ECE 901: Large-scale Machine Learning and Optimization **Spring 2018**

Lecture 13 — March 6

Lecturer: Dimitris Papailiopoulos *Scribe: Yuhua Zhu & Xiaowu Dai*

Note: These lecture notes are still rough, and have only have been mildly proofread.

In this note, we investigate stragglers in the synchronous distributed optimization and follow by discussing how to lift synchronization berries, which is exempted by HogWild in [1]. Recall the objective function takes the form

$$\min_{\mathbf{w}} \frac{1}{n} \sum_{i=1}^{n} l(\mathbf{w}; \mathbf{z}_i)$$

where $l(\mathbf{w}; \mathbf{z}_i)$ is the loss for data z_i. The idea of stochastic gradient descent (SGD) is randomly sampling a data point and then doing a local optimization. More specifically, SGD updates the parameter estimation by

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \cdot \nabla l(\mathbf{w}_k; \mathbf{z}_{i_k}),$$

where $\mathbf{z}_{i_k}$ is the randomly selected data point at step k. Although SGD achieves improvement in speeds comparing with gradient descents for some machine learning tasks, SGD still take 10+ days on large data sets (see, e.g., [2]). Therefore, it is significantly important to speed up SGD.

13.1 Scaling up SGD and stragglers

The minibatch SGD is a popular algorithm used in synchronous computation for scaling up SGD. Figure 13.1 illustrates the synchronous computation, where the master node stores the model and the worker i compute the gradient

$$\mathbf{g}_i = \sum_{i \in \mathcal{S}_i} \nabla l((\mathbf{x}_i, y_i); \mathbf{w}), \quad i = 1, \ldots, P,$$

then workers send back the calcuated gradients to the master node for updating the model:

$$\mathbf{w} = \mathbf{w} - \gamma \sum_{i=1}^{P} \mathbf{g}_i.$$

Minibatch SGD repeats this distributed iterations until we are happy with the model, for example, the parameters do not change much anymore.

In the synchronous computation, the ideal compute time per node is $O(\text{total time}/P)$. But there are a lot of randomness in practice, for example, the communication between

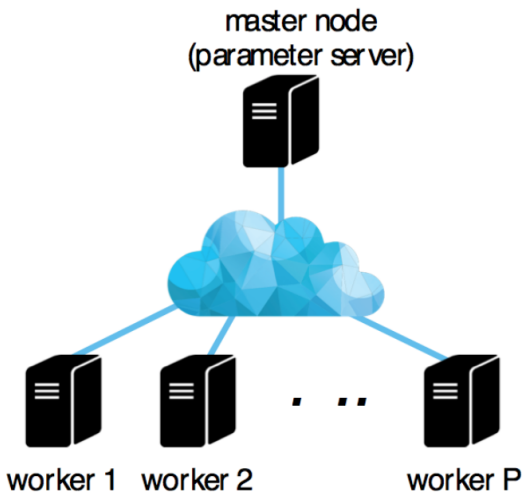

Figure 13.1. Algorithm of choice – minibatch SGD

the storage and computation delays, node failures, and so on. To analyze the synchronous algorithm, we assume the time per node is a random variable:

$$X = \text{constant} + \exp(\lambda), \tag{13.1}$$

where $\exp(\lambda)$ represents an exponential random variable with parameter λ. Denote by $X_{(i)}$ the ith fastest node running time.

Lemma 13.1.

$$\mathbb{E}\{X_{(i)}\} = 1 + \frac{1}{\lambda} \sum_{n-i+1}^{n} \frac{1}{i}.$$

The proof of this lemma can be done by order statistics in elementary probability books. We can see that the slowest node is $\frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = O(\log n)$ times slower than the fastest node. In Figure 13.2, we show the simulation results for the running time represented by random variables in (13.1), where the constant is fixed at 1 and $\lambda = 0.5$. Hence, the slowest nodes are stragglers for distributed ML.

