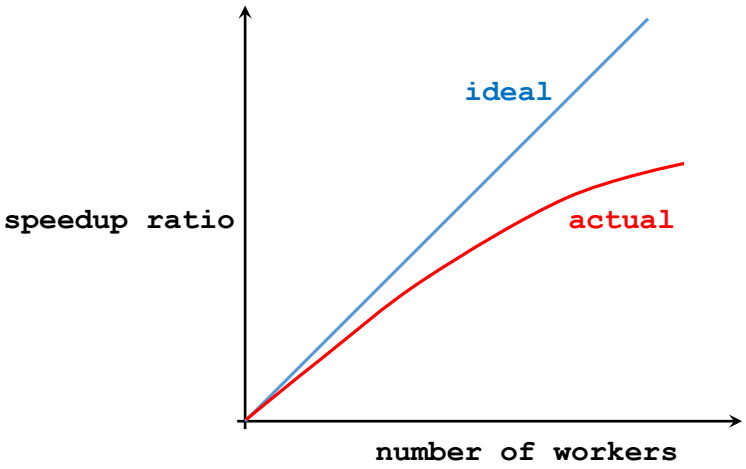

Figure 2: Ideal and actual speedup ratios.

3 Python Implementation for Logistic Regression

Let us implement a simulator that mimics synchronous parallel AGD for the ℓ_2-norm regularized logistic regression:

$$\min_{\mathbf{w}\in\mathbb{R}^d} \left\{ Q(\mathbf{w};\mathbf{X},\mathbf{y}) \triangleq \frac{1}{n}\sum_{j=1}^{n} \log\left(1+\exp\left(-y_j\mathbf{x}_j^T\mathbf{w}\right)\right) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2 \right\}. \tag{3.1}$$

Our code does not actually perform parallel computing, but it will help the reader understand how parallel AGD works. To actually make use of multiple processors, the reader needs to learn Message Passing Interface (MPI), Apache Spark [12], Ray [7], or other software system.

3.1 Worker Node

A worker node receives a partition of data in the beginning and holds the data throughout. Let $\mathcal{S}$ ($|\mathcal{S}| = s$) index the samples held by this worker. Its loss() function computes the local loss

$$\tilde{L}(\mathbf{w}) = \sum_{j\in\mathcal{S}} \log\left(1+\exp(-y_j\mathbf{x}_j^T\mathbf{w})\right).$$

Its gradient() function the local gradient

$$\tilde{\mathbf{g}}(\mathbf{w}) = \sum_{j\in\mathcal{S}} \frac{-y_j\mathbf{x}_j}{1+\exp(y_j\mathbf{w}^T\mathbf{x}_j)}.$$

```
class Worker:
    def __init__(self, x, y):
        self.x = x # s-by-d local feature matrix
        self.y = y # s-by-1 local label matrix
```

```
        self.s = x.shape[0] # number of local samples
        self.d = x.shape[1] # number of features
        self.w = numpy.zeros((d, 1)) # d-by-1 model parameter vector

    # Set the model parameters to the latest
    def set_param(self, w):
        self.w = w

    # Compute the local loss
    def loss(self):
        yx = numpy.multiply(self.y, self.x) # s-by-d matrix
        yxw = numpy.dot(yx, self.w) # s-by-1 matrix
        vec1 = numpy.exp(-yxw) # s-by-1 matrix
        vec2 = numpy.log(1 + vec1) # s-by-1 matrix
        return numpy.sum(vec2) # loss function

    # Compute the local gradient
    def gradient(self):
        yx = numpy.multiply(self.y, self.x) # s-by-d matrix
        yxw = numpy.dot(yx, self.w) # s-by-1 matrix
        vec1 = numpy.exp(yxw) # s-by-1 matrix
        vec2 = numpy.divide(yx, 1+vec1) # s-by-d matrix
        g = -numpy.sum(vec2, axis=0).reshape(self.d, 1) # d-by-1 matrix
        return g
```

3.2 Server

The server maintains and updates the model parameters; it does not store the data samples. In the init() function, we let the server know that there are m workers, n training samples, and d features. After the workers complete computing the local losses $\tilde{L}_1(\mathbf{w}), \cdots, \tilde{L}_m(\mathbf{w})$ and gradients $\tilde{\mathbf{g}}_1(\mathbf{w}), \cdots, \tilde{\mathbf{g}}_m(\mathbf{w})$, the server's aggregate() function computes

$$\sum_{k=1}^{m} \tilde{\mathbf{g}}_k(\mathbf{w}) \qquad \text{and} \qquad \sum_{k=1}^{m} \tilde{L}_k(\mathbf{w}).$$

The gradient() function computes the full gradient

$$\mathbf{g}(\mathbf{w}) \;=\; \frac{1}{n} \sum_{k=1}^{m} \tilde{\mathbf{g}}_k(\mathbf{w}) + \lambda \mathbf{w}.$$