Figure 13.3 shows a real experiment with data CIFAR-10, 148 worker nodes and 1 parameter server, where the results are measured on Amazon AWS. We can see that the slowest workers significantly slow down the speed of the whole task.

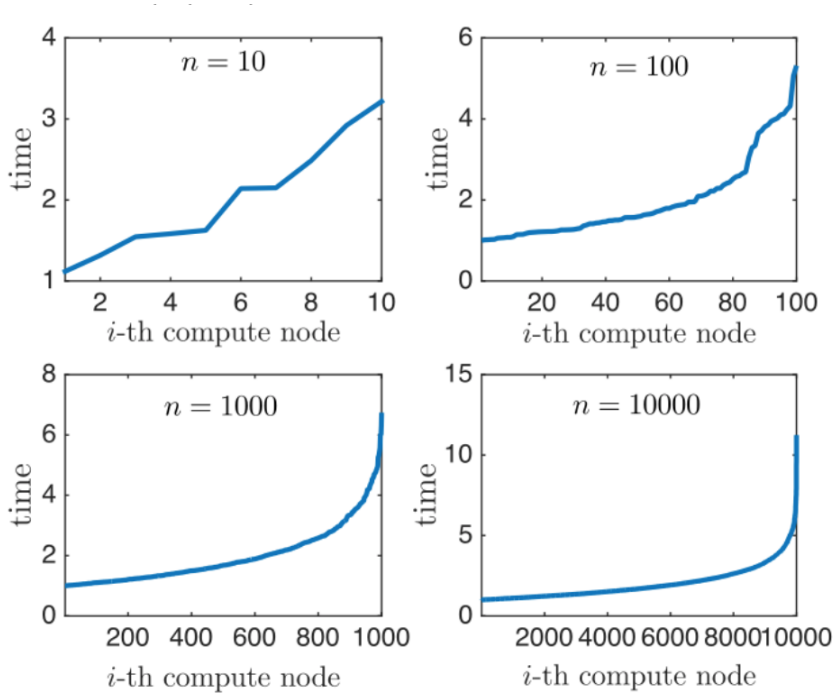

Figure 13.2. Simulation for running time represented by $X(t) = 1 + \exp(0.5)$ with different sample sizes.

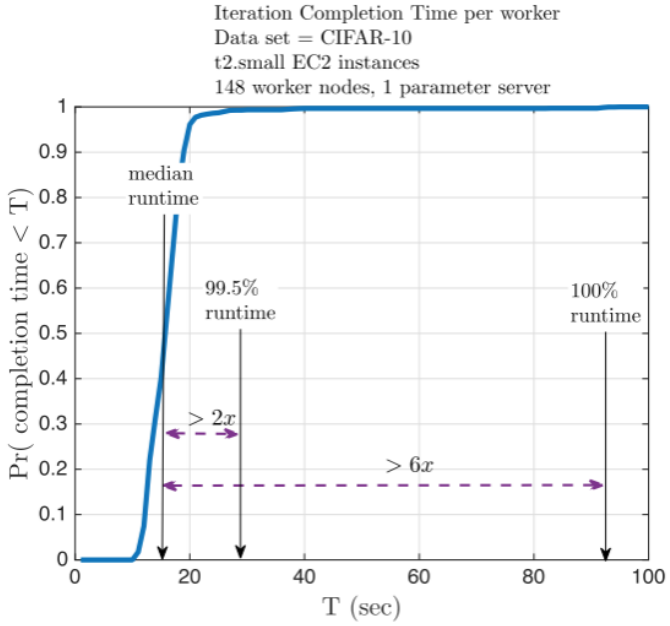

Figure 13.3. Experiment with CIFAR-10, 148 worker nodes and 1 parameter server.

13.2 Asynchronous SGD on sparse functions

We consider the sparse functions

$$f(x) = \sum_{e \in \mathcal{E}} f_e(x_e),$$

where e represents the subset of variables that f_e depends on. The sparse functions arise in many applications, for example, matrix factorization, graph cuts, graph or text classification, topic modeling among others.

The SGD on sparse functions are performed by four steps. First, pick a random data point s_k. Second, can read variables associated with s_k. Third, compute gradient of the local loss $\nabla f_{s_k}(x)$. Last, update the model through

$$x \leftarrow x - \gamma_k \cdot \nabla f_{s_k}(x).$$

Figure 13.4 gives an example of doing SGD on sparse functions.

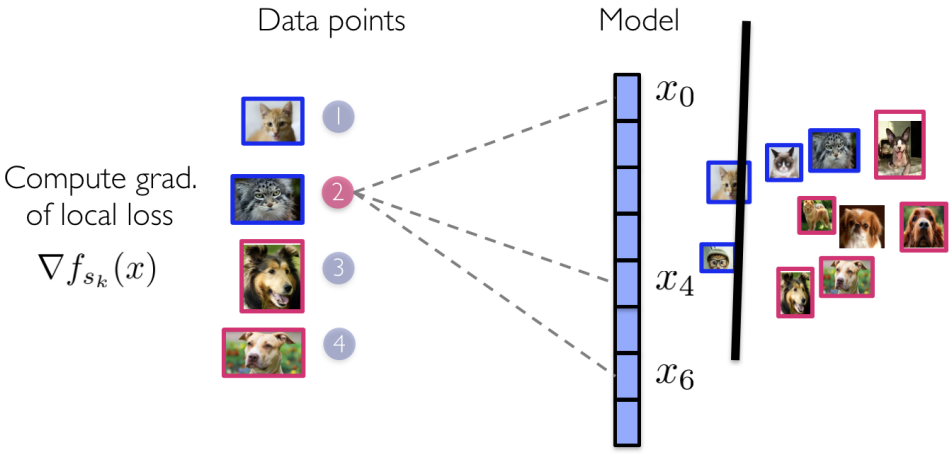

Figure 13.4. An example of SGD on sparse functions.

13.3 Parallelizing Sparse SGD on shared memory architectures

In this section, we are going to discuss about how to parallelize sparse SGD on shared memory architectures. Basically, by some priori estimates, we knows that the each sample function only depends on a subset of all variables. Assume we have a machine with multiple CPU, we assign each CPU to do SGD for a subset of the samples, and, each CPU can get

access to the model. When the sample function in each CPU has no common variables, it is equivalent to do 2 serial iterations, therefore it can speed up the simulation (as showed in Figure 13.5).

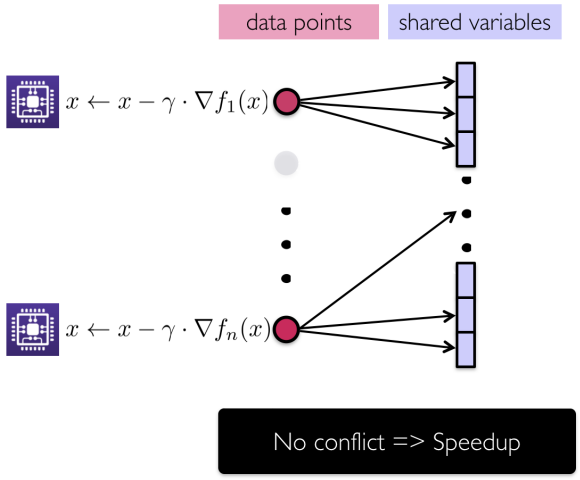

Figure 13.5.

However, when the sample function shared the same variables, they will conflict with each other when they try to get access to the model (as showed in Figure 13.6). In this case, there are two possible ways, one is let the two CPU coordinate with each other by locking the shared variables, so that all the sample function will have effective and equitable access to memory resources. There are many researches on this direction, see for example [3, 4, 5]. However, it is time consuming because different CPU will have different delays, so in each update, you need to wait for the slowest one. Basically, one can only start the next update until every CPU get to the same page.