The objective() function computes

$$Q(\mathbf{w}; \mathbf{X}, \mathbf{y}) \;=\; \frac{1}{n} \sum_{k=1}^{m} \tilde{L}_k(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2,$$

which helps monitor the progress of the optimization. The agd() function updates momentum and then the model parameters by

$$\mathbf{v} \longleftarrow \beta \mathbf{v} + \mathbf{g}(\mathbf{w}) \qquad \text{and} \qquad \mathbf{w} \longleftarrow \mathbf{w} - \alpha \mathbf{v},$$

which completes an iteration.

```
class Server:
    def __init__(self, m, n, d):
        self.m = m # number of worker nodes
        self.n = n # number of training samples
        self.d = d # number of features
        self.w = numpy.zeros((d, 1)) # # d-by-1 model parameter vector
        self.g = numpy.zeros((d, 1)) # d-by-1 gradient
        self.v = numpy.zeros((d, 1)) # d-by-1 momentum
        self.loss = 0 # loss function value
        self.obj = 0 # objective function value

    def broadcast(self):
        return self.w

    # Sum the gradients and loss functions evaluated by the workers
    # Args:
    #   grads: a list of d-by-1 vectors
    #   losses: a list of scalars
    def aggregate(self, grads, losses):
        self.g = numpy.zeros((self.d, 1))
        self.loss = 0
        for k in range(self.m):
            self.g += grads[k]
            self.loss += losses[k]

    # Compute the gradient (from the loss and regularization)
    def gradient(self, lam):
        self.g = self.g / self.n + lam * self.w

    # Compute the objective function (sum of loss and regularization)
    def objective(self, lam):
        reg = lam / 2 * numpy.sum(self.w * self.w)
        self.obj = self.loss / self.n + reg
        return self.obj

    # Update the model parameters using accelerated gradient descent
    # Args:
    #   alpha: learning rate (step size)
    #   beta: momentum parameter
    def agd(self, alpha, beta):
        self.v *= beta
        self.v += self.g
        self.w -= alpha * self.v
```

3.3 Initialization

The following function create one server and a list of m workers. We are given n samples which are stored in $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{y} \in \mathbb{R}^n$. We partition the n samples to m parts and use each part to create one worker.

```python
import math

# Create a server and m worker nodes
def create_server_workers(m, x, y):
    n, d = x.shape
    s = math.floor(n / m)
    server = Server(m, n, d)
    workers = []

    for i in range(m):
        indices = list(range(i*s, (i+1)*s))
        worker = Worker(x[indices, :], y[indices, :])
        workers.append(worker)

    return server, workers
```

We use $m = 4$ workers and apply the function create_server_workers. The returned are server (an object) and workers (a list of objects).

```python
m = 4 # number of worker nodes
server, workers = create_server_workers(m, x_train, y_train)
```

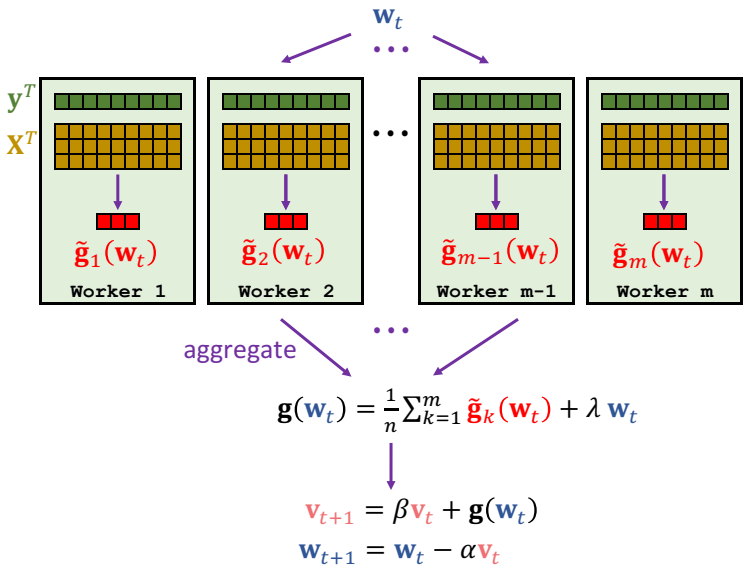

Figure 3: Illustration of parallel AGD.

3.4 Performing Parallel AGD

With the server and workers set up, we are ready to run the parallel AGD algorithm. The following code executes the parallel AGD algorithm described in Section 2.3. We illustrate the procedure in

Figure 3

```
lam = 1E-6 # regularization parameter
alpha = 1E-1 # learning rate
beta = 0.9 # momentum parameter
max_epoch = 50 # number of epochs

for t in range(max_epoch):
    # step 1: broadcast
    w = server.broadcast()
    for i in range(m):
        workers[i].set_param(w)

    # step 2: workers' local computations
    grads = []
    losses = []
    for i in range(m):
        g = workers[i].gradient()
        grads.append(g)
        l = workers[i].loss()
        losses.append(l)

    # step 3: aggregate the workers' outputs
    server.aggregate(grads, losses)

    # step 4: server update the model parameters
    server.gradient(lam) # compute gradient
    obj = server.objective(lam) # compute compute objective function
    print('Objective function value = ' + str(obj))
    server.agd(alpha, beta) # updates the model parameters
```

4 Problems

The readers are encouraged to write simulators for the following algorithms.