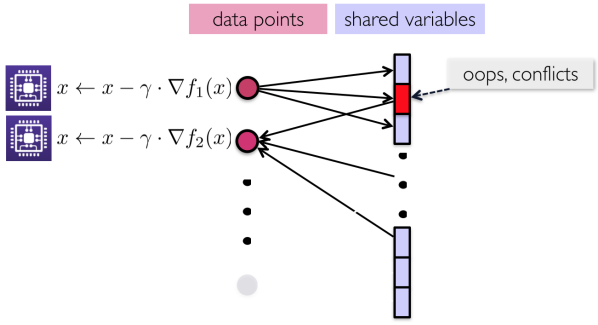

Figure 13.6.

Another approach is Lock-free. In other word, we don't care about whether the sample functions will get access to different models, just let them run parallel lock-free SGD without synchronization. This algorithm is first introduced by [1] (As showed in Figure 13.7). It is

called HOGWILD! which allows processors access to shared memory with the possibility of overwriting each other's work. It is showed in their paper that when the associated optimization problem is sparse, then HOGWILD! achieves a nearly optimal rate of convergence.

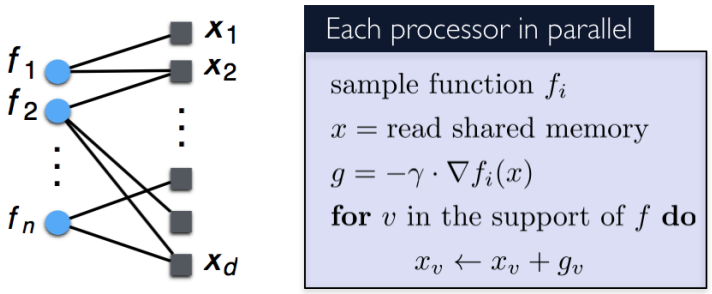

Figure 13.7.

Apparently, this algorithm will be much more speed up compared to the above one (as showed in 13.8). However, the challenge is whether this algorithm converge and how fast does it converge?

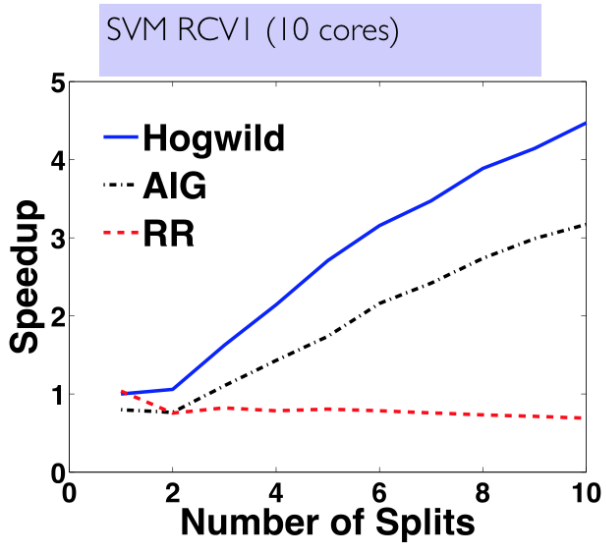

Figure 13.8.

For example, processor 1 and 2 have common variable ω_2 (as showed in Figure 13.9). In each processor, what they are doing is the following,

$$\text{Processor 1:} \quad x_2^{k+1} = x_2^k - \gamma \partial_2 f_1(\vec{x}_k); \tag{13.2}$$

$$\text{Processor 2:} \quad x_2^{k+1} = x_2^k - \gamma \partial_1 f_1(\vec{x}_k). \tag{13.3}$$

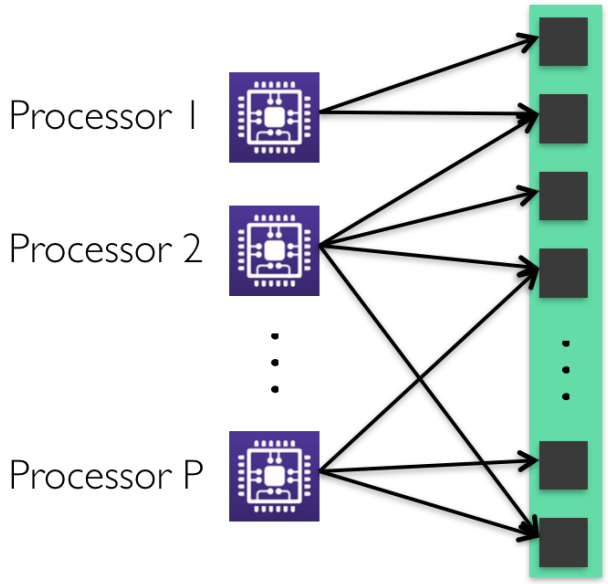

Figure 13.9.

It is possible that when Processor 1 is updating x_2^{k+1}, Processor 2 have already updated x_2^{k+1}, so when Processor 1 return its x_2^{k+1}, the results from Processor 2 is overwritten. So the analysis of Asynchronous Lock-free Algorithm is incompatible with classic SGD.

13.4 General framework for asynchronous lock-free algorithm

Since the seminar work in [1], there are many follow-up works on asynchronous algorithms in different applications. Noteworthy, [6] forwarded a Perturbed Iterate Analysis for Asynchronous Stochastic Optimization. The main idea is the following, we see the updates $\hat{x}_k$ as a noisy iterates, so after T processed samples, one has,

$$x_0 - \gamma \cdot \nabla f_{s_0}(\hat{x}_0) - \cdots - \gamma \cdot \nabla f_{s_{T-1}}(\hat{x}_{T-1}). \tag{13.4}$$

Basically, one can see HogWild as a noisy SGD. However, we don't know what $\hat{x}_k$ actually is and how strong is this noise. By elementary analysis using m-strong convexity assumption on f, one has

$$\begin{aligned}\mathbb{E}\{\|x_{k+1} - x^*\|^2\} \leq & (1 - \gamma \cdot m) \cdot \mathbb{E}\{\|x_k - x^*\|^2\} + \underbrace{\gamma^2 \cdot \mathbb{E}\{\|\nabla f_{s_k}(\hat{x}_k)\|^2\}}_{\text{I}} \\ & \underbrace{2\gamma m \cdot \mathbb{E}\{\|x_k - \hat{x}_k\|^2\} + 2\gamma \cdot \mathbb{E}\{\langle x_k - \hat{x}_k, f_{s_k}(\hat{x}_k)\rangle\}}_{\text{II}}.\end{aligned} \tag{13.5}$$

So if $\text{II} = O(\gamma^2 M^2)$, then the noisy part II can be absorbed by I. Then Noisy SGD gets the same rates as SGD up to multiplication constants.

Therefore, now the question left is whether asynchrony noisy really as small as $O(\gamma^2 M^2)$?

Bibliography

[1] Recht, B, Ré, C, Wright, S and Niu, F (2011). Hogwild: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*. 693–701.

[2] Coleman, C, Narayanan, D, Kang, D, Zhao, T, Zhang, J, Nardi, L, Bailis, P, Olukotun, K, Ré, C and Zaharia, M (2017). DAWNBench: An End-to-End Deep Learning Benchmark and Competition. *Advances in neural information processing systems*.

[3] Chazan, D and Willard, M (1969). Chaotic relaxation. *Linear algebra and its applications*. **(2)**, 199–222.

[4] Tsitsiklis, J.N., Bertsekas, D.P. and Michael A (1986). Distributed asynchronous deterministic and stochastic gradient optimization algorithms. *IEEE transactions on automatic control* . **(9)**, 803–812.

[5] Bertsekas, D.P. and Tsitsiklis, J.N. (1989). Parallel and distributed computation: numerical methods (Vol. 23). Englewood Cliffs, NJ: Prentice hall.

[6] Mania, H., Pan, X., Recht, B., Ramchandran, K. and Jordan, M.I. (2015). Perturbed iterate analysis for asynchronous stochastic optimization. arXiv preprint arXiv:1507.06970.