4.1 Federated Averaging

Federated Averaging (FedAvg) [6] is similar to the parallel gradient descent (GD): FedAvg has a central parameter server, the data are partitioned among the worker nodes, and it is synchronous. Instead of performing one GD or stochastic gradient descent (SGD), every worker locally run multiple GDs (or SGDs) and return the descending direction to the server. In this way, every worker performs more computation, but the overall communication is reduced. If communication is the bottleneck, then FedAvg is more practical than parallel GD.

Here we briefly describe the algorithm. FedAvg repeats the following steps till convergence.

- The server broadcasts $\mathbf{w}$ to all the workers.
- Upon receiving $\mathbf{w}$, a worker (say the k-th) uses its local data to locally performs q (≥ 1) GDs (or SGDs). Let $\tilde{\mathbf{w}}_k$ be the result of the q local GDs (or SGDs). The the ascending direction

computed by the k-th worker is

$$\tilde{\mathbf{p}}_k \;=\; \mathbf{w} - \tilde{\mathbf{w}}_k.$$

- The server aggregates $\tilde{\mathbf{p}}_1, \cdots, \tilde{\mathbf{p}}_m$:

$$\mathbf{p} \;=\; \frac{1}{m}\sum_{k=1}^{m} \tilde{\mathbf{p}}_k.$$

The server then update the model parameters by

$$\mathbf{w} \;\longleftarrow\; \mathbf{w} - \alpha\mathbf{p}$$

and decrease the learning rate α.

In the case of $q = 1$, FedAvg is the same to parallel GD (or SGD).

4.2 Decentralized Optimization

Decentralized optimization allows for worker nodes jointly learn a model without have a central server. Every worker is connected to some other workers; see the illustration in Figure 4. Many decentralized algorithms have been developed for solving this problem [1, 11, 9, 2, 5, 3, 10]. A typical decentralized algorithms [5] works in this way: a node collects its neighbors' model parameters, weighted averages its neighbors' and its own parameters, and performs an (stochastic) gradient descent to update its local parameters.

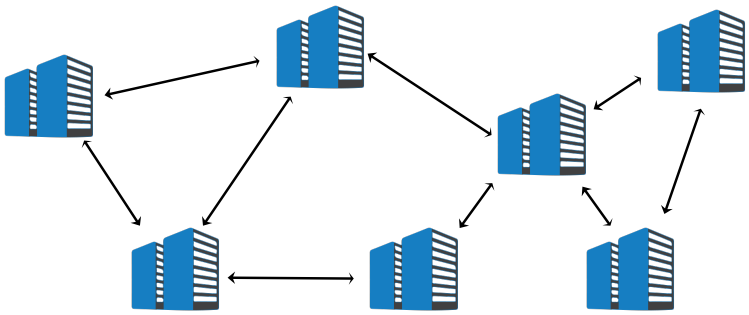

Figure 4: Illustration of a decentralized system.

References

[1] Pascal Bianchi, Gersende Fort, and Walid Hachem. Performance of a distributed stochastic approximation algorithm. *IEEE Transactions on Information Theory*, 59(11):7405–7418, 2013.

[2] Igor Colin, Aurélien Bellet, Joseph Salmon, and Stéphan Clémen**c**con. Gossip dual averaging for decentralized optimization of pairwise functions. *arXiv preprint arXiv:1606.02421*, 2016.

[3] Guanghui Lan, Soomin Lee, and Yi Zhou. Communication-efficient algorithms for decentralized and stochastic optimization. *Mathematical Programming*, pages 1–48, 2017.

[4] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[5] Xiangru Lian, Ce Zhang, Huan Zhang, Cho-Jui Hsieh, Wei Zhang, and Ji Liu. Can decentralized algorithms outperform centralized algorithms? a case study for decentralized parallel stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[6] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics (AISTATS)*, 2017.

[7] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, and Ion Stoica. Ray: A distributed framework for emerging {AI} applications. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 561–577, 2018.

[8] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems (NIPS)*, 2011.

[9] Benjamin Sirb and Xiaojing Ye. Consensus optimization with delayed and stochastic gradients on decentralized networks. In *2016 IEEE International Conference on Big Data (Big Data)*, pages 76–85. IEEE, 2016.

[10] Hanlin Tang, Xiangru Lian, Ming Yan, Ce Zhang, and Ji Liu. D2: Decentralized training over decentralized data. *arXiv preprint arXiv:1803.07068*, 2018.

[11] Kun Yuan, Qing Ling, and Wotao Yin. On the convergence of decentralized gradient descent. *SIAM Journal on Optimization*, 26(3):1835–1854, 2016.

[12] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.